Q1.1

Establish Projection Matrix

$$(1a)\quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = M^a_{\text{in}} M^a_{\text{ext}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad (1b)\quad \begin{bmatrix} x'' \\ y'' \\ 1 \end{bmatrix} = M^b_{\text{in}} M^b_{\text{ext}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

To begin, establish the face that the same three dimenstional point can be mapped to two different cameras, via their intrisics and extrinsics.

Planar Homography Simplification

$$\text{with } Z = 0 \;\Rightarrow\quad (2a)\quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H_1 \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \qquad (2b)\quad \begin{bmatrix} x'' \\ y'' \\ 1 \end{bmatrix} = H_2 \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

By setting Z=0, we can see that the homography comes out of the projection matrix. Since we are doing a projection from a plane we can safely assume that at least one of the 3D points will always be zero.

Simplify

$$(3)\quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H_1 H_2^{-1} \begin{bmatrix} x'' \\ y'' \\ 1 \end{bmatrix} \quad \Rightarrow \quad x_1 \equiv H x_2$$

A few steps are passed over from step 2, but the homographies are inverted and 2a, 2b are equated. Simplifying yields equation 3.

Q1.2

1. How many degrees of freedom does h have?
   - 8
2. How many point pairs are required to solve h?
   - 4
3. Derive A:

$$(4)\quad \lambda\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

$$(5)\ x = \frac{h_{11}X + h_{12}Y + h_{13}}{h_{31}X + h_{32}Y + h_{33}}$$

$$(6)\ y = \frac{h_{21}X + h_{22}Y + h_{23}}{h_{31}X + h_{32}Y + h_{33}}$$

Using the equation of the homography (4), we can write equations out defining points x, y in the camera frame (5), (6).

$$(7)\quad 0 = h_{11}X + h_{12}Y + h_{13} - x(h_{31}X + h_{32}Y + h_{33})$$

$$(8)\quad 0 = h_{21}X + h_{22}Y + h_{23} - y(h_{31}X + h_{32}Y + h_{33})$$

Manipulating (5), (6) will yield (7), (8) which for will be decomposed for each match pair. The accumulation of of these repeated (7), (8) pairs will yield our A matrix.

$$(9)\quad \begin{bmatrix} X & Y & 1 & 0 & 0 & 0 & -\mathrm{xX} & -\mathrm{xY} & -x \\ 0 & 0 & 0 & X & Y & 1 & -\mathrm{yX} & -\mathrm{yY} & -y \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}\begin{bmatrix} h_{11} \\ \vdots \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

4. Is a full rank? What is the trivial solution? What impact will that have on the singular values

A is not a full row matrix, which we know because a full row matrix will only have the trivial solution in its nullspace {0}. However, if we want to generate singular values such that the equation Ah=0 is satisfied, we will need to have access to nontrivial occupants of the nullspace.

Q1.4.1

$$(10)\quad \lambda \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = k_1 \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix} \qquad (11)\quad \lambda \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = k_2 \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix}$$

Initially think of two separate points in the world that are just taken by a camera, which we define in (10), (11).

$$(12)\quad R \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix} = \begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \end{bmatrix} \qquad (13)\quad \lambda k_1^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix}$$

Given the facts in (12), (13) which are that the points in image two are simply rotated points of image one. Also by inverting the intrisitc matrix of (10) we can substitute (13) into (12) to obtain (14).

$$(14)\quad \lambda \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = k_2 R k_1^{-1} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

Finally, we see that we two points are seperated by a rotation are merely seperated by a pure rotation.

Q1.4.2

In order to demonstrate that appling a homography of H^2 is the same as a rotation of 2θ, it is helpful to look at a simple rotation matrix multiplied by itself. Doing so results in double angle theorems, and a simplification of a rotation of 2θ.

$$(15)\quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\theta & -s\theta \\ 0 & s\theta & c\theta \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & c\theta & -s\theta \\ 0 & s\theta & c\theta \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c^2\theta - s^2\theta & -2s\theta c\theta \\ 0 & 2s\theta c\theta & c^2\theta - s^2\theta \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c2\theta & -s2\theta \\ 0 & s2\theta & c2\theta \end{bmatrix}$$

An alternative to this would be using the following approach: x2 = H(x1); x3 = H(x2), x3 = H(H)(x1). This essentially constitutes computing two successive homographies between successive points, then via substituion, replace the intermediate point. Thus showing that you get to the final point by performing the same homography twice: H^2.

Q1.4.3

There are some assumptions that we typically make for a planar homography: either the scene is so far away such that the difference in depth can be considered negligible for non-pure rotations. The other is that the homography undergoes pure rotation. However, these conditions may not always hold true. This may lead assumptions breaking down – thus making the homography insufficient.

Assumption of points as being far away with no depth variation may result in sufficient noise: leading to an A matrix that is not full rank. The homography will end up not being representative only the trivial solution can answer Ah=0.

Q1.4.4

First we can define a line in 3-Dimensions by the parameteriztion of two separate lines (15), (16).

$$(15) \quad x = m_z z + b_x$$
$$(16) \quad y = m_y z + b_y$$

Next, generate the prospective projection equations, by using the focal length and 3D coordinates. Note here that x_o would correspond to the 3D coordinate and f is the focal length (17), (18).

$$(17) \quad x_i = \left(\frac{f}{z_o}\right) x_0$$
$$(18) \quad y_i = \left(\frac{f}{z_o}\right) y_o$$

From there you will substitute (17), (18) into (15), (16) as an intermediate step.

$$(19) \quad \frac{x_i z_o}{f} - m_x z_o = b_x$$
$$(20) \quad \frac{x_y z_o}{f} - m_y z_o = b_y$$

Divide (19) by (20), and after simplying we see that the equation for a 3D has been transformed into a 2D line, and lines have been maintained.

