HOMEWORK 3: AUGMENTED REALITY WITH PLANAR HOMOGRAPHIES

16-720A Computer Vision (Spring 2023)
https://canvas.cmu.edu/courses/32966

OUT: March 3rd, 2023
DUE: March 27th, 2023 11:59 PM
Instructor: Deva Ramanan
TAs: Kangle Deng, Vidhi Jain, Xiaofeng Guo, Chung Hee Kim, Ingrid Navarro

Instructions/Hints

- Please refer to the course logistics page for information on the **Collaboration Policy** and **Late Submission Policy**.
- **Submitting your work:** There will be two submission slots for this homework on **Gradescope**: Written and Programming.
  - **Write-up.** For written problems such as short answer, multiple choice, derivations, proofs, or plots, we will be using the written submission slot. Please use this provided template. **We don't accept handwritten submissions.** Each answer should be completed in the boxes provided below the question. You are allowed to adjust the size of these boxes, but **make sure to link your answer to each question when submitting to Gradescope**. Otherwise, your submission will not be graded.
  - **Code.** You are also required to upload your code, which you wrote to solve this homework, to the Programming submission slot. Your code may be run by TAs so please make sure it is in a workable state. The assignment must be completed using Python 3.7 or newer. We recommend setting up a conda environment, but you are free to set up your environment however you like.
  - Regrade requests can be made after the homework grades are released, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted.
- **Start early!** This homework is difficult and may take a long time to complete.
- **Verify your implementation as you proceed.** If you don't verify that your implementation is correct on toy examples, you will risk having a huge mess when you put everything together.
- **Q&A.** If you have any questions or need clarifications, please post in Slack or visit the TAs during office hours. Additionally, we provide a **FAQ** (Section 7) with questions from previous semesters and some **Helpful Concepts** (Section 8). Make sure you read it prior to starting your implementations.

Overview

In this assignment, you will be implementing an AR application step by step using planar homographies. Before we step into the implementation, we will walk you through the theory of planar homographies. In the programming section, you will first learn to find point correspondences between two images and use these to estimate the homography between them. Using this homography you will then warp images and finally implement your own AR applications.

1 Preliminaries

1.1 Planar Homographies as a Warp

Recall that a planar homography is a warp operation, which represents a mapping from pixel coordinates from one camera frame to another. It makes a fundamental assumption of the points lying on a plane in the real world. Under this particular assumption, pixel coordinates in one view of the points on the plane can be directly mapped to pixel coordinates in another camera view of the same points.

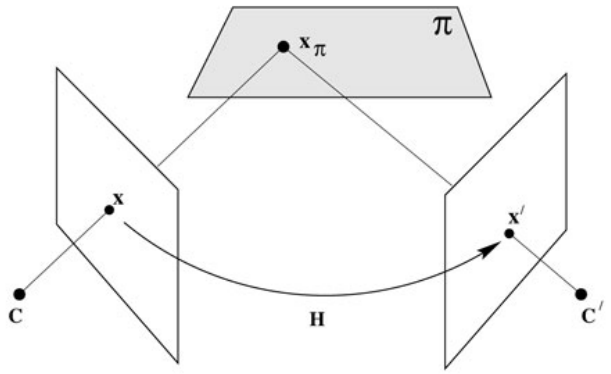


Figure 1.1: A homography **H** links all points $\mathbf{x}_\pi$ lying in plane π between two camera views **x** and **x**' in cameras C and C' respectively such that **x**' $=$ **Hx**. [*From Hartley and Zisserman*]

Q1.1 (5 points): Prove that there exists a homography **H** that satisfies equation 1.1 given two 3×4 camera projection matrices $\mathbf{P}_1$ and $\mathbf{P}_2$ corresponding to the two cameras and a plane Π. You do not need to produce an actual algebraic expression for **H**. All we are asking for is a proof of the existence of **H**.

$$\mathbf{x}_1 \equiv \mathbf{H}\mathbf{x}_2 \tag{1.1}$$

The $\equiv$ symbol stands for identical to. The points $\mathbf{x}_1$ and $\mathbf{x}_2$ are in *homogenous coordinates*, which means they have an additional dimension. If $\mathbf{x}_i$ is a 3D vector $\begin{bmatrix} x_i & y_i & z_i \end{bmatrix}^T$, it represents the 2D point $[\frac{x_i}{z_i} \quad \frac{y_i}{z_i}]$ (called *inhomogenous coordinates*). This additional dimension is a mathematical convenience to represent transformations (like translation, rotation, scaling, etc) in a concise matrix form. The $\equiv$ means that the equation is correct to a scaling factor.

Note: A degenerate case happens when the plane Π contains both cameras' centers, in which case there are infinite choices of **H** satisfying equation 1.1. You can ignore this special case in your answer.

Q1.1

$$\lambda_1 \begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = P_1 \begin{bmatrix} x_\pi^i \\ y_\pi^i \\ 1 \end{bmatrix} \tag{1.2}$$

$$\lambda_2 \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix} = P_2 \begin{bmatrix} x_\pi^i \\ y_\pi^i \\ 1 \end{bmatrix} \tag{1.3}$$

$$P_1^{-1}\lambda_1 \begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = \begin{bmatrix} x_\pi^i \\ y_\pi^i \\ 1 \end{bmatrix} \tag{1.4}$$

$$P_2^{-1}\lambda_2 \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix} = \begin{bmatrix} x_\pi^i \\ y_\pi^i \\ 1 \end{bmatrix} \tag{1.5}$$

From eq. (1.4) and (1.5)

$$P_1^{-1}\lambda_1 \begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = P_2^{-1}\lambda_2 \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix} \tag{1.6}$$

$$\begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = \lambda P_1 P_2^{-1} \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix} \text{ where } \lambda = \frac{\lambda_2}{\lambda_1} \tag{1.7}$$

Hence from above, we can see that there exists an $H = \lambda P_1 P_2^{-1}$ that satisfies the eq. (1.1)

1.2 The Direct Linear Transform

A very common problem in projective geometry is often of the form $\mathbf{x} \equiv \mathbf{A}\mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are known vectors, and $\mathbf{A}$ is a matrix which contains unknowns to be solved. Given matching points in two images, our homography relationship clearly is an instance of such a problem. Note that the equality holds only *up to scale*, *i.e.*, the set of equations are of the form $\mathbf{x} = \lambda\mathbf{H}\mathbf{x}'$. This is why we cannot use an ordinary least squares solution such as what you may have used in the past to solve simultaneous equations. A standard approach to solve these kinds of problems is called the Direct Linear Transform, where we rewrite the equation as proper homogeneous equations which are then solved in the standard least squares sense. Since this process involves disentangling the structure of the $\mathbf{H}$ matrix, it's a *transform* of the problem into a set of *linear* equation, thus giving it its name.

Let $\mathbf{x_1}$ be a set of points in an image and $\mathbf{x_2}$ be the set of corresponding points in an image taken by another camera. Suppose there exists a homography $\mathbf{H}$ such that:

$$\mathbf{x_1^i} \equiv \mathbf{H}\mathbf{x_2^i} \quad (i \in \{1 \dots N\})$$

where $\mathbf{x_1^i} = [x_1^i \quad y_1^i \quad 1]$ are in homogenous coordinates, $\mathbf{x_1^i} \in \mathbf{x_1}$ and $\mathbf{H}$ is a 3×3 matrix. For each point pair, this relation can be rewritten as

$$\mathbf{A_i h} = 0$$

where $\mathbf{h}$ is a column vector reshaped from $\mathbf{H}$, and $\mathbf{A_i}$ is a matrix with elements derived from the points $\mathbf{x_1^i}$ and $\mathbf{x_2^i}$. This can help calculate $\mathbf{H}$ from the given point correspondences.

Q1.2.1 (3 points): How many degrees of freedom does $\mathbf{h}$ have?

Q1.2.1

h has 9 variables but since homography is uniquely determined up to an irrelevant scalar factor, h has **8** degrees of freedom

Q1.2.2 (2 points): How many point pairs are required to solve $\mathbf{h}$?

Q1.2.2

Since we need to find values of **8 unknowns** of h and **each pair of** x_2 **to** x_1 **mapping gives two constraints**, we would require **4 point pairs** to solve h

Q1.2.3 (5 points): Derive $\mathbf{A_i}$.

