# 16-820 Advanced Computer Vision
# Homework 1: Augmented Reality With Planar Homographies

Abhinandan Vellanki

## 1 Planar Homographies as a Warp

### 1.1 Proof of homography

Assume a point $x_\pi$ on the plane $\pi$
$x_1$ is the image of this point in camera C
$x_2$ is the image of this point in camera C'

$$x_\pi = [X, Y, Z]$$
$$x_1 = [x_1, y_1, z_1] \equiv \left[\frac{x_1}{z_1}, \frac{y_1}{z_1}\right]$$
$$x_2 = [x_2, y_2, z_2] \equiv \left[\frac{x_2}{z_2}, \frac{y_2}{z_2}\right]$$

For camera C, the projection equation is

$$\lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = P_1 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \implies \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = P_1^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

Similarly, For camera C', the projection equation is

$$\lambda_2 \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = P_2 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \implies \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = P_2^{-1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

Since the objects points are all on the same plane $\pi$ their Z (depth) coordinate becomes 0, and their projection matrices' third column can be ignored because it is multiplied with the Z coordinate. This makes the P matrices 3x3 and hence, invertible.

From the above equations,

$$\lambda_1 \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} P_1^{-1} = \lambda_2 \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} P_2^{-1} \implies \frac{\lambda_1}{\lambda_2} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = P_1 P_2^{-1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

Removing the relative scaling between the images, denoted by $\frac{\lambda_1}{\lambda_2}$, and relating the terms with '$\equiv$' :

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \equiv P_1 P_2^{-1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

Rewriting equation (3)

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \equiv H \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

Where H is a 3x3 matrix known as the Homography matrix.

## 1.2 The Direct Linear Transform

1. h has 8 degrees of freedom.
2. A minimum of 4 point pairs are required to solve h.
3. From N matching point pairs related by Homography H, for any pair of points $(x_1^i, x_2^i)$:

$$x_1 \equiv Hx_2 \quad (i \in 1...N)$$

$$\implies \begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{01} \\ h_{10} & h_{11} & h_{12} \\ h_{10} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix}$$

Multiplying the matrices, we get:

$$x_1^i = \frac{h_{00}x_2^i + h_{01}y_2^i + h_{02}}{h_{20}x_2^i + h_{21}y_2^i + h_{22}}$$
$$y_1 = \frac{h_{10}x_2^i + h_{11}y_2^i + h_{12}}{h_{20}x_2^i + h_{21}y_2^i + h_{22}}$$

$$\implies x_1^i(h_{20}x_2^i + h_{21}y_2^i + h_{22}) = h_{00}x_2^i + h_{01}y_2^i + h_{02}$$
$$y_1(h_{20}x_2^i + h_{21}y_2^i + h_{22}) = h_{10}x_2^i + h_{11}y_2^i + h_{12}$$

Taking the terms to the left hand side and writing the equations as matrix multiplication, we get:

$$\begin{bmatrix} x_2^i & y_2^i & 1 & 0 & 0 & 0 & -x_1^i x_2^i & -x_1^i y_2^i & -x_1^i \\ 0 & 0 & 0 & x_2^i & y2^i & 1 & -y_1^i x_2^i & -y_1^i y_2^i & -y_2^i \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Relating this equation with $A_i h = 0$, we get

$$A_i = \begin{bmatrix} x_2^i & y_2^i & 1 & 0 & 0 & 0 & -x_1^i x_2^i & -x_1^i y_2^i & -x_1^i \\ 0 & 0 & 0 & x_2^i & y2^i & 1 & -y_1^i x_2^i & -y_1^i y_2^i & -y_2^i \end{bmatrix}$$

This matrix can be extended for N total pairs, resulting in a shape of (2N x 9).

The matrix $A_i$ is not full rank because for a valid homography solution to exist, it must have a non-trivial null space. Specifically, it should have a null space of dimension 1, corresponding to the homography h. For a well-posed problem, $A_i$ should have rank 8, leaving one dimension in the null space. A full rank would mean no valid homography exists.

Due to the null space dimension of 1, we can expect atleast one zero (or very small) singular value.

## 1.3 Using Matrix Decompositions to calculate the homography

No questions to answer.

## 1.4 Theory Questions

### 1.4.1 Homography under rotation

For two given cameras 1 and 2, their projections are given by

$$x_1 = K_1 \begin{bmatrix} I & 0 \end{bmatrix} X \quad and \quad x_2 = K_2 \begin{bmatrix} R & 0 \end{bmatrix} X$$

These equations can be written as

$$K_1^{-1} x_1 = X R^{-1} K_2^{-1} x_2 = X$$

Combining the two equations

$$\begin{aligned} K_1^{-1} x_1 &= R^{-1} K_2^{-1} x_2 \\ \implies x_1 &= K_1 R^{-1} K_2^{-1} x_2 \end{aligned}$$

Rewriting the equation

$$x_1 = H x_2$$

Where H is the homography matrix for the two cameras.

### 1.4.2 Understanding Homographies under rotation

Taking $x_1$ as the original view, $x_2$ as the view after rotating the camera by $\theta$, $x_3$ as the view after another rotation by $\theta$, We get these equations

$$x_2 = H x_1 \quad and \quad x_3 = H x_2$$

From these equations,

$$x_3 = HH x_1 \implies x_3 = H^2 x_1$$

### 1.4.3 Limitations of Planar Homography

In arbitrary scenes, there might be cases where all of the object points do not lie on the same plane, hence the projection matrix would not be invertible, thereby making planar homography insufficient to map this object's images from one viewpoint to another.