$$(21) \quad \frac{\frac{x_i z_o}{f} - m_x}{\frac{x_y z_o}{f} - m_y} = \frac{b_x}{b_y} \quad \Rightarrow \quad x_i = m y_i + b$$

Source: [hw2_line2line.pdf (ucsb.edu)]

Q2.1.1

FAST Detector

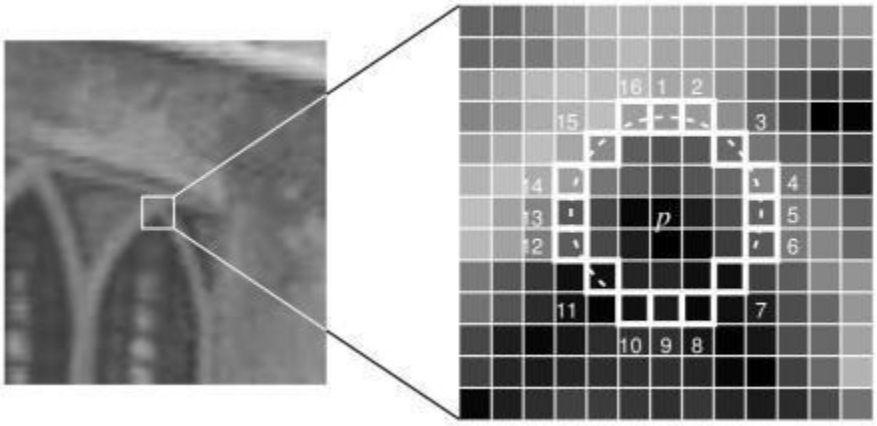


Fig 1. *Visual depiction of a fast detector*

The basic structure of the FAST detector is that it will try to identify an interest point p by searching 16 pixels, arranged in a circle surrounding the pixel of interest, to see if it meets a certain threshold. What makes it especially fast is that it will first check pixels along what constitutes the vertical and horizontal extremes (pixels 1, 5, 9, 13).

This is in constrast to the Harris Corner detector, which will take the sum squared difference between two patches around the interest point p. However, this has to be done for a window, so for each pixel you are guaranteed to have at least 9 computations where as in the FAST detector if you do not have a corner (which is often the case) you have but three computations.

Source: [Introduction to FAST (Features from Accelerated Segment Test) | by Deepanshu Tyagi | Data Breach | Medium]

Q2.1.2

Just to provide context, BRIEF reconstructs the local image patch into a binary feature descriptor. Unfortunately, it is not rotationally or scale invariant. SIFT, on the otherhand, is invariant to both of these variables. The SIFT is a feature bank of all gradients within a region, which displays as a histogram – a stark distnace to a binary descriptor such as BRIEF. SIFT is protected by a patent and therefore you must purchase the rights to use for any algorithms.

Source: Introduction to BRIEF(Binary Robust Independent Elementary Features) | by Deepanshu Tyagi | Data Breach | Medium

Q2.1.3

The hamming is a distance measurement that is typically used to compare bitwise values. A XOR operator is used to compare the two values: whatever values the strings do not share are added into a collective sum. The larger this sum the more dissimilar the strings are. This makes more sense for this current application because standard Euclidean distance doesn't have much relevance between bits or strings. The concept of classic Euclidean distance makes very little sense within the comparison of bitwise strings. Since we are using bits to describe our features within the BRIEF descriptor, it is more appropriate to use hamming distance here.

Source: What is Hamming Distance? (tutorialspoint.com)

Q2.1.4

```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]
    """

    # print("Computing Matches...\n")

    ratio = opts.ratio  # 'ratio for BRIEF feature descriptor'
    sigma = opts.sigma  # 'threshold for corner detection using FAST feature detector'

    Create Jira Issue
    # TODO: Convert Images to GrayScale
    I1g = skimage.color.rgb2gray(I1)
    I2g = skimage.color.rgb2gray(I2)

    Create Jira Issue
    # TODO: Detect Features in Both Images
    locs1_temp = corner_detection(I1g, sigma)
    locs2_temp = corner_detection(I2g, sigma)

    Create Jira Issue
    # TODO: Obtain descriptors for the computed feature locations
    desc1, locs1 = computeBrief(I1g, locs1_temp)
    desc2, locs2 = computeBrief(I2g, locs2_temp)

    Create Jira Issue
    # TODO: Match features using the descriptors
    matches = briefMatch(desc1, desc2, ratio)

    print(
        f"Matches: {matches.shape[0]} | Locs 1: {locs1.shape[0]} | Locs 2: {locs2.shape[0]}\n")

    # Flip y,x to x, y
    locs1 = np.fliplr(locs1)
    locs2 = np.fliplr(locs2)

    return matches, locs1, locs2
```

