Advanced Computer Vision HW1

Ricky Yuan
rickyy@andrew.cmu.edu

September 19, 2024

1 Theory

1.1 Planar Homographies as a Warp

Q1.1 Define a plane: $z = 0$, let $x_\pi = (X, Y)$, which is a point on the plane Π. Let $\mathbf{x}_i = (x_i, y_i)$, which is some points being projected to camera view C_i by projection matrix $\mathbf{P}_i$. We have

$$\lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = K_1 P_1 \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = K_1 P_1' \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

where K_i is the intrinsic matrix and $\mathbf{P}_i'$ is $\mathbf{P}_i$ without the third column. Let's define $\mathbf{K}_1\mathbf{P}_1' = \mathbf{H}_1$, which is the homography from the plane Π to camera view C_1.
Similarly, we can derive the equation for camera view C_2. Combine the two equations, we have

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

Therefore, we prove the existence of $\mathbf{H} = \mathbf{H}_2\mathbf{H}_1^{-1}$

1.2 The Direct Linear Transform

Q1.2

1. 8
2. 4
3. For each point pair, $\mathbf{x}_1^i \equiv H x_2^i$, where $\mathbf{x}_1^i = [x_1^i, y_1^i, 1]^T$ and $\mathbf{x}_2^i = [x_2^i, y_2^i, 1]^T$ are the corresponding points in two images, the homography relationship can be written as:

$$\begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = H \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix}$$

Expanding this gives three equations. However, since the system is homogeneous, we can rearrange the terms to eliminate the scaling factor, giving us two independent equations per point pair. For each point pair (x_1^i, y_1^i) and (x_2^i, y_2^i) , we can derive the following two equations:

$$x_1^i(h_7 x_2^i + h_8 y_2^i + h_9) = h_1 x_2^i + h_2 y_2^i + h_3$$
$$y_1^i(h_7 x_2^i + h_8 y_2^i + h_9) = h_4 x_2^i + h_5 y_2^i + h_6$$

Rearranging these gives us two linear equations in terms of the elements of h :

$$-x_2^i h_1 - y_2^i h_2 - h_3 + x_1^i x_2^i h_7 + x_1^i y_2^i h_8 + x_1^i h_9 = 0$$
$$-x_2^i h_4 - y_2^i h_5 - h_6 + y_1^i x_2^i h_7 + y_1^i y_2^i h_8 + y_1^i h_9 = 0$$

Thus, we can derive A_i as the following matrix:

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

4. Trivial solution would be $\mathbf{h} = \mathbf{0}$.
   Matrix $\mathbf{A}$ is not full rank. If so, the null space of $\mathbf{A}$ would be $\mathbf{0}$, causing the only solution of h to be a zero vector. If $\mathbf{A}$ is not full rank, meaning some non-zero vector exists in the null space of $\mathbf{A}$, and the corresponding singular value would be 0.

1.3 Using Matrix Decompositions to calculate the homography

1.4 Theory Questions

Q1.4.1 For camera 1 and 2, we can simplify their relation to the 3D points as follows:

$$\mathbf{x}_1 = \mathbf{K}_1[\mathbf{I} \mid \mathbf{0}]\mathbf{X} = \mathbf{K}_1\mathbf{X}$$
$$\mathbf{x}_2 = \mathbf{K}_2[\mathbf{R} \mid \mathbf{0}]\mathbf{X} = \mathbf{K}_2\mathbf{R}\mathbf{X}$$

Combine the two equation, we have this relationship:

$$\mathbf{x}_1 = \mathbf{K}_1\mathbf{X} = \mathbf{K}_1\mathbf{R}^{-1}(\mathbf{R}\mathbf{X}) = \mathbf{K}_1\mathbf{R}^{-1}\mathbf{K}_2^{-1}\mathbf{x}_2$$

Which implies $\mathbf{H} = \mathbf{K}_1\mathbf{R}^{-1}\mathbf{K}_2^{-1}$.

Q1.4.2 Suppose we have the points $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ correspond to the original camera position, camera rotating θ degree and camera rotating 2θ degree, respectively. We have the following relationship:

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{K}\mathbf{X} \\ \mathbf{x}_2 &= \mathbf{K}\mathbf{R}\mathbf{X} \\ \mathbf{x}_3 &= \mathbf{K}\mathbf{R}\mathbf{R}\mathbf{X} \end{aligned}$$