Q1.2.3

$$\begin{bmatrix} x_1^i \\ y_1^i \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_2^i \\ y_2^i \\ 1 \end{bmatrix} \tag{1.8}$$

$$x_1^i = \frac{h_{11}x_2^i + h_{12}y_2^i + h_{13}}{h_{31}x_2^i + h_{32}y_2^i + h_{33}}, \; y_1^i = \frac{h_{21}x_2^i + h_{22}y_2^i + h_{23}}{h_{31}x_2^i + h_{32}y_2^i + h_{33}} \tag{1.9}$$

$$x_1^i[h_{31}x_2^i + h_{32}y_2^i + h_{33}] - [h_{11}x_2^i + h_{12}y_2^i + h_{13}] = 0 \tag{1.10}$$

$$y_1^i[h_{31}x_2^i + h_{32}y_2^i + h_{33}] - [h_{21}x_2^i + h_{22}y_2^i + h_{23}] = 0 \tag{1.11}$$

Writing equation (1.10), (1.11) in terms of matrix multiplication

Q1.2.3

$$\begin{bmatrix} -x_2^i & -y_2^i & -1 & 0 & 0 & 0 & x_1^i x_2^i & x_1^i y_2^i & x_1^i \\ 0 & 0 & 0 & -x_2^i & -y_2^i & -1 & y_1^i x_2^i & y_1^i y_2^i & y_1^i \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = 0 \tag{1.12}$$

Hence from above we can see that A_i is as follows

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

Q1.2.4 (5 points): When solving $\mathbf{Ah} = 0$, in essence you're trying to find the $\mathbf{h}$ that exists in the null space of $\mathbf{A}$. What that means is that there would be some non-trivial solution for $\mathbf{h}$ such that the product $\mathbf{Ah}$ turns out to be 0. What will be a trivial solution for $\mathbf{h}$? Is the matrix $\mathbf{A}$ full rank? Why/Why not? What impact will it have on the singular values (i.e. eigenvalues of $\mathbf{A}^\top\mathbf{A}$)?

Q1.2.4

The trivial solution would be $h = 0$ i.e. all variables in the h vector set to 0. We know that when solving for $Ah = 0$, there would be some non-trivial solution h such that Ah becomes 0, then there must be a non-zero eigenvector with a corresponding eigenvalue equal to 0 (for $A^T A$). And since one of the eigenvalues is zero, the square matrix $A^T A$ is not full rank. And since $A^T A$ is not full rank then A is not full rank as well, since rank($A^T A$) = rank(A)

1.3 Using Matrix Decompositions to calculate the homography

A homography $\mathbf{H}$ transforms one set of points (in homogenous coordinates) to another set of points. In this homework, we will obtain the corresponding point coordinates using feature matches and will then need to calculate the homography. You have already derived that $\mathbf{Ax} = 0$ in Question 1. In this section, we will look at how to solve such equations using two approaches, either of which can be used in the subsequent assignment questions.

1.3.1 Eigenvalue Decomposition

One way to solve $\mathbf{Ax} = 0$ is to calculate the eigenvalues and eigenvectors of $\mathbf{A}$. The eigenvector corresponding to 0 is the answer for this. Consider this example:

$$\mathbf{A} = \begin{bmatrix} 3 & 6 & -8 \\ 0 & 0 & 6 \\ 0 & 0 & 2 \end{bmatrix}$$

Using the `numpy.linalg` function `eig()`, we get the following eigenvalues and eigenvectors:

$$\mathbf{V} = \begin{bmatrix} 1.0000 & -0.8944 & -0.9535 \\ 0 & 0.4472 & 0.2860 \\ 0 & 0 & 0.0953 \end{bmatrix}$$
$$\mathbf{D} = \begin{bmatrix} 3 & 0 & 2 \end{bmatrix}$$

Here, the columns of $\mathbf{V}$ are the eigenvectors and each corresponding element in $\mathbf{D}$ it's eigenvalue. We notice that there is an eigenvalue of 0. The eigenvector corresponding to this is the solution for the equation $\mathbf{Ax} = 0$.

$$\mathbf{A} = \begin{bmatrix} 3 & 6 & -8 \\ 0 & 0 & 6 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} -0.8944 \\ 0.4472 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

1.3.2 Singular Value Decomposition

The Singular Value Decomposition (SVD) of a matrix $\mathbf{A}$ is expressed as:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

Here, $\mathbf{U}$ is a matrix of column vectors called the *left singular vectors*. Similarly, $\mathbf{V}$ is called the *right singular vectors*. The matrix $\mathbf{\Sigma}$ is a diagonal matrix. Each diagonal element σ_i is called the *singular value* and these are sorted in order of magnitude. In our case, it is a 9×9 matrix.

- If $\sigma_9 = 0$, the system is *exactly-determined*, a homography exists and all points fit exactly.
- If $\sigma_9 \geq 0$, the system is *over-determined*. A homography exists but not all points fit exactly (they fit in the least-squares error sense). This value represents the goodness of fit.
- Usually, you will have at least four correspondences. If not, the system is *under-determined*. We will not deal with those here.

The columns of $\mathbf{U}$ are eigenvectors of $\mathbf{AA}^T$. The columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{A^TA}$. We can use this fact to solve for $\mathbf{h}$ in the equation $\mathbf{Ah} = 0$. Using this knowledge, let us reformulate our problem of

solving $\mathbf{Ax} = 0$. We want to minimize the error in solution in the least-squares sense. Ideally, the product $\mathbf{Ah}$ should be 0. Thus, the sum-squared error can be written as:

$$\begin{aligned} f(\mathbf{h}) &= \frac{1}{2}(\mathbf{Ah} - \mathbf{0})^T(\mathbf{Ah} - \mathbf{0}) \\ &= \frac{1}{2}(\mathbf{Ah})^T(\mathbf{Ah}) \\ &= \frac{1}{2}\mathbf{h}^T\mathbf{A}^T\mathbf{Ah} \end{aligned}$$

Minimizing this error with respect to $\mathbf{h}$, we get:

$$\begin{aligned} \frac{d}{d\mathbf{h}} f &= 0 \\ \implies \frac{1}{2}(\mathbf{A}^T\mathbf{A} + (\mathbf{A}^T\mathbf{A})^T)\mathbf{h} &= 0 \\ \mathbf{A}^T\mathbf{Ah} &= 0 \end{aligned}$$

This implies that the value of $\mathbf{h}$ equals the eigenvector corresponding to the zero eigen-value (or closest to zero in case of noise). Thus, we choose the smallest eigenvalue of $\mathbf{A}^T\mathbf{A}$, which is σ_9 in Σ and the least-squares solution to $\mathbf{Ah} = 0$ is the the corresponding eigen-vector (in column 9 of the matrix V).

1.4 Theory Questions

Q1.4.1 (5 points): Homography under rotation Prove that there exists a homography $\mathbf{H}$ that satisfies $\mathbf{x_1} \equiv \mathbf{Hx_2}$, given two cameras separated by a pure rotation. That is, for camera 1, $\mathbf{x_1} = \mathbf{K_1}[\mathbf{I}\ \ \mathbf{0}]\mathbf{X}$ and for camera 2, $\mathbf{x_2} = \mathbf{K_2}[\mathbf{R}\ \ \mathbf{0}]\mathbf{X}$. Note that $\mathbf{K_1}$ and $\mathbf{K_2}$ are the 3×3 intrinsic matrices of the two cameras and are different. $\mathbf{I}$ is 3×3 identity matrix, $\mathbf{0}$ is a 3×1 zero vector and $\mathbf{X}$ is a point in 3D space. $\mathbf{R}$ is the 3×3 rotation matrix of the camera.

Q1.4.1

Since X is in homogeneous form. Let $\hat{X}$ be the actual 3D point i.e. X = $\begin{bmatrix} \hat{X} \\ 1 \end{bmatrix}$

$$x_1 = K_1 \begin{bmatrix} I & 0 \end{bmatrix} X \tag{1.13}$$

$$x_1 = K_1 \hat{X} \tag{1.14}$$

$$\hat{X} = K_1^{-1} x_1 \tag{1.15}$$

$$x_2 = K_2 \begin{bmatrix} R & 0 \end{bmatrix} X \tag{1.16}$$

$$x_2 = K_2 \begin{bmatrix} R & 0 \end{bmatrix} \begin{bmatrix} K_1^{-1} x_1 \\ 1 \end{bmatrix} \tag{1.17}$$

$$x_2 = K_2 R K_1^{-1} x_1 \tag{1.18}$$

$$\tag{1.19}$$

From above it's clear that there exists a homography $H = K_2 R K_1^{-1}$ that satisfies $x_1 \equiv H x_2$ given two cameras separated by pure rotation.

Q1.4.2 (5 points): Understanding homographies under rotation Suppose that a camera is rotating about its center $\mathbf{C}$, keeping the intrinsic parameters $\mathbf{K}$ constant. Let $\mathbf{H}$ be the homography that maps the view from one camera orientation to the view at a second orientation. Let θ be the angle of rotation between the two. Show that $\mathbf{H}^2$ is the homography corresponding to a rotation of 2θ. Please limit your answer within a couple of lines. A lengthy proof indicates that you're doing something too complicated (or wrong).