### 1.4.4 Behavior of lines under perspective projections

A line in 3D space is represented by $X(t) = X_o + dt$, where $X_o$ is a point on the line and d is the direction. Applying perspective projection ($x = PX$), to the line equation, we get $x(t) = P(X_o + dt)$, which can be rewritten as $x(t) = PX_o + Pdt$, which is also of the line equation form, where $PX_o$ is a point on the line and $Pd$ is the direction. Hence, perspective projection projects a line in 3D to a line in 2D.

# 2 Computing Planar Homographies

## 2.1 Feature Detection and Matching

### 2.1.1 Fast Detector vs HARRIS Detector

The FAST (Features from Accelerated Segment Test) detector does an initial screening by computing the difference in intensities between only 3 other pixels and the chosen pixel, eliminating all cases that fail, thereby avoiding more computation for failed cases. For the cases that pass this screening, the difference against the other 13 pixels is computed. However, this type of detection might miss some corners where there is no sharp right angle, making it less accurate than the Harris detector.

FAST detector is much faster than the Harris detector which computes the gradient between the chosen pixel and all pixels around it within a certain range, which would perform more computations than the FAST detector.

### 2.1.2 BRIEF vs Filterbanks

Filterbanks are collections of filters that can typically be applied to images to extract interest points.

BRIEF performs binary comparison between randomly sampled pixels from a patch and then compares the difference (Hamming distance) with the result of binary comparison of randomly sampled pixels from another patch, to determine if the two patches are a match.

The filterbanks can be used as descriptors by applying many or all of the filters on a patch, so that for every pixel, we get a unique multidimensional 'filter response', hence creating a description for that patch which can be used for matching.

### 2.1.3 Hamming Distance vs Euclidean Distance

In BRIEF descriptors, the Hamming provides a measure of similarity. A lower Hamming distance value indicates more similar descriptors.

The Hamming distance and nearest neighbor can be used during the matching process of BRIEF descriptors. For each BRIEF descriptor in the query image, the Hamming distance to all descriptors in the reference image is computed. Then, the descriptor in the reference image with the smallest Hamming distance is the nearest neighbor, or in other words, the matching descriptor.

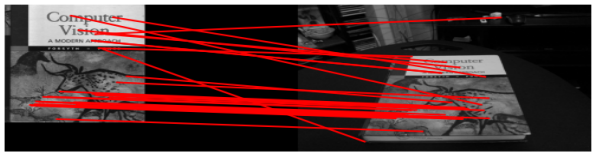
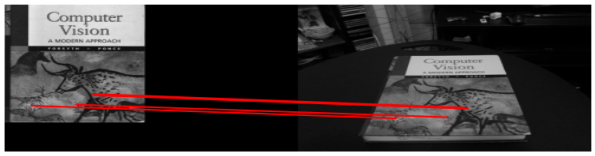
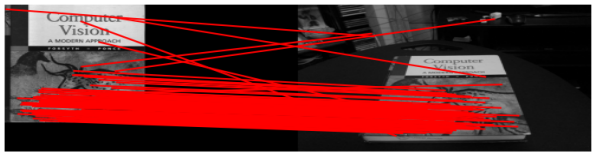
Compared to Euclidean distance, Hamming distance is faster to compute, making the matching process time efficient. Hamming distance, by nature, is more suited to comparing discrete data, making it a good fit for this application.

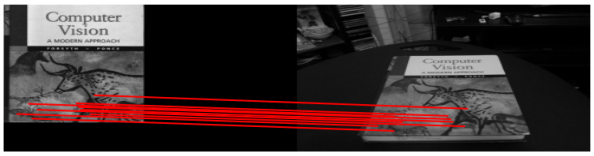
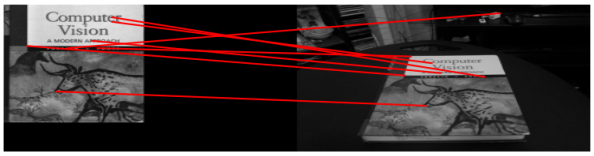
### 2.1.4 Feature Matching

**matchPics():**

```python
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'

    # Convert Images to GrayScale
    gI1 = skimage.color.rgb2gray(I1)
    gI2 = skimage.color.rgb2gray(I2)

    # Detect Features in Both Images
    corners_gI1 = corner_detection(img=gI1, sigma=sigma)
    corners_gI2 = corner_detection(img=gI2, sigma=sigma)
      You, 1 second ago • Uncommitted changes
    # Obtain descriptors for the computed feat
    descriptors_gI1, locs1 = computeBrief(img=gI1, locs=corners_gI1)
    descriptors_gI2, locs2 = computeBrief(img=gI2, locs=corners_gI2)

    # Match features using the descriptors
    matches = briefMatch(desc1=descriptors_gI1, desc2=descriptors_gI2, ratio=ratio)

    return matches, locs1, locs2
```

### 2.1.5 Feature Matching and Parameter Tuning

| Sigma | Ratio | Number of Matches | Visualization |
|---|---|---|---|
| 0.15 | 0.7 | 27 |  |
| 0.15 | 0.5 | 4 |  |
| 0.15 | 0.3 | 0 |  |
| 0.05 | 0.7 | 119 |  |