We can use similar technique to question Q1.4.1. Let $\mathbf{x}_1 = \mathbf{H}_\theta \mathbf{x}_2$ and $\mathbf{x}_1 = \mathbf{H}_{2\theta}\mathbf{x}_3$ We have:

$$\begin{aligned} \mathbf{H}_\theta &= \mathbf{K}\mathbf{R}^{-1}\mathbf{K}^{-1} \\ \mathbf{H}_{2\theta} &= \mathbf{K}\mathbf{R}^{-1}\mathbf{R}^{-1}\mathbf{K}^{-1} \end{aligned}$$

The above two equations imply $\mathbf{H}^2 = \mathbf{H}_\theta^2 = \mathbf{H}_{2\theta}$ because

$$\mathbf{H}_\theta^2 = (\mathbf{K}\mathbf{R}^{-1}\mathbf{K}^{-1})^2 = \mathbf{K}\mathbf{R}^{-1}\mathbf{K}^{-1}\mathbf{K}\mathbf{R}^{-1}\mathbf{K}^{-1} = \mathbf{K}\mathbf{R}^{-1}\mathbf{R}^{-1}\mathbf{K}^{-1} = \mathbf{H}_{2\theta}$$

Q1.4.3 The planar homography is not sufficient to map any arbitrary image to other viewpoints because it requires the points in 3D environment all lie in the same plane. It would cause distortion if the points are not at the same plane.

Q1.4.4 Consider a line in 3D world with (X_0, Y_0, Z_0) as a starting point and a direction vector (d_X, d_Y, d_Z). The line can be defined as $(X_0 + td_X, Y_0 + td_Y, Z_0 + td_Z)$, where t is a parameter. Also consider a projection matrix $\mathbf{P}$. The projected line would be $\mathbf{x} = \mathbf{P}\mathbf{X}$, where $\mathbf{P}$ is

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

Note that we can view the points on the line as a function of t. Therefore, in homogeneous coordinates, the 2D projection (x, y) of the point $(X(t), Y(t), Z(t))$ can be expressed by

$$x(t) = \frac{p_{11}(X_0 + td_X) + p_{12}(Y_0 + td_Y) + p_{13}(Z_0 + td_Z) + p_{14}}{p_{31}(X_0 + td_X) + p_{32}(Y_0 + td_Y) + p_{33}(Z_0 + td_Z) + p_{34}}$$

$$y(t) = \frac{p_{21}(X_0 + td_X) + p_{22}(Y_0 + td_Y) + p_{23}(Z_0 + td_Z) + p_{24}}{p_{31}(X_0 + td_X) + p_{32}(Y_0 + td_Y) + p_{33}(Z_0 + td_Z) + p_{34}}$$

From the above equation, we can see that $x(t), y(t)$ are all linear function of the parameter t. Therefore we conclude that the mapped points also describe a line on the 2D image plane.

2 Computing Planar Homographies

2.1 Feature Detection and Matching

Q2.1.1 The computational cost of FAST detector is much less than the Harris corner detector because the former does not involve in calculating the gradient of each patches, which needs matrices calculation. The FAST detector only focuses on comparing pixel intensities in a small circle around the target pixel. Therefore the FAST detector is faster than the Harris corner detector.

Q2.1.2 The BRIEF descriptor is a type of binary descriptor that compares pixel intensity pairs to generate a descriptor. It offers better efficiency but lacking robustness to different scale and rotation. Filter banks captures image features such as edges, corners or textures through different types of filters, providing more robustness but higher computation cost.
Filter banks can be used as a type of descriptor in scenarios not requiring real-time computation.

Q2.1.3 Since the descriptor of BRIEF is a binary bit-string, we can use Hamming distance to measure the distance of two keypoints. Hamming distance calculates the number of different bits between two binary strings, and can be easily calculated by simple bit-wise XOR operation. Nearest Neighbor matching is done by finding the point with the smallest Hamming distance between descriptors. The benefit of using Hamming distance over conventional Euclidean distance is that it has better efficiency.

Q2.1.4 Below Listing 1 is my implementation and Figure 1 result of matching `cv_cover.jpg` and `cv_desk.png`.

```
def matchPics(I1, I2, opts):
    ratio = opts.ratio  # ratio for BRIEF feature descriptor
    sigma = opts.sigma  # threshold for corner detection using FAST
       feature detector

    # Convert images to grayscale
    I1_gray = skimage.color.rgb2gray(I1)
    I2_gray = skimage.color.rgb2gray(I2)

    # Detect features in both images using the FAST detector
    locs1 = corner_detection(I1_gray, sigma)
    locs2 = corner_detection(I2_gray, sigma)

    # Obtain BRIEF descriptors for the computed feature locations
    desc1, locs1 = computeBrief(I1_gray, locs1)
    desc2, locs2 = computeBrief(I2_gray, locs2)

    # Match features using the BRIEF descriptors
    matches = briefMatch(desc1, desc2, ratio)
    return matches, locs1, locs2
```