Q2.1.5

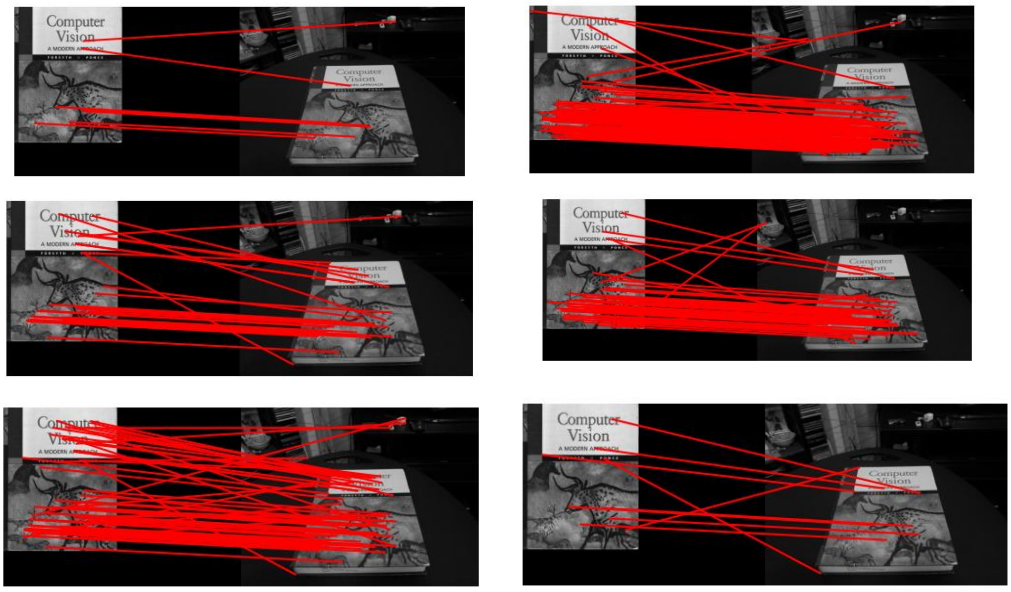


Fig 2. *Displayed matches with varied ratio and sigma parameters. Left side corresponds to a constant sigma (0.15) with varied ratio (0.6, 0.7, 0.8 from top to bottom) and the right column has a constant ratio (0.7) with a varied sigma (0.05, 0.1, 0.2 from top to bottom).*

Table 1. *Table of ablation study for varied sigma and ratio parameters for corner detecting and matching.*

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Trial 6**
Sigma	0.15	0.15	0.15	0.05	0.10	0.20
Ratio	0.6	0.7	0.8	0.7	0.7	0.7
Output:	Matches: 7 Locs 1: 945 Locs 2: 454	Matches: 27 Locs 1: 945 Locs 2: 454	Matches: 62 Locs 1: 945 Locs 2: 454	Matches: 119 Locs 1: 4673 Locs 2: 2555	Matches: 60 Locs 1: 1813 Locs 2: 811	Matches: 9 Locs 1: 601 Locs 2: 283
Comments:	Minimal matches, outliers seen	Significantly more matches, more outliers	Numerous amount of matches, even more outliers	Numerous interest points being detected, some outliers	Less interest points being detected, less outliers	Even less matches and interest points being detected

Comments:

While some comments were given in the table, a general summary is that as one increases the value of the ratio more matches are made for the same corners detected. The threshold for detecting a match is lowered resulting in more outliers. Increasing the ratio results in more corners being detected, thus more matches being found. Since you are increasing the amount of matches to be found the computation for such also takes longer.

Q2.1.6

Brief Rotations Code

```python
def rotTest(opts):

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

    # 2x36 array with reach row [rotation, # of matches]
    matchPerRot = np.zeros(shape=(2, 36))

    for i in range(36):

        # Rotate Image (by 10 degrees each pass)
        rotation = i * 10
        img_rotated = scipy.ndimage.rotate(img, rotation)

        # Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(img, img_rotated, opts)

        # display matches at 10 deg, 100 deg, 200 deg (Visualize)
        # if i in [1, 10, 20]:
        #     plotMatches(img, img_rotated, matches, locs1, locs2)

        # Update histogram
        matchPerRot[0, i] = rotation
        matchPerRot[1, i] = matches.shape[0]

    # Display histograms
    plt.bar(matchPerRot[0, :], matchPerRot[1, :])

    plt.xlabel("Rotation (Degrees)")
    plt.ylabel("Number of Matches")
    plt.title("Number of Matches versus Rotation of Image")
    plt.show()
```