Q1.4.2

Let's use the previous question's result since the camera is only rotated about its center. From the previous question, we know that the homography would be given by the following expression

$$H = K_2 R K_1^{-1}$$

Now, since the camera rotated about its center, that means the rotation is about the z-axis. so the rotation matrix $R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$ and since the camera intrinsics remain the same, then

$$H_\theta = K \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} K^{-1} = K R_\theta K^{-1} \tag{1.20}$$

for a rotation of θ. If rotation was 2θ then,

$$H_{2\theta} = K \begin{bmatrix} \cos 2\theta & -\sin 2\theta & 0 \\ \sin 2\theta & \cos 2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} K^{-1} \tag{1.21}$$

$$H_{2\theta} = K \begin{bmatrix} \cos^2\theta - \sin^2\theta & -2\sin\theta\cos\theta & 0 \\ 2\sin\theta\cos\theta & \cos^2\theta - \sin^2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} K^{-1} \tag{1.22}$$

$$H_{2\theta} = K \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} K^{-1} \tag{1.23}$$

$$H_{2\theta} = K R_\theta^2 K^{-1} \tag{1.24}$$

Since for any matrix A, $A^{-1}A = I$

$$H_{2\theta} = K R_\theta K^{-1} K R_\theta K^{-1} \tag{1.25}$$

Using eq (1.20) (1.26)

$$H_{2\theta} = H_\theta H_\theta \tag{1.27}$$

$$H_{2\theta} = H_\theta^2 \tag{1.28}$$

Hence, H_θ^2 is the homography corresponding to a rotation of 2θ

Q1.4.3 (5 points): Limitations of the planar homography Why is the planar homography not completely sufficient to map any arbitrary scene image to another viewpoint? State your answer concisely in one or two sentences.

Q1.4.3

For planar homography to work, we assume that the homography undergoes pure rotation or the depth of scenes is negligible for the non-pure rotation case such that we can assume Z = 0. Hence, if these conditions break, then the calculated homography will not be sufficient.

Q1.4.4 (5 points): Behavior of lines under perspective projections We stated in class that perspective projection preserves lines (a line in 3D is projected to a line in 2D). Verify algebraically that this is the case, i.e., verify that the projection $\mathbf{P}$ in $\mathbf{x} = \mathbf{PX}$ preserves lines.

Q1.4.4

Let **P** be the projection of 3D point **X** in 2D. The $x = PX$. Let there be 3 points $x_1 = PX_1$, $x_2 = PX_2$, and $x_3 = PX_3$. Let points X1, X2, and X3 lie on the same line then assuming X_1 to be the anchor point and $X_2 - X_1$ to be the direction vector, we can write $X_3 = X_1 + s(X_2 - X_1)$. Here s is some scalar scale factor. Now multiply both sides of the 3D line equation by projection matrix **P**, and we'd get

$$PX_3 = P(X_1 + s(X_2 - X_1)) \tag{1.29}$$

By matrix distributive property (1.30)

$$PX_3 = PX_1 + sP(X_2 - X_1) \tag{1.31}$$

$$PX_3 = PX_1 + s(PX_2 - PX_1) \tag{1.32}$$

Since $x_3 = PX_3$, $x_2 = PX_2$, and $x_1 = PX_1$ we can see that it translates to an equation of a line in 2D as shown below

$$x_3 = x_1 + s(x_2 - x_1)$$

2 Computing Planar Homographies

2.1 Feature Detection and Matching

Before finding the homography between an image pair, we need to find corresponding point pairs between two images. But how do we get these points? One way is to select them manually, which is tedious and inefficient. The CV way is to find interest points in the image pair and automatically match them. **In the interest of being able to do cool stuff, we will not reimplement a feature detector or descriptor here by yourself, but use python modules (both of them are provided in helper.py)**

The purpose of an interest point detector (e.g. Harris, SIFT, SURF, etc.) is to find particular salient points in the images around which we extract feature descriptors (e.g. MOPS, etc.). These descriptors try to summarize the content of the image around the feature points in as succint yet descriptive manner possible (there is often a trade-off between representational and computational complexity for many computer vision tasks; you can have a very high dimensional feature descriptor that would ensure that you get good matches, but computing it could be prohibitively expensive).

Matching, then, is a task of trying to find a descriptor in the list of descriptors obtained after computing them on a new image that best matches the current descriptor. This could be something as simple as the Euclidean distance between the two descriptors, or something more complicated, depending on how the descriptor is composed. For the purpose of this exercise, we shall use the widely used FAST detector in concert with the BRIEF descriptor.

Q2.1.1 (5 points): FAST Detector How is the FAST detector different from the Harris corner detector that you've seen in the lectures? Can you comment on its computational performance compared to the Harris corner detector? Reference links: Original Paper, OpenCV Tutorial

Q2.1.1

FAST corner detector: We test a circular region (usually 16 pixels) around a potential corner center in this method. The algorithm designates the potential corner center as a corner if for a selected threshold (t) the number of contiguous pixels that are brighter than the center (t + center_pixelintensity) or lighter than the center (center_pixelintensity - t) meet a minimum requirement.
Harris corner detector: In this method, the designation of a pixel to be a corner is judged based on directional gradients (horizontal and vertical) in conjunction with the usage of a smoothing gaussian filter. The gaussian filter helps in removing noise but retaining prominent edges which helps in better detection of corners.
The gradients computed in the harris corner detector are localized and the usage of a gaussian filter helps in much more accurate detection as compared to the FAST corner detector which uses a global threshold value to measure the potential corner intensity difference with its neighbors. This causes the FAST corner detector to generate more false positives than the Harris corner detector method.
Complexity wise the harris corner detector would require more computational resources to run the gaussian blur and calculate image gradients where as FAST would be much faster since it's a simple algorithm based on intensity difference.

Q2.1.2 (5 points): BRIEF Descriptor How is the BRIEF descriptor different from the filterbanks you've seen in the lectures? Could you use any one of those filter banks as a descriptor?

Q2.1.2

Filter banks can be thought of as feature detectors and the primary difference between feature detectors and descriptors is the former is used to find the key features from a given image, whereas the latter is used to quantitatively describe the extracted features.
BRIEF descriptors and filter bank based descriptors differ in the way they are used. BRIEF descriptor captures distinctive local features of an image patch as opposed to a filter bank based descriptor which is great at capturing global features of the whole image. Algorithm-wise, brief relies on a relatively small number of intensity difference tests to represent an image patch as a binary string whereas, for the global representation of the image, filter responses from all filters in the filter bank are stored to construct the final global descriptor. As said, a filter bank is good for capturing global representation but there is spatial information loss, and constructing an image patch-level filter bank descriptor would be computationally expensive

Q2.1.3 (5 points): Matching Methods The BRIEF descriptor belongs to a category called binary descriptors. In such descriptors, the image region corresponding to the detected feature point is represented as a binary string of 1s and 0s. A commonly used metric used for such descriptors is called the Hamming distance. Please search online to learn about Hamming distance and Nearest Neighbor, and describe how they can be used to match interest points with BRIEF descriptors. What benefits does the Hamming distance have over a more conventional Euclidean distance measure in our setting?

Q2.1.3

Hamming Distance: It's a metric for comparing two binary strings (let's say a, b) of equal length. Hamming distance is the number of bit positions in which the bits at that position are different for a, b. It can be represented as follows:

$$D(a,b) = \sum_{i}^{N} I(a_i \neq b_i)$$

Here N is the number of bits in a, b. a_i and b_i is the ith bit position value and I is the indicator function
Nearest Neighbor: Nearest neighbor applies a very simple framework to find the nearest match for a test point given a set of available cases. The metric to compare the similarity of a test point with the available cases can be carried out using multiple ways e.g. cosine similarity, euclidean distance, manhattan distance, etc. The test point is then assigned the majority value (in case of classification) or a mean value (in case of regression) of the found nearest neighbors (k). The k here needs to be found empirically.
Euclidean vs Hamming Distance: Hamming distance is efficient to compute as it can be computed using bitwise XOR and then counting the number ones (which can be calculated using efficient bitwise masks). Whereas euclidean would involve **N** multiplications, subtractions, and addition. Since the BRIEF descriptors give binary strings, we do not need to perform euclidean distance as it's going only add extra computation, hence hamming distance is used in BRIEF for faster computation

Q2.1.4 (10 points): Feature Matching

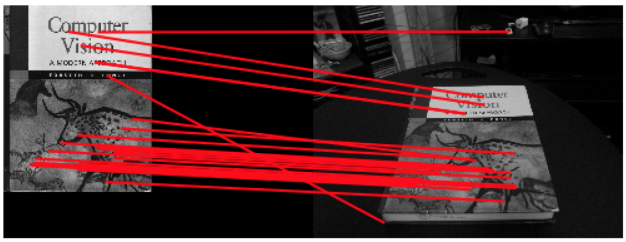


Figure 2.1: A few matched FAST feature points with the BRIEF descriptor.

Please implement a function in `matchPics.py`:

```
matches, locs1, locs2 = matchPics(I1, I2, opts)
```

where `I1` and `I2` are the images you want to match. `opts` stores two parameters. `sigma` is threshold for corner detection using FAST feature detector, and `ratio` is the ratio for BRIEF feature descriptor. `locs1` and `locs2` are $N \times 2$ matrices containing the x and y coordinates of the matched point pairs. `matches` is a $p \times 2$ matrix where the first column is indices into descriptor of features in `I1` and similarly second column contains indices related to `I2`.