Listing 1: matchPics

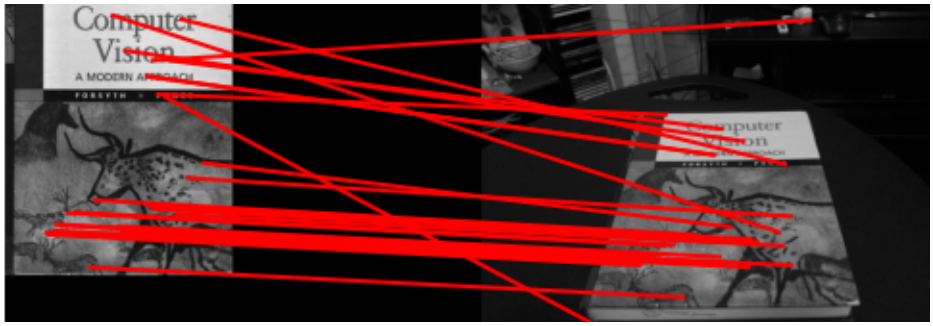
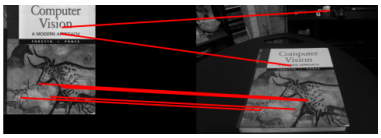
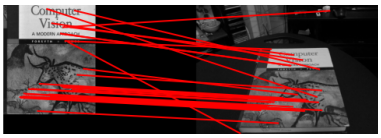


Figure 1: Result of `plotMatches` with $\sigma = 0.15, r = 0.7$

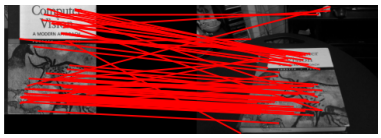
Q2.1.5 Figure 2 shows that the matched points increase as r increases. Figure 3 shows that the matched points decreases as σ increases.



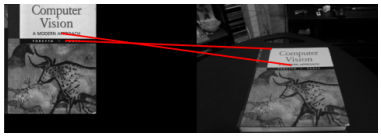
(a) $\sigma = 0.15, r = 0.6$



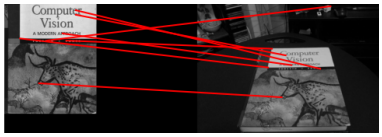
(b) $\sigma = 0.15, r = 0.7$



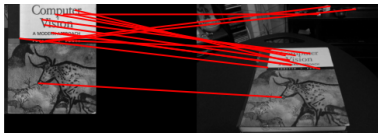
(c) $\sigma = 0.15, r = 0.8$



(d) $\sigma = 0.3, r = 0.6$

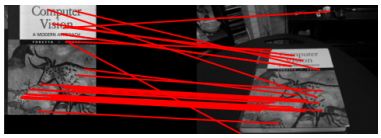


(e) $\sigma = 0.3, r = 0.7$

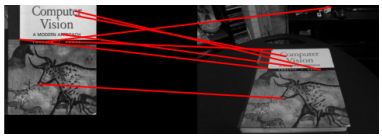


(f) $\sigma = 0.3, r = 0.8$

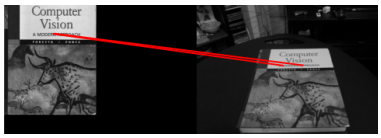
Figure 2: Comparison of fixed σ and different r



(a) $\sigma = 0.15, r = 0.7$



(b) $\sigma = 0.3, r = 0.7$



(c) $\sigma = 0.5, r = 0.7$

Figure 3: Comparison of different σ values with $r = 0.7$

The reason of the result is because as the ratio r increases, the Lowe's ratio test becomes less strict, allowing more matches where the best and second-best matches are closer in distance. σ controls the threshold for FAST feature detector. Larger σ implies less points would be considered as a keypoint.