Histogram & Plotted Matches

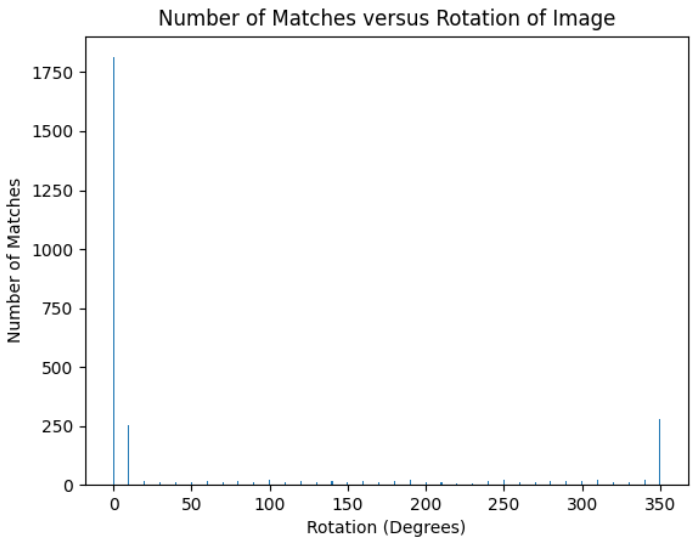


Fig 3. *Number of matches for degree rotation, with sigma = 0.10, ratio = 0.75*

Q2.1.6

Rotated Images

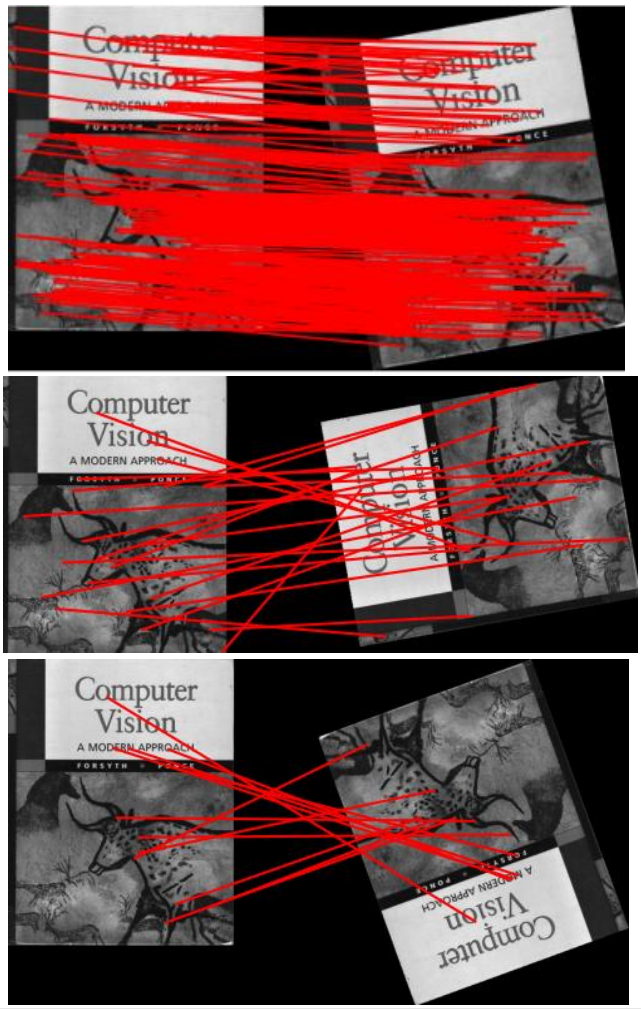


Fig 4. *Number of matches for degree rotation, with sigma = 0.10, ratio = 0.75. Top left is 10 degree, top right 100 degree, and bottom right 200 degree.*

Comments

It is abundantly clear that the BRIEF descriptor is not rotationally invariant. There seems to be a small rotation for which some matche s can be computed (±10 degrees) where the brief detector can still recogize a large portion of the corners detected between images. I think the reason for this is that the Hamming distance with binary descriptors have no concept of orientation, it is simply a similarity score. Overwhelming though, the BRIEF descriptor seems to be rotationally invariant.

Q2.2.1

```python
def computeH(xSource, xDes):
    # Q2.2.1
    # Compute the homography between two sets of points
    """
    Args:
        X1 (np.array): Points of interest in the source image
        X2 (np.array): Points of interest in the desination image

    Returns:
        H2to1: Homography matrix from source to destination.
    """
    #? ------- Steps ----------?#
    """
    1. Init the A matrix (2n x 9), where n is the total number of points we have
    [For Each Point Pair]
    2. Find the x, y values from the source and destination image
    3. Each point pair has the following two rows:

    a. [x2, y2, 1, 0, 0, 0, -x1x2 , -x1y2, -x1]
        [0, 0, 0, x2, y2, 1, -y1x2, -y1y2, -y1]

    4. Append to A matrix
    5. Vstack all match pairs
    6. Perform SVD on the A matrix to get U, Sigma, V.T
    7. Take the last column of the V.T matrix, which should correspond to the eigenvectors with the
    eigenvalue of (ideally) 0. The nontrival solution to Ah=0.
        - SVD gives v.T, and v is the eigvecs of A.T * A.
        - Transpose v, then find the last column
    """

    # 1
    n = len(xSource)
    A = []

    for i in range(n):

        # 2
        x1, y1 = xSource[i, 0], xSource[i, 1]
        x2, y2 = xDes[i, 0], xDes[i, 1]

        # 3
        pairCols = np.array([[x2, y2, 1, 0, 0, 0, -x1*x2, -x1*y2, -x1],
                             [0, 0, 0, x2, y2, 1, -y1*x2, -y1*y2, -y1]])

        # 4
        A.append(pairCols)

    # 5
    A = np.vstack(A)

    # 6
    u, s, vT = np.linalg.svd(A)

    # 7
    v = vT.T
    h = v[:, -1]

    # 8
    H2to1 = np.reshape(h, newshape=(3, 3))

    return H2to1
```

Q2.2.2

```python
def computeH_norm(x1, x2):
    # Q2.2.2
    """ Computes the homography, but with normalized match points.

    Args:
        X1 (np.array): Points of interest in the source image
        X2 (np.array): Points of interest in the destination image

    Returns:
        H2to1: Homography matrix from source to destination.
    """
    #? ------- Steps ------?#
    """
    Steps:
    1. Compute the mean, or centroid of the x1, x2 points
        - sum along all rows, divide by number of rows
        - divide by total num of points
    2. Subtract the centroid from all values of x1, x2
        - leverage broadcasting
    3. Find the average scale
        a. average distance of each vector normalized to sqrt(2)
        b. max dist is sqrt(2)
    4. Multiply points by the mean
    5. Generate similarity transforms T1, T2
    6. Compute the normalized homography
    7. Denormalize the homography with T1 and T2


    """

    # 1
    centroid1 = np.sum(x1, axis=0) / x1.shape[0]
    centroid2 = np.sum(x2, axis=0) / x2.shape[0]

    # 2
    x1_N = x1 - centroid1
    x2_N = x2 - centroid2

    # 3
    scale1 = np.sqrt(
        2) / np.max(np.linalg.norm(x1_N, axis=1), axis=0)
    scale2 = np.sqrt(
        2) / np.max(np.linalg.norm(x2_N, axis=1), axis=0)

    # 4
    x1_N *= scale1
    x2_N *= scale2

    # 5
    T1 = np.array([[scale1, 0, (-scale1*centroid1)[0]],
                   [0, scale1, (-scale1*centroid1)[1]],
                   [0, 0, 1]])

    T2 = np.array([[scale2, 0, (-scale2*centroid2)[0]],
                   [0, scale2, (-scale2*centroid2)[1]],
                   [0, 0, 1]])

    # 6
    H = computeH(x1_N, x2_N)

    # 7
    invT1 = np.linalg.inv(T1)
    H2to1 = np.matmul(invT1, np.matmul(H, T2))

    return H2to1
```