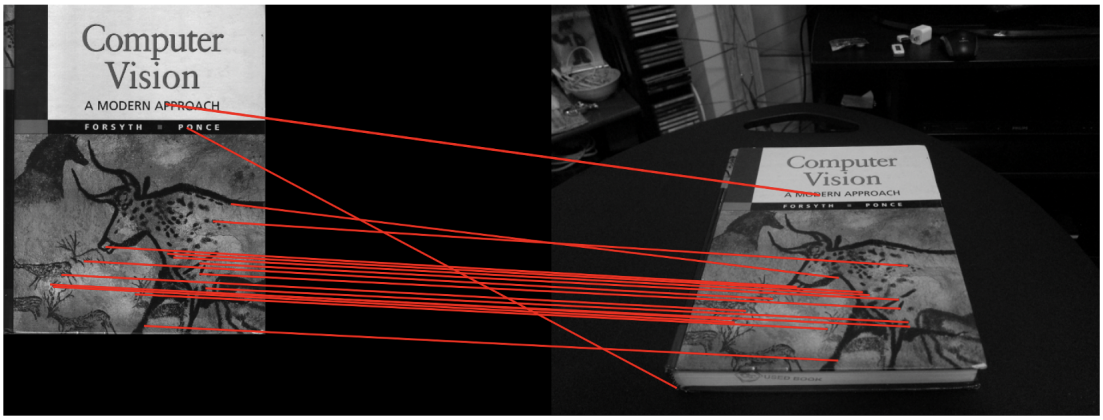
Use the provided helper function `corner_detection()` to compute the features, then build descriptors using `computeBrief()`, and finally compare them using `briefMatch()`. Use `plotMatches()` to visualize your matched points and include the result image in the first box below. An example is shown in Fig. 2.1.

The number of matches between the 2 images varies based on the values of `sigma` in `corner_detection()` and `ratio` in `briefMatch()`. You can vary these to get the best results. The example shown in Fig. 2.1 is with `sigma`$= 0.15$ and `ratio`$= 0.7$. We provide you with the following helper functions in `helper.py`:

```
locs = corner_detection(img, sigma)
desc, locs = computeBrief(img, locs)
matches = briefMatch(desc1, desc2, ratio)
plotMatches(im1, im2, matches, locs1, locs2)
```

`locs` is an $N \times 2$ matrix in which each row represents the location (x, y) of a feature point. Please note that the number of valid output feature points could be less than the number of input feature points. `desc` is the corresponding matrix of BRIEF descriptors for the interest points. **Include the code you wrote for this part in the second box below:**

Q2.1.4 (a) result image



Q2.1.4 (b) codes

```python
import numpy as np
import cv2
# import skimage.color
from helper import briefMatch
from helper import computeBrief
from helper import corner_detection
from helper import plotMatches
from opts import get_opts
# Q2.1.4
def matchPics(I1, I2, opts):
    """
    Match features across images
    Input
    -----
    I1, I2: Source images
    opts: Command line args
    Returns
    -------
    matches: List of indices of matched features across I1, I2 [p x 2]
    locs1, locs2: Pixel coordinates of matches [N x 2]
    """
    ratio = opts.ratio  # 'ratio for BRIEF feature descriptor'
    sigma = opts.sigma  # 'threshold for corner detection using FAST feature
    ↪  detector'
    # TODO: Convert Images to GrayScale
    if len(I1.shape) > 2 and I1.shape[2] == 3:
        I1 = cv2.cvtColor(I1, cv2.COLOR_BGR2GRAY)
    if len(I2.shape) > 2 and I2.shape[2] == 3:
        I2 = cv2.cvtColor(I2, cv2.COLOR_BGR2GRAY)
    # TODO: Detect Features in Both Images
    fastI1 = corner_detection(I1, sigma)
    fastI2 = corner_detection(I2, sigma)
    # TODO: Obtain descriptors for the computed feature locations
    descI1, locs1 = computeBrief(I1, fastI1)
    descI2, locs2 = computeBrief(I2, fastI2)
    # TODO: Match features using the descriptors
    matches = briefMatch(descI1, descI2, ratio)
    return matches, locs1, locs2
if __name__ == "__main__":
    opts = get_opts()
    image1 = cv2.imread('../data/cv_cover.jpg')
    image2 = cv2.imread('../data/cv_desk.png')
    matches, locs1, locs2 = matchPics(image1, image2, opts)
    plotMatches(image1, image2, matches, locs1, locs2)
```

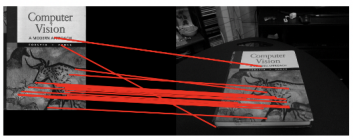
Q2.1.5 (10 points): Feature Matching and Parameter Tuning

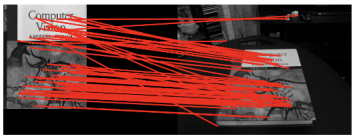
Run the provided starter code `displayMatch.py` to display matched features. There are two tunable parameters, both stored in the `opts` variable, and are loaded from `opts.py`. You can change the values by changing their default fields or by command-line arguments. For example, `python displayMatch.py --sigma 0.15 --ratio 0.7`.

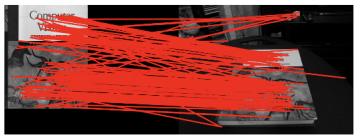
Conduct a small ablation study by running `displayMatch.py` with various `sigma` and `ratio` values. Include the figures displaying the matched features with various parameters in your writeup, and explain the effect of these two parameters respectively.

Q2.1.5

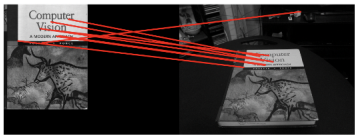
experiment	Sigma	Ratio	#Matches	#locs1	#locs2
1	0.15	0.7	18	945	477
2	0.15	0.8	61	945	477
3	0.15	0.9	128	945	477
4	0.15	1	204	945	477
5	0.3	0.7	8	320	132
6	0.45	0.7	2	213	43
7	0.5	0.7	3	154	30
8	0.6	0.7	0	122	8

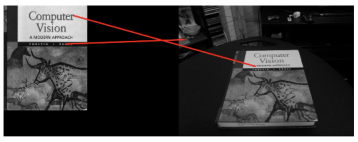

Experiment 1

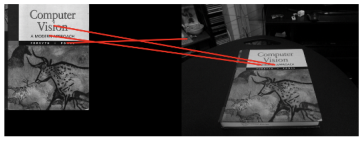

Experiment 2

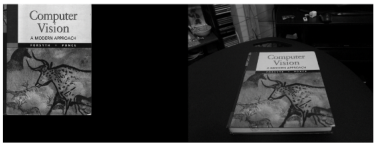

Experiment 3


Experiment 4


Experiment 5


Experiment 6


Experiment 7


Experiment 8

Q2.1.5

The experiments have been planned by keeping one of the two components (sigma and ratio) constant and increasing the other.

As can be observed from the table, as we increase the value of the ratio (i.e. the ratio of BRIEF descriptor) we observe more matches are made. This makes sense because as we increase the max ratio argument in the match descriptors function we allow more potential matches to be retained resulting in retaining more ambiguous matches. But, as we lower the number of matches, we lower the permissible threshold hence disallowing ambiguous matches to retain therefore we see less number of matches.

As we increase sigma, we observe a lesser number of matches which is in accordance with what we might expect. This is because, as we increase sigma we increase the threshold in the corner_fast function which means we want the difference between the potential corner pixel and its neighboring pixels on the locally enclosed boundary to be larger. Hence, this applies a tighter bound to detecting a corner but this threshold might be too strong that it filters out a lot of true corner points.

Based on the above discussion we can conclude that we'd want the sigma to be moderate (not too high to filter true positives and not too low to get false positives) and also the ratio (ratio of the hamming distance between descriptors) to again be moderate (not too low that very low matches are retained and not too high so that ambiguous points are retained)

Q2.1.6 (10 points): BRIEF and Rotations

Let's investigate how BRIEF works with rotations. Write a script `briefRotTest.py` that:

- Takes the `cv_cover.jpg` and matches it to itself rotated from 0 to 360 degrees in increments of 10 degrees. [*Hint*: use `scipy.ndimage.rotate`]
- Stores a histogram of the count of matches for each orientation.
- Plots the histogram using `matplotlib.pyplot.hist` (x axis: rotation, y axis: number of matches).