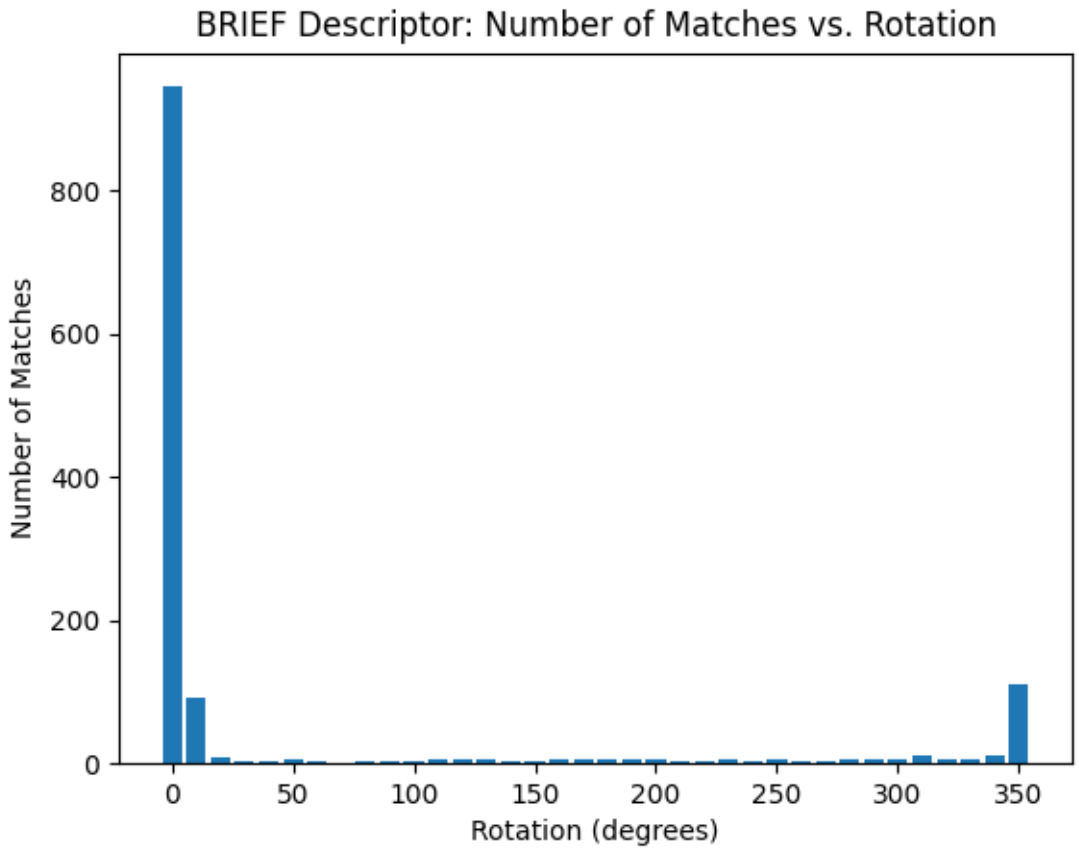
Q2.1.6 Below is my implementation and the output histogram. BRIEF behave this way because it is not rotation-invariant, meaning rotation would cause the matching to fail.

```python
def rotTest(opts):
    image = cv2.imread('../data/cv_cover.jpg')
    # Initialize histogram
    rotations = np.arange(0, 360, 10)
    match_counts = np.zeros(36)

    for angle in tqdm(rotations):
        rotated_image = rotate(image, angle=angle, reshape=False)
        # Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(image, rotated_image, opts)
        # Update histogram
        match_counts[angle//10] = len(matches)

    # Display histogram
    plt.figure()
    plt.bar(rotations, match_counts, width=8)
    plt.xlabel('Rotation (degrees)')
    plt.ylabel('Number of Matches')
    plt.title('BRIEF Descriptor: Number of Matches vs. Rotation')
    plt.show()
```

Listing 2: rotTest



2.2 Homography Computation

Q2.2.1 See the code below.

```python
def computeH(points1: np.array, points2: np.array) -> np.array:
    assert points1.shape == points2.shape, "The number of points should be the same"
    n_points = points1.shape[0]
    A = np.zeros((2*n_points, 9), dtype=np.float64)

    for i in range(n_points):
        x1, y1 = points1[i]
        x2, y2 = points2[i]
        A[2*i] = [-x2, -y2, -1, 0, 0, 0, x1*x2, x1*y2, x1]
        A[2*i+1] = [0, 0, 0, -x2, -y2, -1, y1*x2, y1*y2, y1]

    U, S, V = np.linalg.svd(A)
    H2to1 = V[-1].reshape(3, 3)

    H2to1 /= H2to1[2, 2]
    return H2to1
```

Q2.2.2 See the code below. I defined another helper function for normalization.

```python
def normalize(points: np.array) -> Tuple[np.array, np.array]:
    m = np.mean(points, 0)
    s = np.sqrt(2) / np.linalg.norm(points - m, axis=1).max()
    T = np.array([[s, 0, m[0]],
                  [0, s, m[1]],
                  [0, 0,    1]])
    transformation_mat = np.linalg.inv(T)
    normalized_points = (transformation_mat @ np.concatenate((points, np.ones((points.shape[0], 1))), axis=1).T)

    return transformation_mat, normalized_points[:2].T


def computeH_norm(points1: np.array, points2: np.array) -> np.array:
    T1, points1_norm = normalize(points1)
    T2, points2_norm = normalize(points2)

    H2to1_norm = computeH(points1_norm, points2_norm)
    H2to1 = np.linalg.inv(T1) @ H2to1_norm @ T2

    H2to1 /= H2to1[2, 2]
    return H2to1
```