Q2.2.3

```python
def computeH_ransac(locs1, locs2, opts):
    """ Using RANSAC to compute the best homography between source and destination.

    Args:
        locs1 (np.array): Ordered list of matched points within the source image (Nx2).
        locs2 (np.array): Ordered list of matched points within the destination image (Nx2).
        opts (Namespace): Holds the parameters for RANSAC: mat_inters to run RANSAC for and the error threhold to
                          to determine inliers.

    Returns:
        bestH2to1 (np.array): Best homography (3x3) calculated from inliners obtained from max_iters iterations of RANSAC.
    """
    # 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
    # the tolerance value for considering a point to be an inlier
    inlier_tol = opts.inlier_tol

    # print("Performing RANSAC to compute the homography...\n")

    # 1.
    N = locs2.shape[0]
    inliers = np.zeros(shape=(N, 1))

    # tqdm.tqdm(range(max_iters), desc="Running RANSAC: "):
    for i in range(max_iters):

        # 2.
        # randInd = np.random.randint(0, N, size=4)
        randInd = np.array(rd.sample(range(N), 4))
        rand_locs1 = locs1[randInd]
        rand_locs2 = locs2[randInd]

        # 3.
        tempH2to1 = computeH_norm(rand_locs1, rand_locs2)  # change source/dest

        # 4.
        # 4.1
        Hom_locs1 = np.hstack((locs1, np.ones(shape=(N, 1))))
        Hom_locs2 = np.hstack((locs2, np.ones(shape=(N, 1))))
        # 4.2
        estimated_locs1 = np.dot(tempH2to1, Hom_locs2.T)
        # 4.3
        eLocs1Norm = (estimated_locs1 / estimated_locs1[2, :]).T
        # 4.4
        temp_inliers = np.linalg.norm(
            (Hom_locs1 - eLocs1Norm), axis=1)
        temp_inliers = temp_inliers < inlier_tol
        temp_inliers = temp_inliers.astype(np.int8)

        # 5.
        highestNumInliers = inliers[inliers == 1].shape[0]
        currNumInliers = temp_inliers[temp_inliers == 1].shape[0]

        # 6.
        if currNumInliers > highestNumInliers:
            inliers = temp_inliers
```

Q2.2.4: Code

warpImage(opts) Function

```python
def warpImage(opts):

    # Open images and opts
    opts = get_opts()
    image1 = cv2.imread('../data/cv_cover.jpg')
    image2 = cv2.imread('../data/cv_desk.png')

    # match features
    matches, locs1, locs2 = matchPics(image1, image2, opts)

    # order features
    x1, x2 = orderMatches(matches, locs1, locs2)

    # compute homography
    H2to1, inliers = computeH_ransac(x1, x2, opts)

    # import the harry potter image
    imageHP = cv2.imread('../data/hp_cover.jpg')
    imageHP = cv2.resize(imageHP, dsize=(image1.shape[1], image1.shape[0]))

    # find composite img
    compositeImg = compositeH(H2to1, imageHP, image2)

    # show the image
    # stitch images together and visualize
    cv2.imshow("Source", imageHP)
    cv2.imshow("Destination", image2)
    cv2.imshow("Warped Source", compositeImg)
    cv2.waitKey(0)
    cv2.destroyAllWindows()


if __name__ == "__main__":

    opts = get_opts()
    warpImage(opts)
```

compositeH(H, template, img) Function

```python
def compositeH(H2to1, source, destination):
    """Shows a warp from the source to the destination.

    Args:
        H (np.array): 3x3 homography matrix
        source (np.array): source image to be mapped to the destination
        destination (np.array): dest image to be mapped to
    """

    # find the inverse for 2 to 1
    # print("Inverse the homography H2to1 -> H1to2 ...")
    H = np.linalg.inv(H2to1)

    # warp view
    warpedImg = cv2.warpPerspective(
        source, H, dsize=(destination.shape[1], destination.shape[0]))

    # find where the warped image is equal to 0
    zeroInd = np.where(warpedImg == 0)
    # replace the 0 values from the warped with values from the destination
    warpedImg[zeroInd] = destination[zeroInd]

    return warpedImg
```