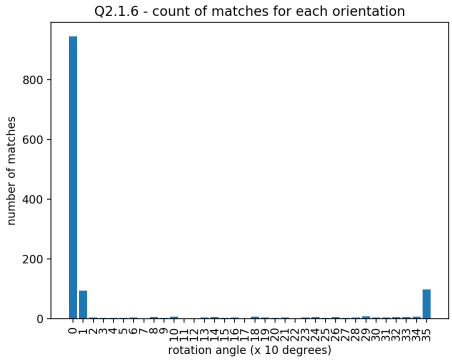
Please include the code you wrote for this part in the box below:

Q2.1.6 (a)

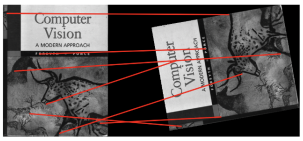
```python
import numpy as np
import cv2
from matchPics import matchPics
from opts import get_opts
import scipy
import matplotlib.pyplot as plt
import skimage
# Q2.1.6
def plot(im1, im2, matches, locs1, locs2):
    fig, ax = plt.subplots(nrows=1, ncols=1)
    plt.axis('off')
    skimage.feature.plot_matches(
        ax, im1, im2, locs1, locs2, matches, matches_color='r',
        ↪  only_matches=True)
    plt.show()
    return
def rotTest(opts):
    # Read the image and convert to grayscale, if necessary
    img = cv2.imread("../data/cv_cover.jpg")
    if img.shape[2] == 3:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    x = []
    y = []
    for i in range(36):
        angle = i * 10
        # Rotate Image
        rotated_img = scipy.ndimage.rotate(img, angle)
        # Compute features, descriptors and Match features
        matches, locs1, locs2 = matchPics(img, rotated_img, opts)
        # visualize
        if i in [1, 10, 20]:
            plot(img, rotated_img, matches, locs1, locs2)
        # Update histogram
        x.append(i)
        y.append(len(matches))
        print(f"Iteration: {i}")
    # Display histogram
    plt.figure()
    plt.bar(x, y)
    plt.xlabel("rotation angle (x 10 degrees)")
    plt.ylabel("number of matches")
    plt.title("Q2.1.6 - count of matches for each orientation")
    plt.show()
if __name__ == "__main__":
    opts = get_opts()
    rotTest(opts)
```

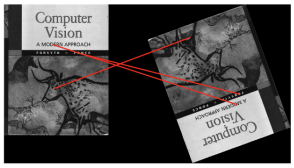
Visualize the histogram and the feature matching result at three different orientations and include them in your write-up. Explain why you think the BRIEF descriptor behaves this way.

Q2.1.6 (b)




10 degrees


100 degrees


200 degrees

As can be seen from the above histogram we get the maximum matches at 0 degrees and as we rotate the image, we observe there is a drastic decrease in the number of matches. This is probably because of the fact that BRIEF descriptors depend upon the fixed set of sampled points and their orientation. Hence changing orientation will lead to poor performance

Q2.1.7 (Extra Credit): Improving Performance The extra credit opportunities described below are optional and provide an avenue to explore computer vision and improve the performance of the techniques developed above.

Q2.1.7.1 (Extra Credit - 5 points): As we have seen, BRIEF is not rotation invariant. Design a simple fix to solve this problem using the tools you have developed so far (think back to edge detection and/or Harris corner's covariance matrix). You are not allowed to use any additional OpenCV or Scikit-Image functions. **Include the code you wrote for this part in the box below** and explain your design decisions and how you selected any parameters that you use. Demonstrate the effectiveness of your algorithm on image pairs

related by large rotation.

Q2.1.7.1

Q2.1.7.2 (Extra Credit - 5 points): This implementation of BRIEF has some scale invariance, but there are limits. What happens when you match a picture to the same picture at half the size? Look to section 3 of [Lowe2004], for a technique that will make your detector more robust to changes in scale. Implement it and demonstrate it in action with several test images. **Include the code you wrote for this part in the box below.** You are not allowed to call any additional OpenCV or Scikit-Image functions. You may simply rescale some of the test images we have given you.

Q2.1.7.2

2.2 Homography Computation

Q2.2.1 (15 points): Computing the Homography

Write a function `computeH` in `planarH.py` that estimates the planar homography from a set of matched point pairs.

```
H2to1 = computeH(x1, x2)
```

`x1` and `x2` are $N \times 2$ matrices containing the coordinates (x, y) of point pairs between the two images. `H2to1` should be a 3×3 matrix for the best homography from image 2 to image 1 in the least-squares sense. The `numpy.linalg` function `eig()` or `svd()` will be useful to get the eigenvectors (see Section 1 of this handout for details). **Include the code you wrote for this part in the box below.**

Q2.2.1

```python
def computeH(x1, x2):
    #Q2.2.1
    #Compute the homography between two sets of points
    assert x1.shape == x2.shape
    total_points = x1.shape[0]
    A = np.zeros((2 * total_points, 9))
    for i in range(0, total_points):
        X1, Y1 = x1[i][0], x1[i][1]
        X2, Y2 = x2[i][0], x2[i][1]
        A[2 * i] = np.array([
            -X2, -Y2, -1, 0, 0, 0, X1 * X2, X1 * Y2, X1
        ])
        A[2 * i + 1] = np.array([
            0, 0, 0, -X2, -Y2, -1, Y1 * X2, Y1 * Y2, Y1
        ])
    u, s, vt = np.linalg.svd(A)
    H2to1 = vt[-1].reshape((3, 3))
    return H2to1
```

Q2.2.2 (10 points): Homography Normalization

Normalization improves numerical stability of the solution and you should always normalize your coordinate data. Normalization has two steps:

1. Translate the mean of the points to the origin.
2. Scale the points so that the largest distance to the origin is $\sqrt{2}$

This is a linear transformation and can be written as follows:

$$\tilde{\mathbf{x}}_1 = \mathbf{T_1}\mathbf{x_1}$$
$$\tilde{\mathbf{x}}_2 = \mathbf{T_2}\mathbf{x_2}$$

where $\tilde{\mathbf{x}}_1$ and $\tilde{\mathbf{x}}_2$ are the normalized homogeneous coordinates of $\mathbf{x_1}$ and $\mathbf{x_2}$. $\mathbf{T_1}$ and $\mathbf{T_2}$ are 3×3 matrices. The homography $\mathbf{H}$ from $\tilde{\mathbf{x}}_2$ to $\tilde{\mathbf{x}}_1$ computed by `computeH` satisfies

$$\tilde{\mathbf{x}}_1 = \mathbf{H}\tilde{\mathbf{x}}_2$$

By substituting $\tilde{\mathbf{x}}_1$ and $\tilde{\mathbf{x}}_2$ with $\mathbf{T_1}\mathbf{x_1}$ and $\mathbf{T_2}\mathbf{x_2}$, we have:

$$\mathbf{T_1}\mathbf{x_1} = \mathbf{H}\mathbf{T_2}\mathbf{x_2}$$
$$\mathbf{x_1} = \mathbf{T_1}^{-1}\mathbf{H}\mathbf{T_2}\mathbf{x_2}$$

Implement the function `computeH_norm`:

```
H2to1 = computeH_norm(x1, x2)
```

This function should normalize the coordinates in `x1` and `x2` and call `computeH(x1, x2)` as described above. **Include the code you wrote for this part in the box below.**

Q2.2.2

```python
def computeH_norm(x1, x2):
    #Q2.2.2
    #Compute the centroid of the points
    mewx1 = np.mean(x1, axis=0)
    mewx2 = np.mean(x2, axis=0)
    #Shift the origin of the points to the centroid
    x1shifted = x1 - mewx1
    x2shifted = x2 - mewx2
    #Normalize the points so that the largest distance from the origin is equal
    ↪  to sqrt(2)
    max_euclidean_x1 = np.max(np.linalg.norm(x1shifted, ord=2, axis=1))
    max_euclidean_x2 = np.max(np.linalg.norm(x2shifted, ord=2, axis=1))
    scale1 = np.sqrt(2) / max_euclidean_x1
    scale2 = np.sqrt(2) / max_euclidean_x2
    x1shifted = scale1 * x1shifted
    x2shifted = scale2 * x2shifted
    #Similarity transform 1
    T1 = np.array([
        [scale1, 0, -scale1 * mewx1[0]],
        [0, scale1, -scale1 * mewx1[1]],
        [0, 0, 1]
    ])
    #Similarity transform 2
    T2 = np.array([
        [scale2, 0, -scale2 * mewx2[0]],
        [0, scale2, -scale2 * mewx2[1]],
        [0, 0, 1]
    ])
    #Compute homography
    H = computeH(x1shifted, x2shifted)
    #Denormalization
    H2to1 = np.linalg.inv(T1) @ (H @ T2)
    return H2to1
```

Q2.2.3 (25 points): Implement RANSAC

The RANSAC algorithm can generally fit any model to noisy data. You will implement it for (planar) homographies between images. Remember that 4 point-pairs are required at a minimum to compute a ho-

mography.