Q2.2.3 See the code below.

```python
def computeH_ransac(locs1: np.array, locs2: np.array, opts: argparse.Namespace) -> Tuple[np.array, np.array]:
    max_iters = opts.max_iters
    inlier_tol = opts.inlier_tol

    N = locs1.shape[0]
    bestH2to1 = None
    best_inliers = np.zeros(N)
    max_inliers = 0

    for _ in range(max_iters):
        idx = np.random.choice(N, 4, replace=False)
        H2to1 = computeH_norm(locs1[idx], locs2[idx])

        # Apply homography to all points, calculate the inliners
        locs2_homo = np.hstack((locs2, np.ones((N, 1))))
        mapped_points = (H2to1 @ locs2_homo.T).T
        mapped_points /= mapped_points[:, 2].reshape(-1, 1) # shape: (N, 3)

        # Computer errors (L2 norm)
        errors = np.linalg.norm(mapped_points[:, :2] - locs1, axis=1)

        # Count inliers (where error is less than inlier tolerance)
        inliers = (errors < inlier_tol).astype(np.int)
        num_inliers = np.sum(inliers)

        # Update best H
        if num_inliers > max_inliers:
            max_inliers = num_inliers
            bestH2to1 = H2to1
            best_inliers = inliers

    return bestH2to1, best_inliers
```

Q2.2.4 See the code and result below. For the function `warpImage`, I externally defined some other opts for different choices of match method, file path, etc.

```python
def warpImage(opts, template_img, from_img, to_img):
    '''
    Function to warp from_img(hp_cover.jpg) onto to_img(cv_desk.png)
       using homography
    computed between from template image (cv_cover.jpg) and to_img(
       cv_desk.png).
    '''
    # Resize from_img such that the warpped image fit the correct
       size as cv_cover
    from_img = cv2.resize(from_img, (template_img.shape[1],
       template_img.shape[0]))

    if opts.match_method == 'BRIEF_FAST':
        matches, locs1, locs2 = matchPics(to_img, template_img, opts)
        # Important! Swap x, y axis because cv2.warpPerspective
           treats x, y differently
        locs1 = locs1[:, ::-1]
        locs2 = locs2[:, ::-1]
    elif opts.match_method == 'ORB':
        matches, locs1, locs2 = matchPicsORB(to_img, template_img,
           opts)
    elif opts.match_method == 'SIFT':
        matches, locs1, locs2 = matchPicsSIFT(to_img, template_img,
           opts)

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

    # Homography maps locs2 (template_img) to locs1 (to_img)
    bestH2to1, _ = computeH_ransac(locs1, locs2, opts)
    composite_img = compositeH(bestH2to1, from_img, to_img)

    if opts.show_warpped_img:
        cv2.imshow("Composite Image", composite_img)
        cv2.waitKey()

    if opts.store_Q225_img:
        cv2.imwrite(f"../Figures/composite_{opts.max_iters}_{opts.
           inlier_tol}.png", composite_img)

    return composite_img


if __name__ == "__main__":
    opts = get_opts()
    cv_cover = cv2.imread('../data/cv_cover.jpg')
    cv_desk = cv2.imread('../data/cv_desk.png')
    hp_cover = cv2.imread('../data/hp_cover.jpg')
    warpImage(opts, cv_cover, hp_cover, cv_desk)
```

Listing 3: `HarryPotterize.py`

```python
def compositeH(H2to1, template, img):
    '''
    Create a composite image after warping the template image on top
    of the image using the homography
    '''

    # Create mask of same size as template
    mask = np.ones_like(template, dtype=np.uint8)

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

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

    # Use mask to combine the warped template and the image
    composite_img = np.where(warped_mask > 0, warped_template, img)

    return composite_img
```