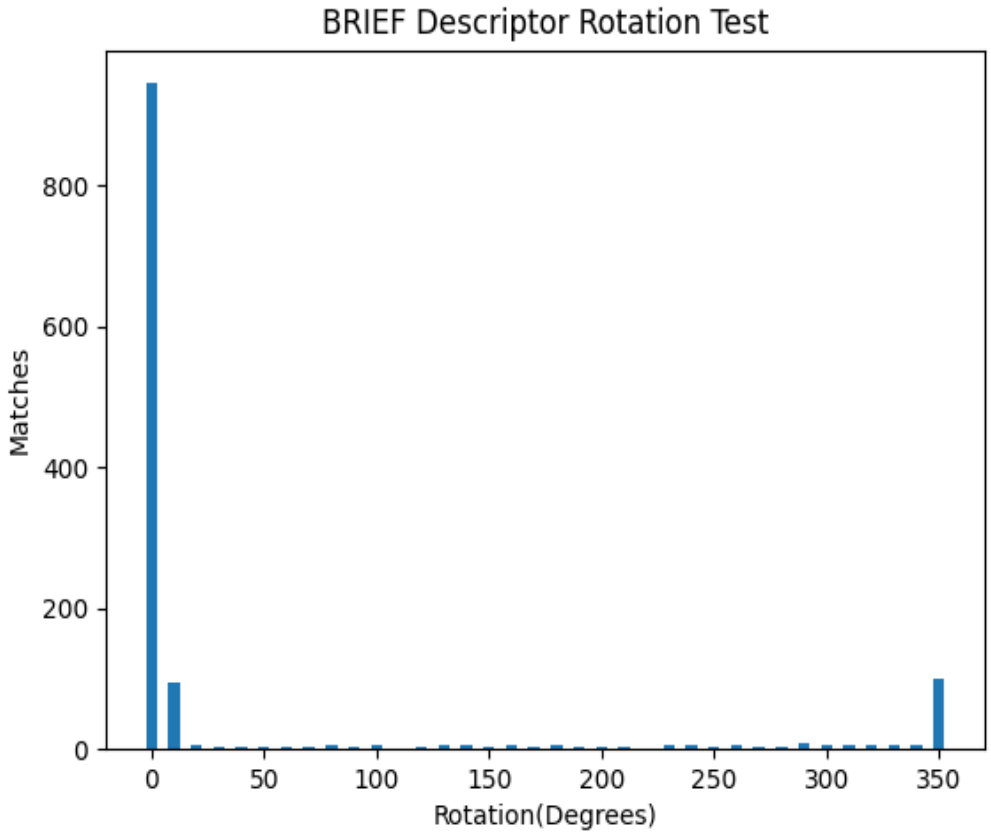
| Sigma | Ratio | Number of Matches | Visualization |
|---|---|---|---|
| 0.05 | 0.5 | 9 |  |
| 0.05 | 0.3 | 0 |  |
| 0.30 | 0.7 | 7 |  |
| 0.30 | 0.5 | 0 |  |

### 2.1.6 BRIEF and Rotations

**briefRotTest.py:**

```python
def rotTest(opts):

    # TODO: Read the image and convert to grayscale, if necessary
    img = cv2.imread("../data/cv_cover.jpg")

    matches_count = []
    rotations = []
    step = 0

    for i in range(36):
        step += 1

        # TODO: Rotate Image
        rot_img = scipy.ndimage.rotate(img, 10.0 * (i))

        # TODO: Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(I1=img, I2=rot_img, opts=opts)

        # TODO: Update histogram
        matches_count.append(len(matches))
        rotations.append(10.0 * i)

        # Display matches every 12 steps
        if step == 12:
            plotMatches(im1=img, im2=rot_img, matches=matches, locs1=locs1, locs2=locs2)
            step = 0

    # TODO: Display histogram
    plt.bar(x=rotations, height=matches_count, width=5.0)

    # add labels to axes and add title
    plt.xlabel("Rotation(Degrees)")
    plt.ylabel("Matches")
    plt.title("BRIEF Descriptor Rotation Test")

    # display plot
    plt.show()

if __name__ == "__main__":

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

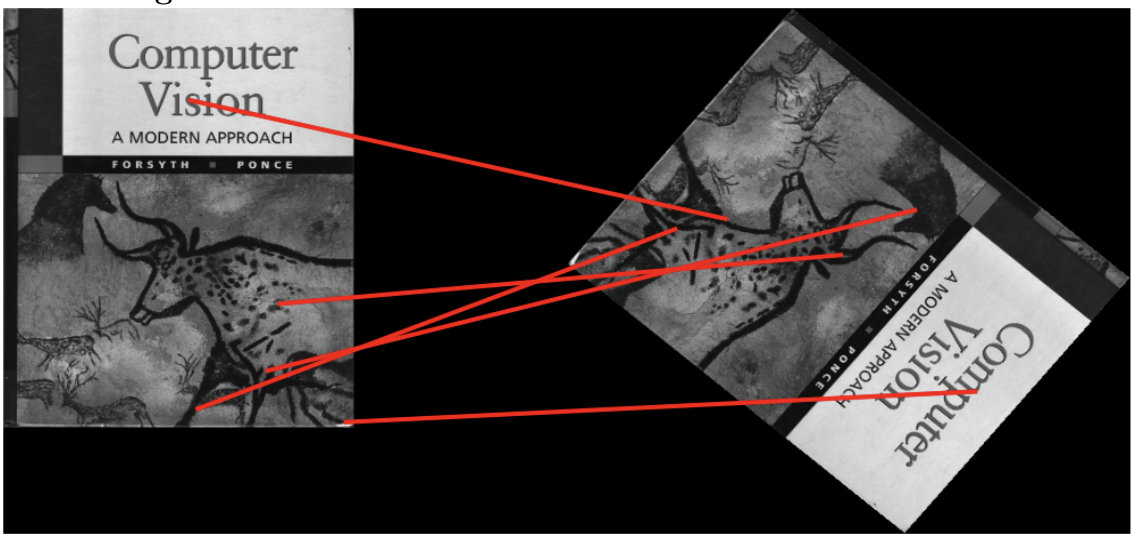
**Histogram of degree of rotation and number of matches:**




**Vizualization of BRIEF descriptor matching at different rotations:**

- **At 120 degree rotation:**

- **At 240 degree rotation:**




- **At 350 degree rotation:**


The BRIEF descriptor directly compares pixel intensities at fixed offsets from the keypoint. When the image is rotated, these relative pixel positions change. That means, when an image is rotated, pixel pairs being compared will correspond to different parts of the image content. Hence, the intensity relationships between sampled pixels will change and the resulting descriptor will be different, even for the same feature point. Hence, the BRIEF descriptor is not rotation invariant.

## 2.2 Homography Computation

### 2.2.1 Computing the Homography

computeH():