Write a function:

```
bestH2to1, inliers = computeH_ransac(locs1, locs2, opts)
```

where `locs1` and `locs2` are $N \times 2$ matrices containing the matched points. `opts` stores two RANSAC parameters. `max_iters` is the number of iterations to run RANSAC for, and `inlier_tol` is the tolerance value for considering a point to be an inlier. `bestH2to1` should be the homography $\mathbf{H}$ with most inliers found during RANSAC. $\mathbf{H}$ will be a homography such that if $\mathbf{x_2}$ is a point in `locs2` and $\mathbf{x_1}$ is a corresponding point in `locs1`, then $\mathbf{x_1} \equiv \mathbf{H}\mathbf{x_2}$. `inliers` is a vector of length N with a 1 at those matches that are part of the consensus set, and 0 elsewhere. Use `computeH_norm` to compute the homography. **Include the code you wrote for this part in the box below.**

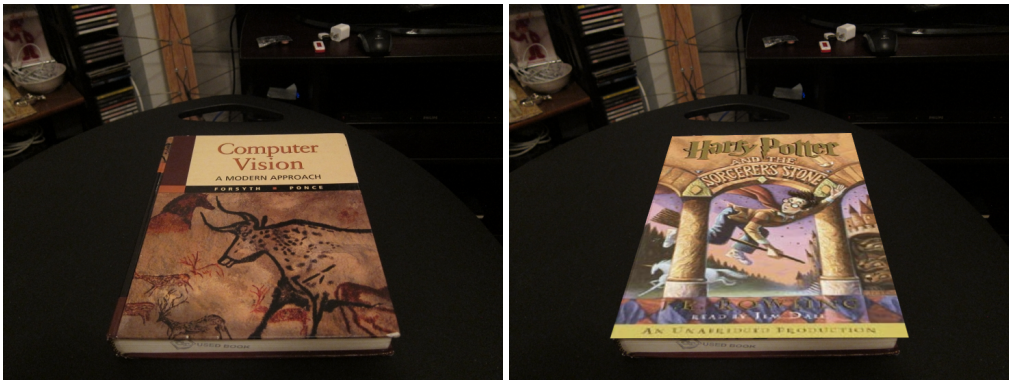


Figure 2.2: An ordinary textbook (left). Harry-Potterized book (right)

Q2.2.3

```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 inliers
    assert locs1.shape[0] == locs2.shape[0]
    total_matched_points = locs1.shape[0]
    bestH2to1 = np.zeros((3, 3))
    locs1 = np.hstack((locs1, np.ones((total_matched_points, 1))))
    locs2 = np.hstack((locs2, np.ones((total_matched_points, 1))))
    inliers = 0
    all_indexes = range(0, total_matched_points)
    for i in range(max_iters):
        selected_indexes = random.sample(all_indexes, 4)
        leftImage = locs1[selected_indexes, :]
        rightImage = locs2[selected_indexes, :]
        H = computeH_norm(leftImage, rightImage)
        locs1hat = (H @ locs2.T).T
        locs1hat = locs1hat / (locs1hat[:, 2][:, np.newaxis] + 1e-10)
        t = np.linalg.norm(locs1hat - locs1, ord=2, axis=1)
        tmp_inliers = np.sum(t[t < inlier_tol])
        if tmp_inliers > inliers:
            bestH2to1 = np.copy(H)
            inliers = tmp_inliers
    return bestH2to1, inliers
```

Q2.2.4 (10 points): Automated Homography Estimation and Warping

Write a function `warpImage()` in `HarryPotterize.py` that

1. Reads `cv_cover.jpg`, `cv_desk.png`, and `hp_cover.jpg`.
2. Computes a homography automatically using `matchPics` and `computeH_ransac`.
3. Uses the computed homography to warp `hp_cover.jpg` to the dimensions of the `cv_desk.png` image using the OpenCV function `cv2.warpPerspective` function.
4. At this point you should notice that although the image is being warped to the correct location, it is not filling up the same space as the book. Why do you think this is happening? How would you modify `hp_cover.jpg` to fix this issue?
5. Implement the function: `composite_img = compositeH(H2to1, template, img)` to now compose this warped image with the desk image as in in Figure 2.2

6. **Include the code you wrote for this part and the result image in the box below**

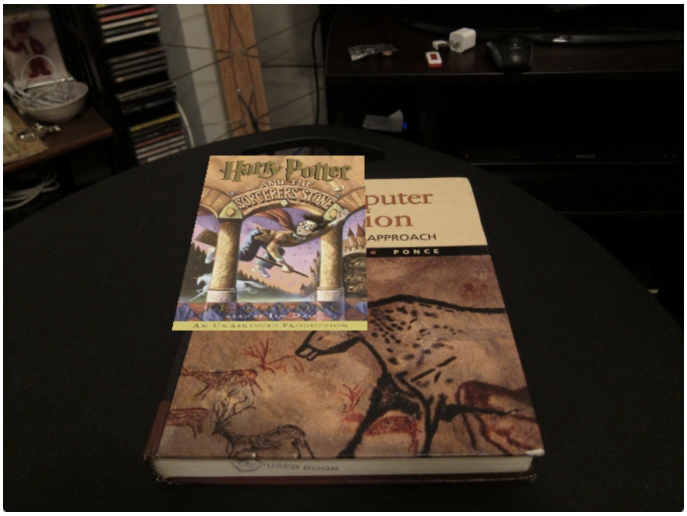
Q2.2.4

```python
def compositeH(H2to1, template, img):
    #Create a composite image after warping the template image on top
    #of the image using the homography
    #Note that the homography we compute is from the image to the template;
    #x_xtemplate = H2to1*x_photo
    #For warping the template to the image, we need to invert it.
    #Create mask of same size as template
    #Warp mask by appropriate homography
    #Warp template by appropriate homography
    #Use mask to combine the warped template and the image
    Hinv = np.linalg.inv(H2to1)
    composite_image = np.zeros(img.shape)
    composite_image = np.copy(img)
    mask = np.ones(template.shape, dtype=np.uint8) * 255
    warped_mask = cv2.warpPerspective(
        mask, Hinv, (img.shape[1], img.shape[0])
    )
    warp_template = cv2.warpPerspective(
        template, Hinv, (img.shape[1], img.shape[0])
    )
    indexes = np.where(warped_mask == 255)
    composite_image[indexes] = warp_template[indexes]
    return composite_image

def warpImage(opts):
    print(f"opts max_iters: {opts.max_iters}, inlier_tol: {opts.inlier_tol}")
    image1 = cv2.imread('../data/cv_cover.jpg')
    image2 = cv2.imread('../data/cv_desk.png')
    image3 = cv2.imread("../data/hp_cover.jpg")
    image3 = cv2.resize(image3, (image1.shape[1], image1.shape[0]))
    matches, locs1, locs2 = matchPics(image1, image2, opts)
    locs1 = locs1[matches[:, 0], :]
    locs2 = locs2[matches[:, 1], :]
    locs1[:, [0, 1]] = locs1[:, [1, 0]]
    locs2[:, [0, 1]] = locs2[:, [1, 0]]
    H, _ = computeH_ransac(locs1, locs2, opts)
    composite_img = compositeH(H, image3, image2)
    cv2.imshow("warped", composite_img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

Q2.2.4

HP cover has smaller dimensions when compared with the CV cover hence once a homography is found between the CV cover and the CV desk and when we apply this to the HP cover the warped HP cover would only cover a small part of the CV desk (i.e. composite image will be formed from top left on the CV desk) because only a subset of coordinates are transformed



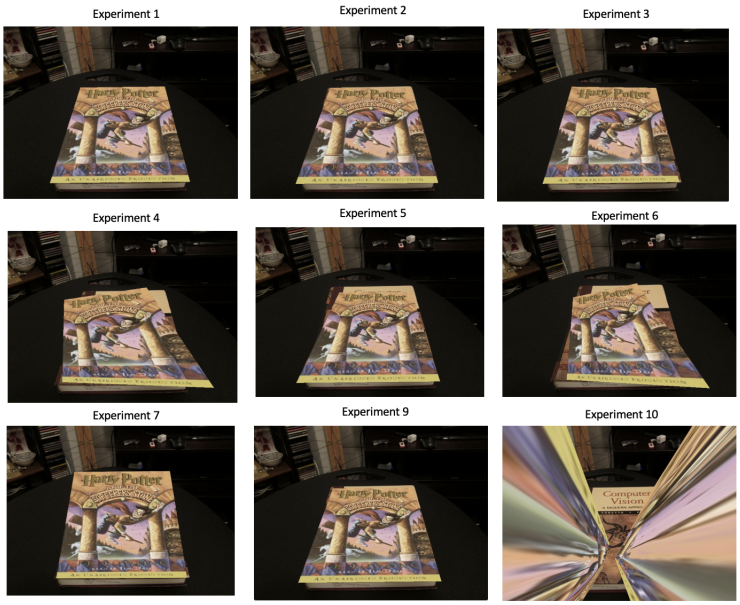
In order for the warped image to cover the whole cover in the CV desk, we'd require the HP cover to be of the same size as the CV cover so that the same number of coordinates are projected.

Q2.2.5 (10 points): RANSAC Parameter Tuning

Just like how we tune parameters for feature matching, there are two tunable parameters in RANSAC as well. You can change the values by changing their default fields or by command-line arguments. For example, `python HarryPotterize.py --max_iters 500 --inlier_tol 2.0`. Conduct a small ablation study by running `HarryPotterize.py` with various `max_iters` and `inlier_tol` values. Plot the result images in the box below and explain the effect of these two parameters respectively.

Q2.2.5

Experiment	Max iterations	Inlier Tolerance
1	500	2.0
2	500	1.8
3	500	2.2
4	500	10
5	50	2.0
6	50	25.0
7	600	1.2
8	500	0 (Will throw linalg error due to singular matrix)
9	50	0.5
10	500	100



Setup: The experiments were not performed by using a fixed random seed. It's been observed due to random sampling we might get different results for the same hyperparameter setting. The images shown below are taken after performing 10 trials for each setup and seeing how it performed each time and then selecting the image which was seen most of the times

Observations:

- Inlier tolerance is important to accurately estimate the performance of a Homography matrix. We can see from experiments 4, 6, and 10. When the tolerance is increased we are probably taking into account more ambiguous points which are causing RANSAC to select the wrong H matrix. A lower value of around 1 to 2 seems to be a good choice. Even when the iterations were low in experiment 9 a strict tolerance value allowed us to get a good warp
- As max iterations are considered, intuitively it makes to increase the number of iterations so that we get to sample more match points and see which generates the best H. But, according to experiments though the tolerance seems to have more importance than the max iteration. As can be seen from experiment 10, even with a large number of iterations, a higher tolerance would result in a highly distorted warp
- Finally Setup used in experiment 6 gave the best warp in general when the trail was performed multiple times

3 Creating your Augmented Reality application

3.1 Incorporating video (20 points)

Now with the code you have, you're able to create you own Augmented Reality application. What you're going to do is HarryPoterize the video `ar_source.mov` onto the video `book.mov`. More specifically, you're going to track the computer vision text book in each frame of `book.mov`, and overlay each frame of `ar_source.mov` onto the book in `book.mov`. Please write a script `ar.py` to implement this AR application and save your result video as `ar.avi` in the `result/` directory. You may use the function `loadVid()` that we provide to load the videos. Your result should be similar to the LifePrint project. You'll be given full credits if you can put the video together correctly. See Figure 3.1 for an example frame of what the final video should look like. Hint for saving videos with OpenCV: Make sure the size of the images that you add to the VideoWriter match the size of the image used to instantiate the VideoWriter. Also, if you're using a Mac, you may need to specify a different type of codec than MPV4. AVI usually works.

Figure 3.1: Rendering video on a moving target

Note that the book and the videos we have provided have very different aspect ratios (the ratio of the image width to the image height). You must crop each frame to fit onto the book cover. You must crop that image such that only the central region of the image is used in the final output. See Figure 3.2 for an example.
The two videos have different lengths, you can use the shorter length for your video. Also, the video

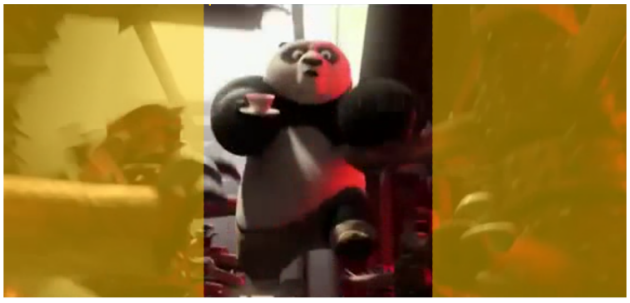


Figure 3.2: Rendering video on a moving target

`book.mov` only has translation of objects. If you want to account for rotation of objects, scaling, etc, you

would have to pick a better feature point representation (like ORB).

Finally, note that this is a very time-intensive job and may take many hours on a single core (parallel processing on 8 cores takes around 30 minutes). Debug before running your full script (e.g. by saving a few early AR frames and verifying that they look correct).

In your writeup, include three screenshots of your ar.avi at three distinct timestamps (e.g. when the overlay is near the center, left, and right of the video frame). See Figure 3.1 as an example of where the overlay is in the center of the video frame. Also include the code you wrote for this part in the box below

If the video is too large, please include a Google Drive link to your video in the writeup instead and ensure the shared link gives the TA's viewing permission.

Q3.1