Q2.2.4: Result

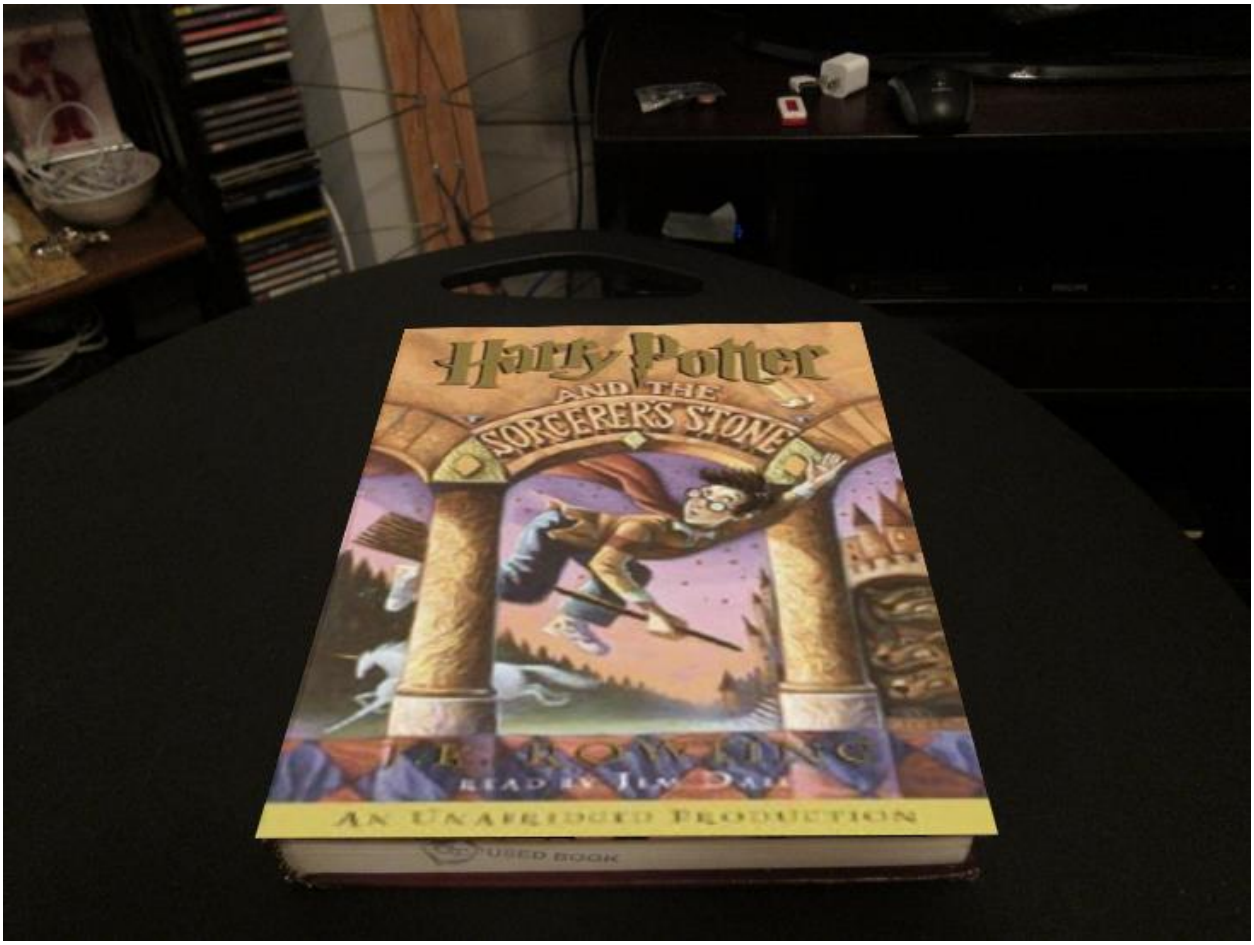


Fig 5. *Harry Potterized CV book Cover (sigma = 0.1, ratio = 0.75).*

2.2.5

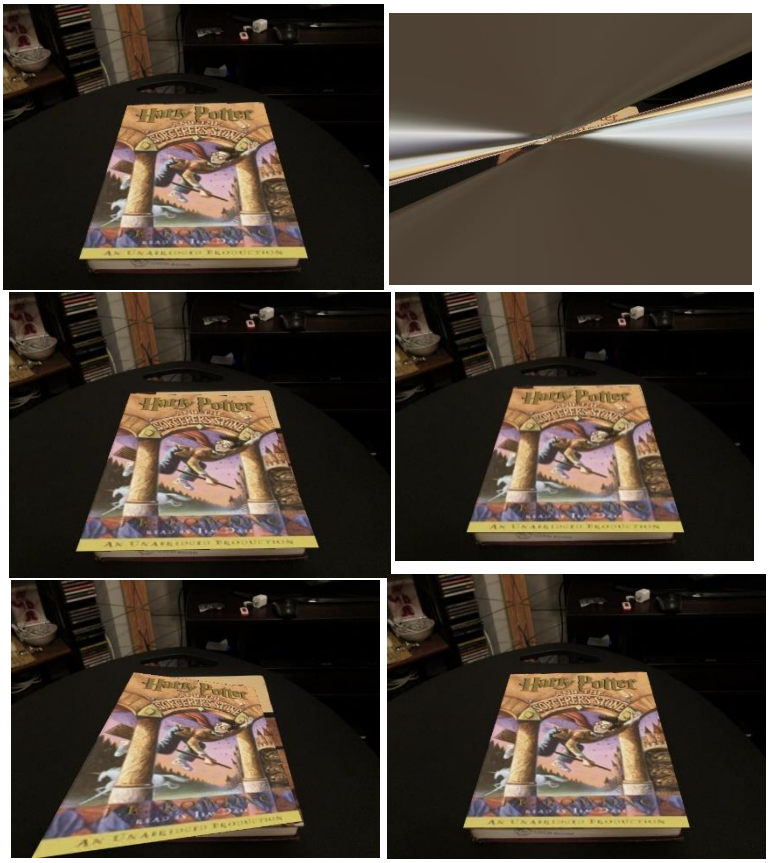


Fig 6. *Harry Potterized books with varied max iteration and tolerance parameters. Left side corresponds to a constant iterations (500) with varied tolerances (1, 50, 100 from top to bottom) and the right column has a constant tolerance () with a varied sigma (5, 50, 250 from top to bottom).*

Table 2. *Table of ablation study for varied max iteration and tolerance volues for RANSAC (sigma = 0.15, ratio = 0.90).*

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Trial 6**
Max_iters	500	500	500	5	50	250
Tolerance	1	50	100	2	2	2
Inliers/Matches	32/117	68/117	85/117	4/117	34/117	41/117
Comments:	Good Transformation	Slightly Offset	Noticably Different	Unrecognizable Warp	Slightly Offset	Good Transformation

Q2.2.5

Comments

As a general trend, if one makes the tolerances too high the warps become unrepresentative, with the inverse happening for max iterations. With too low a tolerance, the warps become unrecognizable. This is an intuitive idea: if one increases the tolerance too high the ability to filter out potential outliers is lowered and if one decreases the iterations the probability one will select an outlier as a random point to generate the homography increases.

Also it would stand to reason that depending on the amount of outliers one expects, these values will shift around. If you expect more outliers you may want to increase iterations. If your data can handle some noise, the tolerance could be increased – and so on.