```python
def computeH(x1, x2):
    # Q2.2.1
    # TODO: Compute the homography between two sets of points

    if x1.shape[0] == x2.shape[0]:
        N = x1.shape[0]
        if N < 4:
            return None
    else:
        return None

    # declare A matrix
    A = []

    # populate A
    for i in range(N):
        row1 = [
            x2[i][0],
            x2[i][1],
            1,
            0,
            0,
            0,
            -x1[i][0] * x2[i][0],
            -x1[i][0] * x2[i][1],
            -x1[i][0],
        ]
        row2 = [
            0,
            0,
            0,
            x2[i][0],
            x2[i][1],
            1,
            -x1[i][1] * x2[i][0],
            -x1[i][1] * x2[i][1],
            -x1[i][1],
        ]
        A.append(row1)
        A.append(row2)      You, yesterday • Uncommitted changes

    A = np.array(A)

    # svd method:
    U, s, V = np.linalg.svd(A, full_matrices=True)
    H2to1 = V[-1].reshape((3, 3))
```

### 2.2.2 Homography Normalization

**computeH_norm():**

```python
def computeH_norm(x1, x2):
    # Q2.2.2
    # TODO: Compute the centroid of the points

    centroid_x1 = compute_centroid(x1)
    centroid_x2 = compute_centroid(x2)

    # TODO: Shift the origin of the points to the centroid
    x1_shifted = shift_points(x1, centroid_x1)
    x2_shifted = shift_points(x2, centroid_x2)

    # TODO: Normalize the points so that the largest distance from the origin is equal to sqrt(2)
    x1_normalized, x1_scale = normalize(x1_shifted)
    x2_normalized, x2_scale = normalize(x2_shifted)

    # TODO: Similarity transform 1
    T_1 = np.array(
        [
            [1, 0, -centroid_x1[0]],
            [0, 1, -centroid_x1[1]],
            [0, 0, 1 / x1_scale],
        ]
    )
    T_1 = x1_scale * T_1      You, 1 second ago • Uncommitted changes

    # TODO: Similarity transform 2
    T_2 = np.array(
        [
            [1, 0, -centroid_x2[0]],
            [0, 1, -centroid_x2[1]],
            [0, 0, 1 / x2_scale],
        ]
    )
    T_2 = x2_scale * T_2

    # TODO: Compute homography
    norm_H2to1 = computeH(x1=x1_normalized, x2=x2_normalized)

    # TODO: Denormalization - convert norm_H2to1 to H2to1
    T_1_inverse = np.linalg.inv(T_1)
    H2to1 = T_1_inverse @ norm_H2to1 @ T_2

    return H2to1
```

**Helper functions written for normalization:**

```python
def compute_centroid(x):
    # compute centroid by taking mean along both axes
    length = x.shape[0]
    sum_x = np.sum(x[:, 0])
    sum_y = np.sum(x[:, 1])
    return (sum_x // length, sum_y // length)


def shift_points(points, origin):
    # shift the origin of points to the new origin
    N = points.shape[0]
    for i in range(N):
        points[i][0] = points[i][0] - origin[0]
        points[i][1] = points[i][1] - origin[1]
    return points


def normalize(points):
    max_distance = -1
    for i in range(points.shape[0]):
        dist = np.linalg.norm(points[i])
        if dist > max_distance:
            max_distance = dist
    scale = (2**0.5) / max_distance
    points = scale * points
    return points, scale
```

### 2.2.3 Implement RANSAC

computeH_ransac():

```python
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

    if locs1.shape == locs2.shape:
        N = locs1.shape[0]
    else:
        return None

    iterations = 0
    inliers = []
    bestH2to1 = None

    while iterations < max_iters:
        # select four matching pairs randomly
        chosen_points_idx = np.random.choice(len(locs1), 4, replace=False)
        x1 = locs1[chosen_points_idx]
        x2 = locs2[chosen_points_idx]

        # compute homography H      You, 1 second ago • Uncommitted changes
        this_H2to1 = computeH_norm(x1=x1, x2=x2)

        # transform all source points using H, compare with actual points and count inliers
        this_inliers = []

        for i in range(N):
            # transform source points
            point = np.array([[locs2[i][0]], [locs2[i][1]], [1]])
            transformed_point = this_H2to1 @ point
            transformed_point = 1 / transformed_point[2][0] * transformed_point
            actual_point = np.array([[locs1[i][0]], [locs1[i][1]], [1]])