```python
#Import necessary functions
import numpy as np
import cv2
from helper import loadVid
from opts import get_opts
from matchPics import matchPics
from planarH import computeH_ransac, compositeH
import multiprocessing

def ar_one_frame(i, frame, template, src_vidw, im1, opts):
    template = template[44:-44, src_vidw//2 - im1.shape[1]//2: src_vidw//2 +
    ↪  im1.shape[1]//2]
    template = cv2.resize(template, im1.shape[:2][::-1])

    matches, locs1, locs2 = matchPics(im1, frame, opts)
    locs1 = locs1[matches[:, 0], :]
    locs2 = locs2[matches[:, 1], :]

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

    H, _ = computeH_ransac(locs1, locs2, opts)
    frame = compositeH(H, template, frame)
    frame = frame.astype(np.uint8)
    return i, frame

def ar(opts):
    im1 = cv2.imread("../data/cv_cover.jpg")
    src_vid = loadVid("../data/ar_source.mov")
    dest_vid = loadVid("../data/book.mov")
    # saved below to inspect how to remove black borders
    # cv2.imwrite("./results/arframe0.png", src_vid[0])
    diff = dest_vid.shape[0] - src_vid.shape[0]
    extra = np.zeros((diff, src_vid.shape[1], src_vid.shape[2],
    ↪  src_vid.shape[3]))
    for i in range(diff):
        extra[i, :, :, :] = src_vid[i, :, :, :]
    src_vid = np.concatenate((extra, src_vid), axis=0)
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    writer = cv2.VideoWriter(
        "./results/ar.mp4",
        fourcc,
        25,
        (dest_vid[0].shape[1], dest_vid[0].shape[0])
    )
    _, src_vidw, _ = src_vid[0].shape
```

Q3.1

```
mp_inputs = [(
        i,
        dest_vid[i],
        src_vid[i],
        src_vidw,
        im1,
        opts
    ) for i in range(dest_vid.shape[0])]
    ar_frames = np.zeros(dest_vid.shape, dtype=np.uint8)
    def handle_result(result):
        ar_frames[result[0]] = result[1]
    n_cpus = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(n_cpus)
    for i in range(len(mp_inputs)):
        pool.apply_async(ar_one_frame, args=mp_inputs[i], callback=handle_result)
    pool.close()
    pool.join()
    for frame in ar_frames:
        writer.write(frame)
    writer.release()
#Write script for Q3.1
if __name__=="__main__":
    opts = get_opts()
    ar(opts)
```

Link: Video Link

3.2 Make Your AR Real Time (Extra Credit - 15 points)

Write a script `ar_ec.py` that implements the AR program described in Q3.1 in real time. As an output of the script, you should process the videos frame by frame and have the combined frames played in real time. You don't need to save the result video for this question. The extra credits will be given to fast programs measured by FPS (frames per second). More specifically, we give 5 points to programs that run faster than 10 FPS, 10 points to programs running faster than 20 FPS and 15 points to programs running faster than 30 FPS. **You are not allowed to use any additional OpenCV or Scikit-Image functions. Make sure to note the achieved fps in your write-up in addition to all the steps taken to achieve real-time performance. Also include the code you wrote for this part in the box below.**

Q3.2

4 Create a Simple Panorama (10 points)

Take two pictures with your own camera, separated by a pure rotation as best as possible, and then construct a panorama with `panorama.py`. Be sure that objects in the images are far enough away so that there are no parallax effects. You can use python module `cpselect` to select matching points on each image or some automatic method. We have provided two images for you to get started (`data/pano_left.jpg` and `data/pano_right.jpg`). Please use your own images when submitting this project. Include your code, the original images, the final panorama result image in the box below. See Figure 4.1 below for example.

Figure 4.1: Original Image 1 (top left). Original Image 2 (top right). Panorama (bottom).

Q4

```python
# Import necessary functions
import numpy as np
import cv2
from cpselect.cpselect import cpselect
from opts import get_opts
from planarH import computeH_ransac


# Q4
def parse(objs):
    x1 = np.zeros((len(objs), 2))
    x2 = np.zeros((len(objs), 2))

    for i, obj in enumerate(objs):
        x1[i, :] = np.array([
            obj["img1_x"], obj["img1_y"]
        ])
        x2[i, :] = np.array([
            obj["img2_x"], obj["img2_y"]
        ])

    return x1, x2


def warp(im1_path, im2_path, opts):
    im1 = cv2.imread(im1_path)
    im2 = cv2.imread(im2_path)
    # objs = cpselect(im1_path, im2_path)
    # print(objs)

    # cmu
    objs = [{'point_id': 1, 'img1_x': 1622.6784279977046, 'img1_y':
    ↪  1638.734816818293, 'img2_x': 374.8462830915496, 'img2_y':
    ↪  1485.4686910908936}, {'point_id': 2, 'img1_x': 2956.9952872715344,
    ↪  'img1_y': 1386.2964920908116, 'img2_x': 1871.4449225473318, 'img2_y':
    ↪  1431.374764363576}, {'point_id': 3, 'img1_x': 1541.5375379067286,
    ↪  'img1_y': 1007.6390049995898, 'img2_x': 356.8149741824436, 'img2_y':
    ↪  746.1850258175555}, {'point_id': 4, 'img1_x': 1992.3202606343739,
    ↪  'img1_y': 2188.68973854602, 'img2_x': 753.5037701827714, 'img2_y':
    ↪  2152.6271207278087}]
```

Q4