Q3.1: Code

```python
def preprocessImg(image, shape):

    #!!! preprocess image so that black values don't appear in the image
    zeroInd = np.where(image == 0)
    image[zeroInd] = 1

    image = cv2.resize(image, dsize=shape)

    return image


def arSuperimpose(path: str):
    """ Superimpose a mov file onto the book of the cv book scene."""

    # ? ------------------- Steps ----------------------------

    """

    1. Open both videos & the CV cover page
        1.a Remove the black bars from the video
            - Find where the black bars are on the first frame
            - Crop the first frame
            - Make all the rest of the frames the same size
        1.b Each frame of the video to be warped to be the same size as the book
            - Take the pixels from the cv book and only take those from the video
    2. Find which one is shorter: make the final video that many frames
        2.a find the num frames for each vid
        2.b Which ever video is shorter, clip the longer frame to match
        2.c Change black pixels from warp video to slighly less black
    3. Match the cv_book to each frame of the cv_video
        - Input: I1, I2, opts
        - Output: matches, locs1, locs2
    4. Order the matches from 3
        - Input: matches, locs1, locs2
        - Output: x1, x2
    5. Compute the homography of each frame with x1, x2
        - Input: Locs1, Locs2, opts
        - Output: H2to1
    6. Using that homography calculated in 3, apply it to each kunfu panda frame
    7. Save the video

    """
```

Q3.1: Code

```
opts = get_opts()
n_cpu = multiprocessing.cpu_count()

# 1.
print("Openiing the videos...")
warp_vid = loadVid(path)
cv_video = loadVid("../data/book.mov")
cv_book = cv2.imread("../data/cv_cover.jpg")
print("Done opening the videos!")

# 1.a
print("Resizing the video to be warped...")
frame0 = warp_vid[0, :, :, :]
y_nonzero, x_nonzero, _ = np.nonzero(frame0 > 20)
warp_vid = warp_vid[:, np.min(y_nonzero):np.max(y_nonzero),
                    np.min(x_nonzero):np.max(x_nonzero), :]

# 1.b
warpCenterY = int(warp_vid.shape[0] / 2)

aspectRatio = cv_book.shape[0] / cv_book.shape[1]
numCols = int(warp_vid.shape[1] / aspectRatio)
cropLeft = warpCenterY
cropRight = warpCenterY + (numCols)

# 1.c
warp_vid = warp_vid[:, :, cropLeft:cropRight, :]
print("Completed resizing!")

# 2.
# 2.a
warp_frames = warp_vid.shape[0]
cv_frames = cv_video.shape[0]

# 2.b
totalFrames = 0
if warp_frames > cv_frames:
    warp_vid = warp_vid[0:cv_frames, :, :, :]
    totalFrames = cv_frames
else:
    cv_video = cv_video[0:warp_frames, :, :, :]
    totalFrames = warp_frames
```

Q3.1: Code

```
    # # 2.c
    print("Preprocessing the Ar video...")
    frames = [(warp_vid[i, :, :, :], (cv_book.shape[1], cv_book.shape[0]))
              for i in range(totalFrames)]
    with multiprocessing.Pool(processes=n_cpu) as pool:
        results = pool.starmap(preprocessImg, frames)
    pool.close()
    warp_vid = np.stack(results, axis=0)

    print("Done preprocessing the video!")

    # 3.
    print("Find the match points between all frames...")
    matchParams = [(cv_book, cv_video[i, :, :, :], opts)
                   for i in range(totalFrames)]
    with multiprocessing.Pool(processes=n_cpu) as pool:
        results = pool.starmap(matchPics, matchParams)
    pool.close()
    print("Done finding matches!")

    # 4.
    print("Order the matches between pictures...")
    orderParams = results
    with multiprocessing.Pool(processes=n_cpu) as pool:
        results = pool.starmap(orderMatches, orderParams)
    pool.close()
    print("Done ordering matches!")

    # 5.
    print("Find the Homography between image 2 and 1...")
    orderResults = results
    hParams = [(match[0], match[1], opts) for match in orderResults]
    with multiprocessing.Pool(processes=n_cpu) as pool:
        results = pool.starmap(computeH_ransac, hParams)
    pool.close()
    print("Done finding homographies!")

    # 6.
    print("Superimposing video onto CV Book scene...")
    H2to1_all = results
    finalParams = [(H2to1_all[i][0], warp_vid[i, :, :, :], cv_video[i, :, :, :])
                   for i in range(totalFrames)]

    with multiprocessing.Pool(processes=n_cpu) as pool:
        results = pool.starmap(compositeH, finalParams)
    pool.close()
    print("Done generating video... saving result...")

    # 7.
    finalVideo = np.stack(results, axis=0)

    out = cv2.VideoWriter('../data/outpy.avi', cv2.VideoWriter_fourcc(
        'M', 'J', 'P', 'G'), 24, (finalVideo.shape[2], finalVideo.shape[1]))

    for i in range(totalFrames):
        out.write(finalVideo[i, :, :, :])
    out.release()


if __name__ == "__main__":
    arSuperimpose("../data/ar_source.mov")
```

Q3.1: Video Output

Fig 7. *AR video, with Kung Fu Panda video super imposed on the CV book cover, for frames at the beginning, middle, and end (top to bottom) and sigma = 0.1, ratio = 0.75.*

Video Link: *https://drive.google.com/file/d/1WclaRsEhb_N0qmuDUejQVarETCbOoiYM/view?usp=sharing*