            # calculate error
            error = np.linalg.norm(transformed_point - actual_point)

            # update best inliers and best H
            if error < inlier_tol:
                this_inliers.append(1)
            else:
                this_inliers.append(0)

        if np.count_nonzero(this_inliers) > np.count_nonzero(inliers):
            bestH2to1 = this_H2to1
            inliers = this_inliers

        iterations += 1
```

**computeH_ransac() (contd.):**

```
    # Re compute H with all the inliers
    best_inliers_1 = []
    best_inliers_2 = []
    for i in range(N):
        if inliers[i] == 1:
            best_inliers_1.append(locs1[i])
            best_inliers_2.append(locs2[i])

    bestH2to1 = computeH_norm(x1=np.array(best_inliers_1), x2=np.array(best_inliers_2))

    return bestH2to1, inliers
```

### 2.2.4 Automated Homography Estimation and Warping

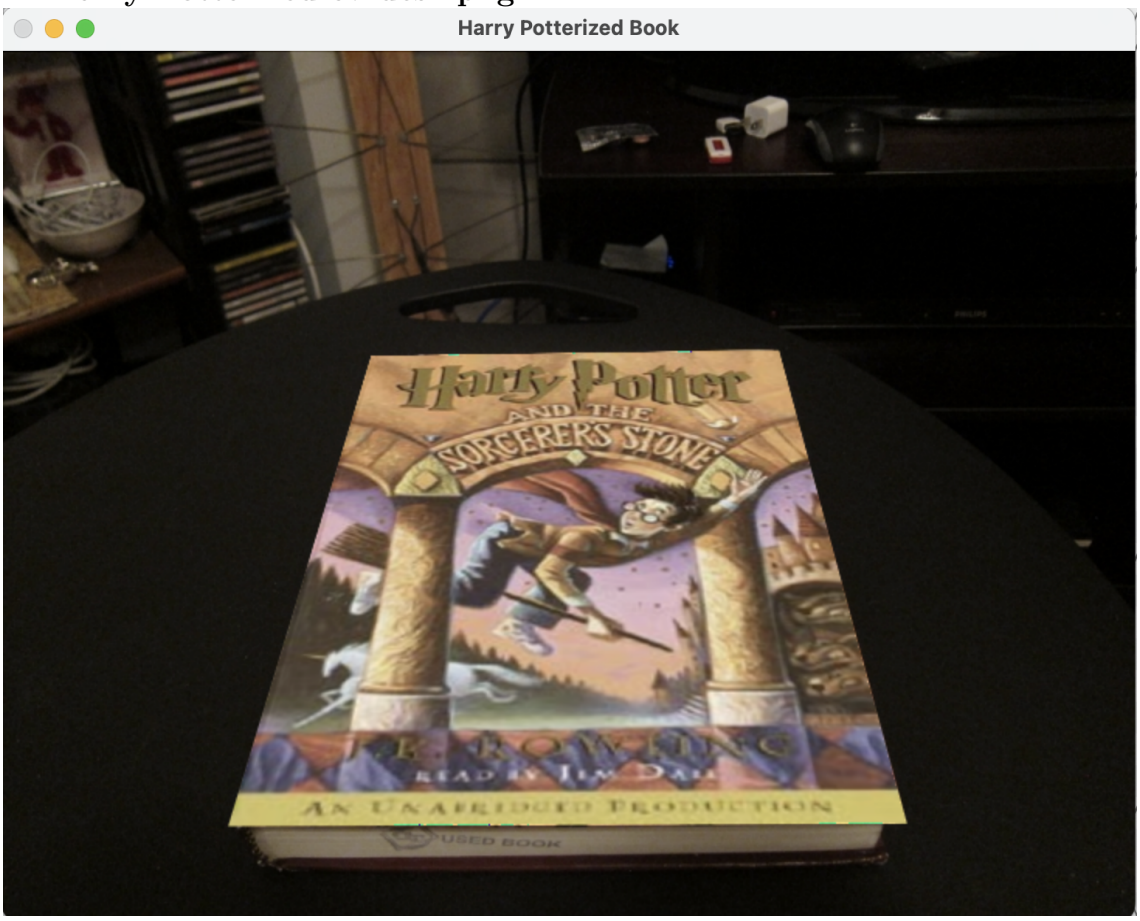