```
# craig
# objs = [{'point_id': 1, 'img1_x': 2794.713507089582, 'img1_y':
↪  998.6233505450368, 'img2_x': 1646.0535611835094, 'img2_y':
↪  989.6076960904838}, {'point_id': 2, 'img1_x': 1821.0228259978685,
↪  'img1_y': 2188.68973854602, 'img2_x': 501.0654454552896, 'img2_y':
↪  2179.6740840914676}, {'point_id': 3, 'img1_x': 2695.5413080895, 'img1_y':
↪  2215.736701909679, 'img2_x': 1483.7717810015565, 'img2_y':
↪  2170.6584296369147}, {'point_id': 4, 'img1_x': 3308.6058109990977,
↪  'img1_y': 1918.2201049094333, 'img2_x': 2033.7267027292837, 'img2_y':
↪  1873.14183263666888}]
# for pano right left
# objs = [
#     {
#         'point_id': 1,
#         'img1_x': 1424.8665847061711,
#         'img1_y': 260.36406851897414,
#         'img2_x': 1081.9980985165148,
#         'img2_y': 309.6233505450375
#     },
#     {
#         'point_id': 2,
#         'img1_x': 791.0638226374886,
#         'img1_y': 772.6606015900337,
#         'img2_x': 520.4422834193915,
#         'img2_y': 792.3643144004591
#     },
#     {
#         'point_id': 3,
#         'img1_x': 942.1256208507498,
#         'img1_y': 559.2037128104255,
#         'img2_x': 668.2201294975821,
#         'img2_y': 582.1913777559218
#     },
#     {
#         'point_id': 4,
#         'img1_x': 1339.4838291943279,
#         'img1_y': 641.3025161871979,
#         'img2_x': 1006.4671994098844,
#         'img2_y': 644.5864683222687
#     }
# ]
```

Q4

```
x1, x2 = parse(objs)
    H, _ = computeH_ransac(x1, x2, opts)
    canvas = np.ones((im2.shape[0] * 2, im2.shape[1] * 2, 3), dtype=np.uint8) *
    ↪  255
    for i in range(im1.shape[0]):
        for j in range(im1.shape[1]):
            canvas[i, j] = im1[i, j]
    mask = np.ones(im2.shape, dtype=np.uint8) * 255
    warped_mask = cv2.warpPerspective(
        mask, H, canvas.shape[:2]
    )
    warped_im2 = cv2.warpPerspective(
        im2, H, canvas.shape[:2]
    )
    indexes = np.where(warped_mask == 255)
    canvas[indexes] = warped_im2[indexes]
    cv2.imshow("warped", canvas)
    cv2.waitKey(0)
    cv2.destroyAllWindows()


if __name__=="__main__":
    opts = get_opts()
    # warp("../data/pano_left.jpg", "../data/pano_right.jpg", opts)
    # warp("../data/craig_left.jpg", "../data/craig_right.jpg", opts)
    warp("../data/cmu_left.jpg", "../data/cmu_right.jpg", opts)
```

(a) left

(b) right

(c) panaroma

Figure 4.2: CMU Panaroma

(a) left

(b) right

(c) panaroma

Figure 4.3: Craig Panaroma

5 HW3 Distribution Checklist

After unpacking `hw3.zip`, you should have a folder `hw3` containing one folder for the data (`data`), one for your code (`code`) and one for extra credit questions (`ec`). In the `code` folder, where you will primarily work, you will find:

- `ar.py`: script for Section 3
- `briefRotTest.py`: script to test BRIEF with rotations
- `displayMatch.py`: script to display matched features
- `HarryPoterize.py`: script to automate homography and warp between source and target images
- `helper.py`: some helper functions
- `matchPics.py`: script to match features between pair of images
- `opts.py`: some command line arguments
- `planarH.py`: script to estimate planar homography
- `panaroma.py`: script you need to accomplish for Section 4

6 HW3 submission checklist

Submit your write-up and code to Gradescope.

- **Writeup.** The write-up should be a pdf file named <**AndrewId**>**_hw3.pdf**.
- **Code.** The code should be submitted as a zip file named <**AndrewId**>**.zip**. By extracting the zip file, it should have the following files in the structure defined below. (Note: Neglecting to follow the submission structure will incur a huge score penalty!)

  When you submit, remove the folder `data/` if applicable, as well as any large temporary files that we did not ask you to create.

  - <**andrew_id**>**/** # A directory inside .zip file
    - `code/`
      - <!– all of your .py files >
    - `ec/`
      - <!– all of your .py files >
    - `<andrew_id>_hw3.pdf` make sure you upload this pdf file to Gradescope. Please assign the locations of answers to each question on Gradescope.

7 FAQs

Credits: Cherie Ho

1. In `matchPics`, `locs` stands for pixel locations. `locs1` and `locs2` can have different sizes, since matches gives the mapping between the two for corresponding matches. We use `skimage.feature.match` descriptors (API) to calculate the correspondences.
2. Normalized homography - The function `computeH` norm should return the homography `H` between unnormalized coordinates and not the normalized homography `H` norm. As mentioned in the writeup, you can use the following steps as a reference:

$$\mathbf{H}_{\text{norm}} = \texttt{computeH}(\texttt{x1\_normalized}, \texttt{x2\_normalized})$$
$$\mathbf{H} = \mathbf{T}_1^{-1} @ \mathbf{H}_{\text{norm}} @ \mathbf{T}_2$$

3. The `locs` produced by matchPics are in the form of [`row`, `col`], which is (y,x) in coordinates. Therefore, you should first swap the columns returned by matchPics and then feed into Homography estimation.
4. Note that the third output `np.linalg.svd` is `vh` when computing homographies.
5. When debugging homographies, it is helpful to visualize the matches, and checking homographies with the same image. If there is not enough matches, try tuning the parameters.
6. If your images look blue-ish, your red and blue channels may be flipped. This is common when using cv2.imread. You can flip the last channel like so: hp_cover = hp_cover[:, :, [2, 1, 0]] or using another library (skimage.io.imread).
7. For Extra credit Q3.2, we'd like for you to speed up AR so that the processing is happening in real-time. This means we want each "for loop" you run with the ar.py to run in less than 1/30 seconds. You should not need to use multiprocessing. Take a look at your Q3.1 timings. Which step/steps are taking the most time? Can you replace these with faster functions? You are allowed to use functions from other libraries.
8. A common bug is reversing the direction of homography. Make sure it's what you expect!

8 Helpful Concepts

Credits: Jack Good

- **Projection vs. Homography:** A projection, usually written as $\mathbf{P}_{3\times4}$, maps homogeneous 3D world coordinates to the view of a camera in homogeneous 2D coordinates. A planar homography, usually written as $\mathbf{H}_{3\times3}$, under certain assumptions, maps the view of one camera in 2D homogeneous coordinates to the view of another camera in 2D homogeneous coordinates.
- **Deriving homography from projections:** When deriving a homography from projections given assumptions about the world points or the camera orientations, make sure to include the intrinsic matrices of the two cameras, $\mathbf{K}_1$ and $\mathbf{K}_2$, which are 3×3 and invertible, but generally cannot be assumed to be identity or diagonal. Note the rule for inverting matrix products: $(AB)^{-1} = B^{-1}A^{-1}$. When this rule is applied, even when both views are the same camera and $\mathbf{K} = \mathbf{K}_1 = \mathbf{K}_2$, $\mathbf{K}$ is still part of H and does not cancel out.
- **Conditions for planar homography to exist:** For a planar homography to exist between two views, we need either the points in the 3D world lie on a plane (as shown in 1.1 and applied in the Harry Potterize task), or there is pure rotation between the two views (as shown in 1.3 and applied in the Panorama task). We do not require both conditions to hold - only one or the other.

- **Definition of a line in 3D:** While we can define a line in 2D as the points (x, y) satisfying $ax + by + c = 0$, or equivalently in homogeneous coordinates, this does not generalize to 3D. More specifically, (x, y, z) such that $ax + by + cz + d = 0$ defines a plane in 3D. Moreover, while a line is uniquely identified by two or more collinear points, that does not define the line in its entirety. The simplest way to do so is to specify a line as all points $\mathbf{x} \in \mathbb{R}^3$ such that $\mathbf{x} = \mathbf{x}_1 + \lambda(\mathbf{x}_2 - \mathbf{x}_1)$, where $\mathbf{x}_1$ and $\mathbf{x}_2$ are two different points lying on the line, and $\lambda \in \mathbb{R}$. Several equivalent forms exist, and these definitions can be extended to homogeneous coordinates by appending a 1 value to each point.