Q4: Panorama

Image 1

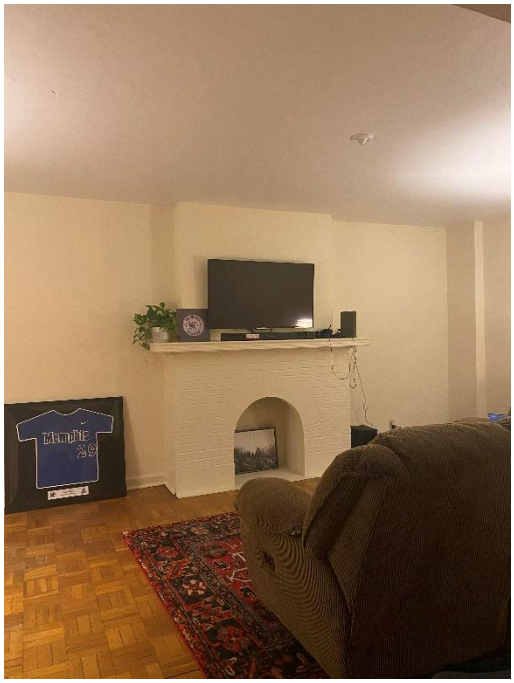
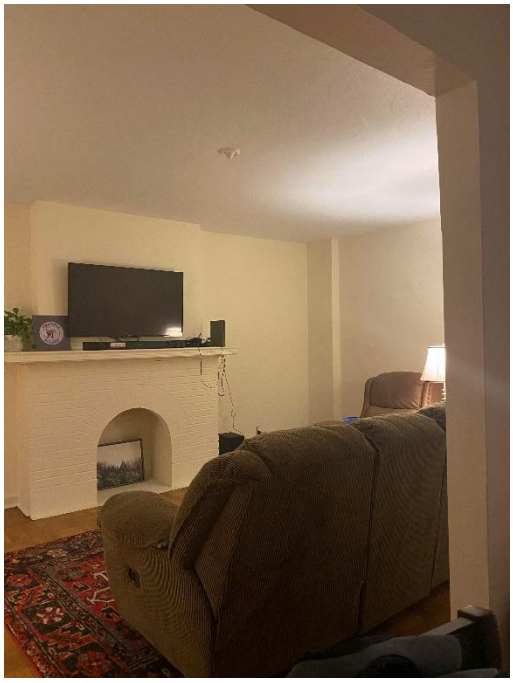


Image 2



Stitched Panorama

Q4: Code

```python
def rescaleImg(img, scale_percent):
    width = int(img.shape[1] * scale_percent / 100)
    height = int(img.shape[0] * scale_percent / 100)
    dim = (width, height)

    # resize image
    resized = cv2.resize(img, dim, interpolation=cv2.INTER_AREA)

    return resized


def intP2coords(interestPoints: dict):
    """ Transforms the image coordantes mapped using cpselect to x,y locations for image 1 and image 2.

    Args:
        interestPoints (dict): dict returned from cpselect, containing all the manually selected matches from image 1 and image 2.

    Returns:
        x1: np.array of the x, y locations coming from image 1
        x2: np.array of the x, y locations coming from image 2
    """

    # ? ---------------- Steps -----------------------------
    """

    1. Establish how many interest points were selected
    2. generate x1, x2 points of length = # interest points
    3. [For i interest points] extract x, y points and save them to x1, x2

    """
    # 1
    N = len(interestPoints)

    # 2
    x1 = np.zeros(shape=(N, 2))
    x2 = np.zeros(shape=(N, 2))

    # 3
    for i in range(N):

        x1[i, 0], x1[i, 1] = interestPoints[i]["img1_x"], interestPoints[i]["img1_y"]

        x2[i, 0], x2[i, 1] = interestPoints[i]["img2_x"], interestPoints[i]["img2_y"]

    return x1, x2
```

Q4: Code

```python
def panorama(image1_dest, image2_dest, show_matches=False):
    """ Given the file name of two images, generate and display a panorama.

    Args:
        image1_dest (str): File name, must relative to folder as follows: '../data/image1_dest')
        image2_dest (str): File name, must relative to folder as follows: '../data/image1_dest')
    """

    # ? ----------------- Steps ----------------------------

    """
    1. Open the images
    2. Find the interest points manually
    3. Change the dict to two np.arrays that work with functions from planarH
    4. Find the homography
    5. Apply the homography to image 2
    6. Rescale the images to reasonable sizes (15% of original)
    7. Replace all the black pixels with image 1
    8. Show the panorama

    """

    # import opts
    opts = get_opts()

    # 1
    image1 = cv2.imread(filename="../data/" + image1_dest)
    image2 = cv2.imread(filename="../data/" + image2_dest)

    # 2
    interestPoints = cpselect(
        img_path1="../data/" + image1_dest, img_path2="../data/" + image2_dest)

    #! good interest points from one trial run
    # interestPoints = [{'point_id': 1, 'img1_x': 2174.615105346898, 'img1_y': 1994.8799720360635, 'img2_x': 1393.670078588
    #     'point_id': 12, 'img1_x': 1553.7231522105976, 'img1_y': 3268.7394773636756, 'img2_x': 736.1202979605623, 'img2_y'

    # 3
    x1, x2 = intP2coords(interestPoints)

    # 4
    H2to1, inliers = computeH_ransac(x1, x2, opts)

    # 5
    image2_warped = cv2.warpPerspective(image2, H2to1, dsize=(
        image1.shape[1], image1.shape[0]))

    # 6
    image2_warped = rescaleImg(image2_warped, 15)
    image1 = rescaleImg(image1, 15)

    # 7
    zeroInd = np.where(image2_warped == 0)
    image2_warped[zeroInd] = image1[zeroInd]

    # 8
    cv2.imshow("image 2 warped", image2_warped)
    cv2.waitKey(0)
    cv2.destroyAllWindows()


if __name__ == "__main__":
    panorama("livingroom_left.jpg", "livingroom_right.jpg")
```