Listing 4: `plnarH.py`

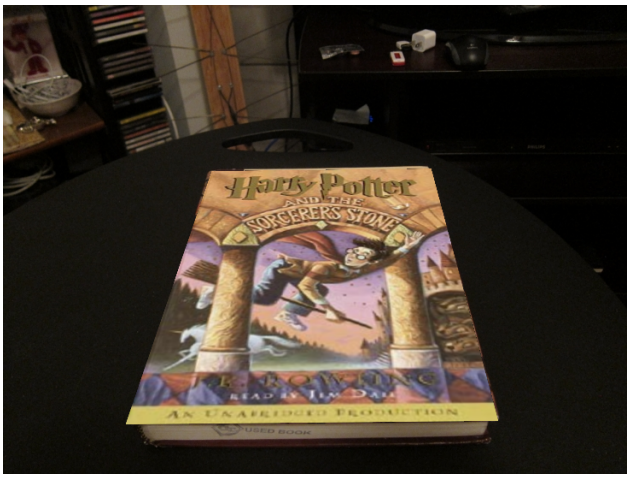
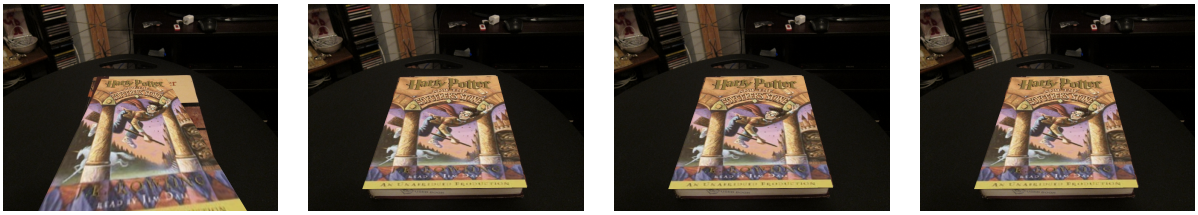


Figure 4: Composite image with default parameters

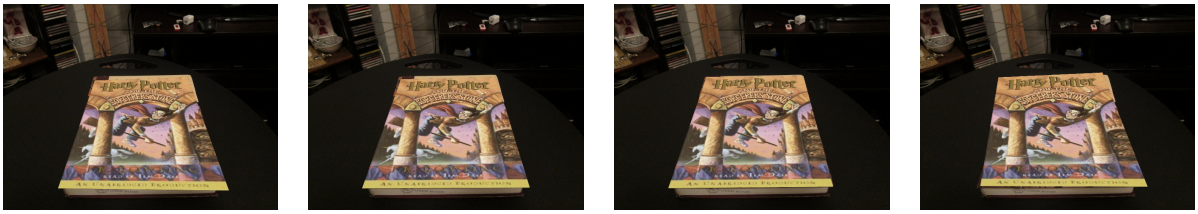
Q2.2.5 Figure 5 shows the results of varying the `max_iters` (denote as `MI`) parameter while keeping `inlier_tol` (denote as `IT`) fixed at 2.0. As `max_iters` increases, the number of iterations during the RANSAC process grows, potentially improving the robustness of the homography estimation.



(a) $\texttt{MI} = 10, \texttt{IT} = 2.0$ (b) $\texttt{MI} = 50, \texttt{IT} = 2.0$ (c) $\texttt{MI} = 100, \texttt{IT} = 2.0$ (d) $\texttt{MI} = 500, \texttt{IT} = 2.0$

Figure 5: Comparison of fixed `inlier_tol` = 2.0 and varying `max_iters`

Figure 6 shows the results of varying the `inlier_tol` parameter while keeping `max_iters` fixed at 500. The `inlier_tol` parameter controls the tolerance for considering points as inliers during the homography estimation. If `inlier_tol` we choose is too small, very few points would be selected to determine a homography, causing too many false negative pairs. On the other hand, if `inlier_tol` is too large, meaning we allow too many pairs of points, causing too many noisy points (false positive) to calculate the homography.



(a) $\texttt{M} = 500, \texttt{IT} = 0.5$ (b) $\texttt{M} = 500, \texttt{IT} = 1.0$ (c) $\texttt{M} = 500, \texttt{IT} = 2.0$ (d) $\texttt{M} = 500, \texttt{IT} = 10.0$

Figure 6: Comparison of fixed `max_iters` = 500 and varying `inlier_tol`

3 Creating your Augmented Reality application

3.1 Incorporating video

Below are the screenshots of my AR video. Video Link.
(https://drive.google.com/file/d/1y89MDxjumkipy3hqY3X-2iO_e8bDUmcz/view?usp=sharing)