Initially, the warped *hp_cover.jpg* image does not fill up the same space as the book in the destination image because it is smaller in dimensions than the *cv_cover.jpg* image which is used to compute the homography. Hence, there are simply not enough pixels in the image to fill up the space.

This can be fixed by resizing the *hp_cover.jpg* image to the same dimensions as the *cv_cover.jpg* image before warping and pasting it onto the destination image.

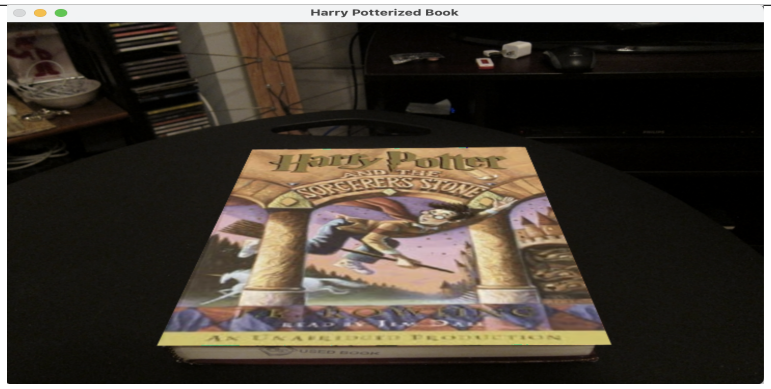
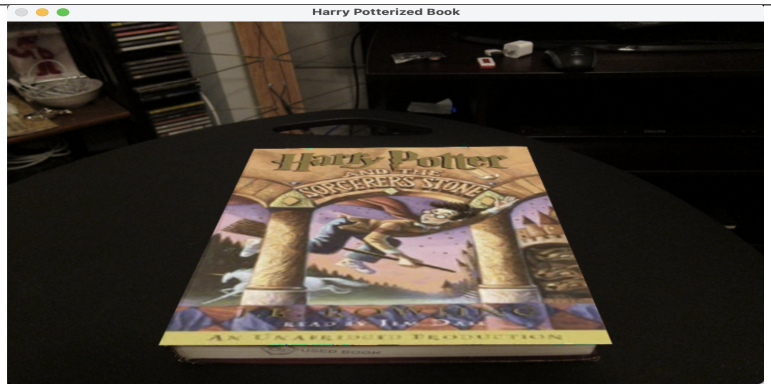
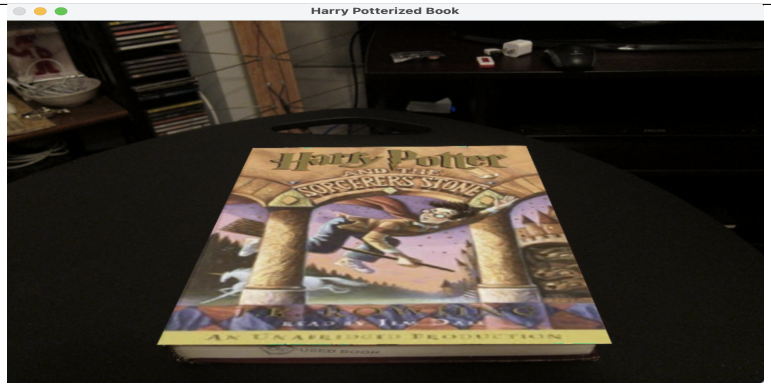
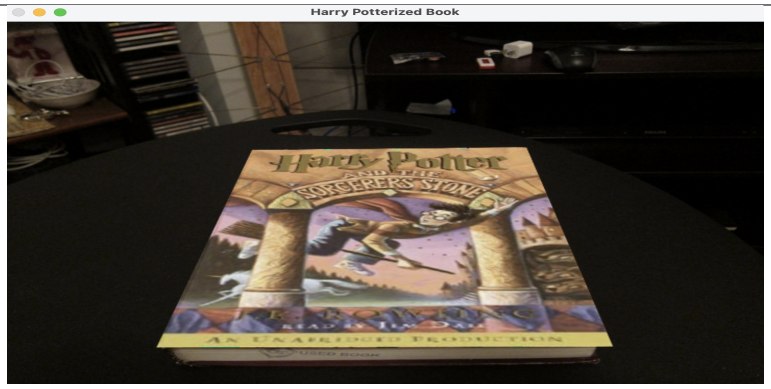
**warpImage():**

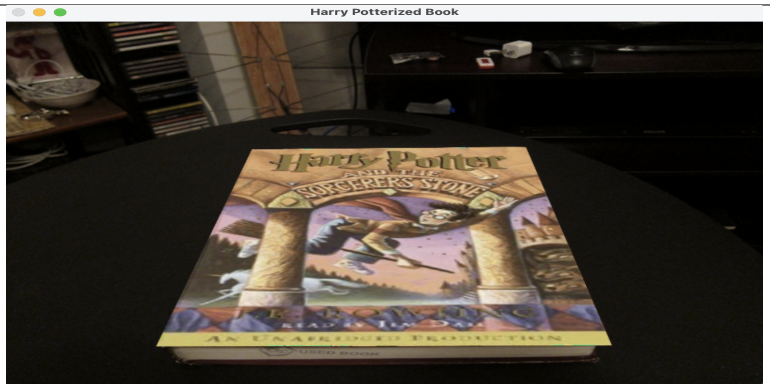
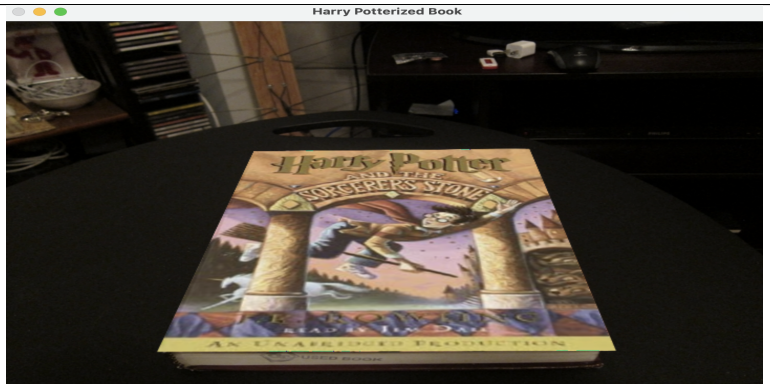
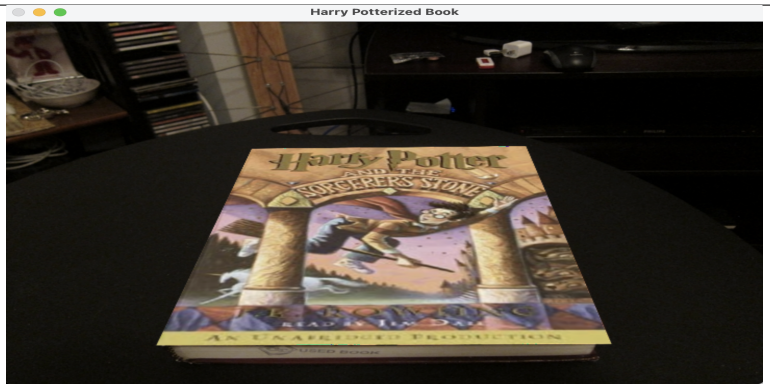
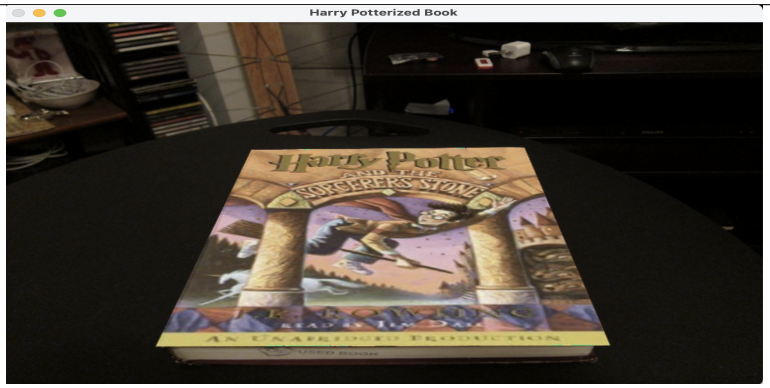
```
def warpImage(opts):
    img_1 = np.array(cv2.imread("../data/cv_cover.jpg"))
    img_2 = np.array(cv2.imread("../data/cv_desk.png"))
    hp_cover = np.array(cv2.imread("../data/hp_cover.jpg"))