(a) Left (b) Center (c) Right

Figure 7: Screenshots of generated AR video

For the code, I defined a helper function called `crop_save_video`, which would first crop the source video into the same ratio of the image (`cv_cover.jpg`) used to calculate homography. For the logic of the main function, first read the source video, target video, and the book cover image. For each frame, I calculate the warped image by calling `warpImage()`, and then store each frame to preserve the progress. Finally, reconstruct and save the whole video from the stored frames.

```
def crop_save_video(ar_src_vid_path: str, ar_img_path: str,
    ar_crop_vid_path: str) -> None:
    '''
    Function to crop the video to the center of each frame.add()
    The cropped video's ratio will be the same as the reference image. (
        cv_cover.jpg)
    The cropped video will be saved as '../data/cropped_ar_source.mov'.
    '''
    input_video = loadVid(ar_src_vid_path) # shape: (num_frames, height,
        width, 3)
    vid_height, vid_width = input_video[0].shape[:2] # (360, 640)
    vid_ratio = vid_width / vid_height

    ref_image = cv2.imread(ar_img_path)
    ref_height, ref_width = ref_image.shape[:2] # (440, 350)
    ref_ratio = ref_width / ref_height

    if vid_ratio > ref_ratio: # This is our case
        # Crop horizontally
        new_width = int(vid_height * ref_ratio) # 286
        new_height = vid_height # 360
    else:
        # Crop vertically
        new_width = vid_width
```

```
        new_height = int(vid_width / ref_ratio)

    start_x = (vid_width - new_width) // 2
    start_y = (vid_height - new_height) // 2

    cropped_video = input_video[:, start_y:start_y+new_height, start_x:
        start_x+new_width, :]

    # Save the cropped video
    fourcc = cv2.VideoWriter_fourcc(*'MJPG')
    fps = 30 # ar_source.mov panda video has 30 fps
    out = cv2.VideoWriter(ar_crop_vid_path, fourcc, fps, (new_width,
        new_height))

    for frame in tqdm(cropped_video):
        out.write(frame)

    print(f"Video saved successfully at {ar_crop_vid_path}.")
    out.release()


if __name__ == "__main__":
    opts = get_opts()

    # Only first run should crop and save the video
    if opts.crop_video:
        crop_save_video(opts.ar_src_vid_path, opts.ar_img_path, opts.
            ar_crop_vid_path)

    # Create the output directory if it does not exist
    if not os.path.exists(opts.ar_out_dir):
        os.makedirs(opts.ar_out_dir)

    # Load the cropped video
    cropped_video = loadVid(opts.ar_crop_vid_path)
    print(f"Loaded cropped videos at {opts.ar_crop_vid_path}, shape: {
        cropped_video.shape}") # (511, 360, 286, 3)

    # Load the target video
    target_video = loadVid(opts.ar_tgt_vid_path)
    print(f"Loaded target videos at {opts.ar_tgt_vid_path}, shape: {
        target_video.shape}") # (641, 480, 640, 3)

    # Load the reference image
    ref_image = cv2.imread(opts.ar_img_path)
    print(f"Loaded reference image at {opts.ar_img_path}, shape: {
        ref_image.shape}") # (440, 350, 3)

    # Generate warpped images and save each frames
    result_frame = min(cropped_video.shape[0], target_video.shape[0])
    for i in tqdm(range(result_frame)): # 511 frames
        composite_img = warpImage(opts, ref_image, cropped_video[i],
            target_video[i])
        cv2.imwrite(f"{opts.ar_out_dir}frame_{i}.png", composite_img)
```

```
69
70      # Reconstruct videos from saved frames
71      out = cv2.VideoWriter(f"{opts.ar_out_dir}{opts.ar_out_vid}", cv2.
            VideoWriter_fourcc(*'MJPG'), 30, (640, 480))
72      for i in tqdm(range(result_frame)):
73          img = cv2.imread(f"{opts.ar_out_dir}frame_{i}.png")
74          out.write(img)
75
76      out.release()
77      print(f"Video saved successfully at {opts.ar_out_dir}{opts.ar_out_vid
            }.")
```