    # Resizing the template to the same size as the source image
    hp_cover = cv2.resize(
        src=hp_cover,
        dsize=(img_1.shape[1], img_1.shape[0]),
    )

    # compute homography between img_1 and img_2
    matches, locs1, locs2 = matchPics(I1=img_1, I2=img_2, opts=opts)

    # display matches
    plotMatches(img_1, img_2, matches, locs1, locs2)

    # filter out unmatched interest points from locs1 and locs2
    locs1 = locs1[matches[:, 0]]
    locs2 = locs2[matches[:, 1]]

    # swap columns of locs outputs of matchPics
    locs1[:, [0, 1]] = locs1[:, [1, 0]]
    locs2[:, [0, 1]] = locs2[:, [1, 0]]

    H2to1, inliers = computeH_ransac(locs1=locs1, locs2=locs2, opts=opts)
    H1to2 = np.linalg.inv(H2to1)

    composite_img = compositeH(img=img_2, template=hp_cover, H2to1=H1to2)

    cv2.imshow("Harry Potterized Book", composite_img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

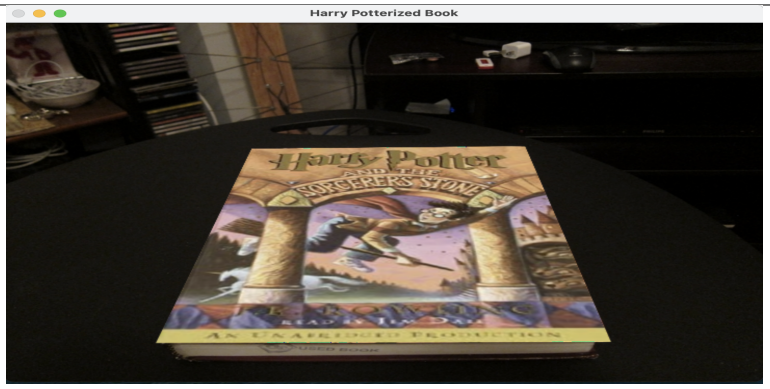
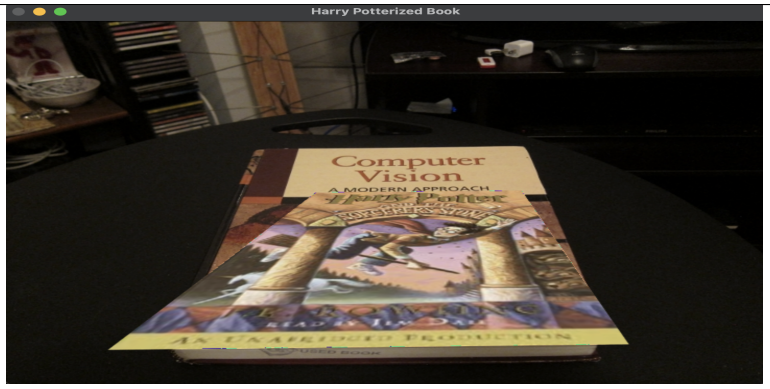
**Harry Potterized cv_desk.png:**



### 2.2.5 RANSAC Parameter Tuning

| Max Iterations | Inlier Tolerance | Visualization |
|---|---|---|
| 200 | 1.0 |   |
| 200 | 4.0 |   |
| 50 | 2.0 |   |
| 50 | 1.0 |   |

| Max Iterations | Inlier Tolerance | Visualization |
|---|---|---|
| 50 | 4.0 |   |
| 800 | 2.0 |   |
| 800 | 1.0 |   |
| 800 | 4.0 |   |

| Max Iterations | Inlier Tolerance | Visualization |
|---|---|---|
| 3200 | 2.0 |   |
| 10 | 100.0 |   |

From this study, it was observed that increasing the maximum iterations increased the number of trials conducted to estimate the Homography with most inliers, which would mean that the odds of finding a correct Homography increases with number of iterations. However, in this specific case, no significant difference was observed in the output image even when the maximum iterations was increased eightfold.

Increasing the inlier tolerance would loosen the strictness of finding inliers for each Homography estimation, hence resulting in a less accurate Homography, while a lower inlier tolerance would increase the strictness and make the Homography more accurate. However, in this specific case, no significant difference was observed in the output image when the inlier tolerance was halved or doubled.

An error case was found with a very high inlier tolerance and low number of maximum iterations, hence confirming the above understandings.

# 3 Creating Augmented Reality Application

## 3.1 Incorporating Video

**Video Screenshots:**


ar.py:

```python
def main(opts):

    # load video
    source_video_frames = loadVid(path="../data/ar_source.mov")
    destination_video_frames = loadVid(path="../data/book.mov")

    # load element for calculating homography
    element = cv2.imread("../data/cv_cover.jpg")

    # determine ratio
    element_ratio = element.shape[1] / element.shape[0]  # width / height

    # video writer
    output = cv2.VideoWriter(
        "../data/ar_video.avi",
        cv2.VideoWriter_fourcc("M", "J", "P", "G"),
        20,
        (destination_video_frames[0].shape[1], destination_video_frames[0].shape[0]),
    )

    for source_frame, destination_img in zip(
        source_video_frames, destination_video_frames
    ):

        # Get the template to be pasted onto the destination frame
        template = crop_middle_resize(
            element=element, frame=source_frame, ratio=element_ratio
        )

        # Get matching points between element and destination frame
        matches, locs1, locs2 = matchPics(I1=element, I2=destination_img, opts=opts)

        # filter out unmatched interest points from locs1 and locs2
        locs1 = locs1[matches[:, 0]]
        locs2 = locs2[matches[:, 1]]

        # swap columns of locs outputs of matchPics
        locs1[:, [0, 1]] = locs1[:, [1, 0]]
        locs2[:, [0, 1]] = locs2[:, [1, 0]]

        HEtoD, inliers = computeH_ransac(locs1=locs1, locs2=locs2, opts=opts)
        HEtoD = np.linalg.inv(HEtoD)

        new_frame = compositeH(img=destination_img, template=template, H2to1=HEtoD)

        output.write(new_frame)

    output.release()

if __name__ == "__main__":

    opts = get_opts()
    main(opts=opts)
```

**Helper function:**

```python
def crop_middle_resize(element, frame, ratio):
    # To extract the centre part of the frame by a ratio and resize it to the same size as the element

    # Get width of centre part from height and ratio
    crop_width = ratio * frame.shape[0]
    crop_height = frame.shape[0]

    # Crop out centre part
    mid_x, mid_y = int(frame.shape[1] / 2), int(frame.shape[0] / 2)
    cw2, ch2 = int(crop_width / 2), int(crop_height / 2)
    crop_img = frame[mid_y - ch2 : mid_y + ch2, mid_x - cw2 : mid_x + cw2]

    # Remove zeros from the top and bottom
    crop_img = crop_img[
        43:-43, :, :
    ]  # these dimensions were found for the specific video

    # Resize the cropped part to the element dimensions
    crop_img = cv2.resize(src=crop_img, dsize=(element.shape[1], element.shape[0]))

    return crop_img
```

AR Video Location: hw1/result/ar.avi

# 4 Creating Simple Panoramas

panorama.py:

```python
def main(opts):
    # Load images
    img1 = cv2.imread("../data/mypano_left.jpeg")
    img2 = cv2.imread("../data/mypano_right.jpeg")

    # Taking img2 as the destination image, pad zeros in all directions of width equal to img1

    img2 = cv2.copyMakeBorder(
        img2,
        img1.shape[0],
        img1.shape[0],
        img1.shape[1],
        img1.shape[1],
        cv2.BORDER_CONSTANT,
        None,
        value=0,
    )

    # Find matches
    matches, locs1, locs2 = matchPics(I1=img1, I2=img2, opts=opts)
    # plotMatches(im1=img1, im2=img2, matches=matches, locs1=locs1, locs2=locs2)

    # filter out unmatched interest points from locs1 and locs2
    locs1 = locs1[matches[:, 0]]
    locs2 = locs2[matches[:, 1]]

    # swap columns of locs outputs of matchPics
    locs1[:, [0, 1]] = locs1[:, [1, 0]]
    locs2[:, [0, 1]] = locs2[:, [1, 0]]

    # compute Homography
    HLtoR, inliers = computeH_ransac(locs1=locs2, locs2=locs1, opts=opts)

    # warp and paste left image into right image coordinates
    panorama = compositeH(template=img1, H2to1=HLtoR, img=img2)

    # Remove borders from panorama
    _, thresh = cv2.threshold(
        cv2.cvtColor(panorama, cv2.COLOR_BGR2GRAY), 1, 255, cv2.THRESH_BINARY
    )
    contours, h = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    x, y, w, h = cv2.boundingRect(contours[0])
    panorama = panorama[y : y + h, x : x + w]

    cv2.imwrite("../data/pano.jpeg", panorama)

    return


if __name__ == "__main__":

    opts = get_opts()
    main(opts)
```

**Testing:**

Original Images:

Panorama Image:

# 5 References

## 5.1 Collaborators:

- Ayush Fadia
- Parth Gupta
- Sreeharsha Paruchuri
- Yu Jin Goh

## 5.2 Resources:

- OpenCV Python Documentation
- NumPy Python Documentation
- Scikit Image Python Documentation