3.2 Make Your AR Real Time

For this problem, I re-write my `warpImage()` function by replacing the feature matching and homography calculation function to OpenCV's API. I also added some helper function like `cropFrame()` to help me crop each frame to the same ratio of the template image, and `calculate_fps()` to help me calculate FPS. I calculated and printed the FPS each second, and the final average is 34.92. Below is my code. Note that I defined the argument `match_method` in `opts.py`, and I used ORB as my match method to accelerate. Therefore the command is `python ar_ec.py --match_method ORB`.

```
1   def warpImage(opts, template_img, from_img, to_img):
2       '''
3       Function to warp from_img onto to_img using homography computed
            between template_img and to_img.
4       Return the composite image after warping the from_img on top of to_img
             using the homography.
5       '''
6       # Resize from_img such that the warpped image fit the correct size as
            template_img
7       from_img = cv2.resize(from_img, (template_img.shape[1], template_img.
            shape[0]))
8
9       # Use cv2 ORB to match features
10      orb = cv2.ORB_create(nfeatures=2000)
11      kp1, des1 = orb.detectAndCompute(template_img, None)
12      kp2, des2 = orb.detectAndCompute(to_img, None)
13
14      bf = cv2.BFMatcher(cv2.NORM_HAMMING)
15      matches = bf.knnMatch(des1, des2, k=2)
16
17      good_matches = []
18      for m, n in matches:
19          if m.distance < 0.7 * n.distance:
20              good_matches.append(m)
21
22      if len(good_matches) < 4:
23          print("Not enough matches found!", file=sys.stderr)
24          return None
25
26      locs1 = np.array([kp1[m.queryIdx].pt for m in good_matches])
```

```
    locs2 = np.array([kp2[m.trainIdx].pt for m in good_matches])

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

    composite_img = compositeH(H, from_img, to_img)

    return composite_img


def cropFrame(frame, ratio):
    '''
    Function to crop a frame to the center based on the ratio of the reference image.
    Return the cropped frame.
    '''
    vid_height, vid_width = frame.shape[:2] # (360, 640)
    vid_ratio = vid_width / vid_height

    if vid_ratio > ratio:
        # Crop horizontally
        new_width = int(vid_height * ratio) # 286
        new_height = vid_height # 360
    else:
        # Crop vertically
        new_width = vid_width
        new_height = int(vid_width / ratio)

    start_x = (vid_width - new_width) // 2
    start_y = (vid_height - new_height) // 2

    cropped_frame = frame[start_y:start_y+new_height, start_x:start_x+new_width, :]
    return cropped_frame


def calculate_fps(start_time, frame_count):
    elapsed_time = time.time() - start_time
    fps = frame_count / elapsed_time if elapsed_time > 0 else 0
    return fps


if __name__ == "__main__":
    opts = get_opts()

    # Load the source video
    source_video = loadVid(opts.ar_src_vid_path)
    print(f"Loaded source videos at {opts.ar_src_vid_path}, shape: {source_video.shape}") # (511, 360, 640, 3)

    # Load the target video
    target_video = loadVid(opts.ar_tgt_vid_path)
    print(f"Loaded target videos at {opts.ar_tgt_vid_path}, shape: {target_video.shape}") # (641, 480, 640, 3)

```

```
    # Load the reference image
    ref_image = cv2.imread(opts.ar_img_path)
    print(f"Loaded reference image at {opts.ar_img_path}, shape: {
        ref_image.shape}") # (440, 350, 3)

    ref_height, ref_width = ref_image.shape[:2] # (440, 350)
    ref_ratio = ref_width / ref_height

    # Initialize variables for FPS calculation
    n_frames_sec = 0
    start_time = time.time()
    FPS = []

    # Generate warpped images and use cv2.imshow to show each frames
    result_frame = min(source_video.shape[0], target_video.shape[0])
    for i in range(result_frame):
        cropped_frame = cropFrame(source_video[i], ref_ratio)
        composite_img = warpImage(opts, ref_image, cropped_frame,
            target_video[i])

        n_frames_sec += 1
        fps = calculate_fps(start_time, n_frames_sec)
        # Reset FPS calculation every 1 second
        if time.time() - start_time > 1:
            print(f"FPS: {fps:.2f}")
            FPS.append(fps)
            n_frames_sec = 0
            start_time = time.time()

        cv2.imshow("AR Video", composite_img)
        cv2.waitKey(1)

    print(f"Average FPS: {np.mean(FPS):.2f}")
```

4 Create a Simple Panorama

Below are my code, original images, and the panorama result image.

Figure 8: Original Image 1 (top left). Original Image 2 (top right). Panorama (bottom).

For the coding part, I reused my implementation of `warpImage()` from `HarryPotterize.py`, and implemented some helper functions. I also added some arguments for specifying the input and output file paths.

```python
def crop_black_borders(img):
    # Crop the right black area
    for i in range(img.shape[1] - 1, 0, -1):
        if np.sum(img[:, i]) > 0:
            img = img[:, :i]
            break

    return img

def create_panorama(left_img, right_img, opts):
    width = left_img.shape[1] + right_img.shape[1]
    height = left_img.shape[0]

    panorama = np.zeros((height, width, 3), dtype=np.uint8)
    panorama[0:left_img.shape[0], 0:left_img.shape[1]] = left_img
    panorama = warpImage(opts, right_img, right_img, panorama)
    panorama = crop_black_borders(panorama)

    return panorama

if __name__ == "__main__":
    opts = get_opts()

    img1 = cv2.imread(opts.pano_left_img)
    img2 = cv2.imread(opts.pano_right_img)

    panorama = create_panorama(img1, img2, opts)

    # Save and display the result
    cv2.imwrite(opts.pano_out_img, panorama)
    cv2.imshow('Panorama', panorama)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

5 Collaboration

For this homework, I discussed with Ethan Lai, Daniel Yang, Willy Huang, Yu-Hsuan Li, and Tian-Zhi Li.