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 22nd, 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.

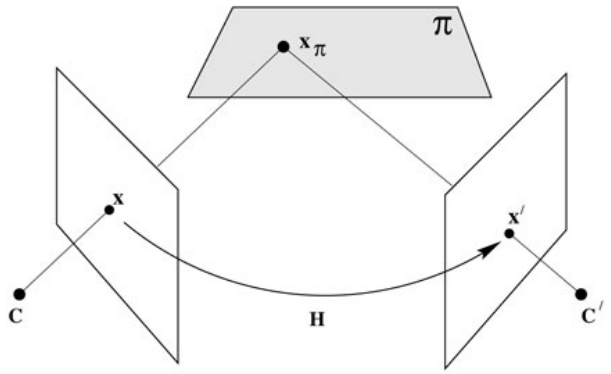


Figure 1.1: A homography **H** links all points $\mathbf{x}_\pi$ lying in plane π between two camera views **x** and **x'** in cameras C and C' respectively such that **x'** $=$ **Hx**. [*From Hartley and Zisserman*]

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 $\begin{bmatrix} \frac{x_i}{z_i} & \frac{y_i}{z_i} \end{bmatrix}$ (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

From Lecture 9 slide:
For Normalized image plane:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \equiv \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

With Plug in Z=0:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \equiv \begin{bmatrix} m_{11} & m_{12} & m_{14} \\ m_{21} & m_{22} & m_{24} \\ m_{31} & m_{32} & m_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Relating 2 camera views of the same 3D plane:

For x_1: $\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \equiv H_1 \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$

For $\mathbf{x}_2$: $\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \equiv H_2 \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$

Then we get: $\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \equiv H_1 \; H_2^{-1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$

So: $\mathbf{x}_1 \equiv H \; \mathbf{x}_2$ (proof of the existence of H)

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 $\mathbf{x_1^i} = [x_1^i \;\; y_1^i \;\; 1]$ 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

From lecture 10:
degrees of freedom number of $\mathbf{h}$:8

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

Q1.2.2

From lecture 10:
point pairs number are required :8/2 = 4

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

Q1.2.3

From lecture 10 and 11:

$$\lambda \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

Then we can get:

$x_2 = \frac{\lambda x_2}{\lambda} = \frac{ax_1+by_1+c}{gx_1+hy_1+i}$

$y_2 = \frac{\lambda y_2}{\lambda} = \frac{dx_1+ey_1+f}{gx_1+hy_1+i}$

Then, we get equation:

$x_2(gx_1 + hy_1 + i) = ax_1 + by_1 + c$

$y_2(gx_1 + hy_1 + i) = dx_1 + ey_1 + f$

Same as:

$x_2(gx_1 + hy_1 + i) - ax_1 - by_1 - c = 0$

$y_2(gx_1 + hy_1 + i) - dx_1 - ey_1 - f = 0$

Same as:

$gx_2x_1 + hy_1x_2 + ix_2 - ax_1 - by_1 - c = 0$

$gy_2x_1 + hy_1y_2 + iy_2 - dx_1 - ey_1 - f = 0$

Reshape H as H(:): $\begin{bmatrix} a & b & c & d & e & f & g & h & i \end{bmatrix}^T$

Then for just one pointer, $A^i = \begin{bmatrix} -x_1^i & -y_1^i & -1 & 0 & 0 & 0 & x_2^i x_1^i & x_2^i y_1^i & x_2^i \\ 0 & 0 & 0 & -x_1^i & -y_1^i & -1 & y_2^i x_1^i & y_2^i y_1^i & y_2^i \end{bmatrix}$

For all pointers from 0 to N: $A^i = \begin{bmatrix} -x_1^0 & -y_1^0 & -1 & 0 & 0 & 0 & x_2^0 x_1^0 & x_2^0 y_1^0 & x_2^0 \\ 0 & 0 & 0 & -x_1^0 & -y_1^0 & -1 & y_2^0 x_1^0 & y_2^0 y_1^0 & y_2^0 \\ \vdots & & & & & & & & \\ -x_1^N & -y_1^N & -1 & 0 & 0 & 0 & x_2^N x_1^N & x_2^N y_1^N & x_2^N \\ 0 & 0 & 0 & -x_1^N & -y_1^N & -1 & y_2^N x_1^N & y_2^N y_1^N & y_2^N \end{bmatrix}$

Formulation:

$$\begin{bmatrix} -x_1^0 & -y_1^0 & -1 & 0 & 0 & 0 & x_2^0 x_1^0 & x_2^0 y_1^0 & x_2^0 \\ 0 & 0 & 0 & -x_1^0 & -y_1^0 & -1 & y_2^0 x_1^0 & y_2^0 y_1^0 & y_2^0 \\ \vdots & & & & & & & & \\ -x_1^N & -y_1^N & -1 & 0 & 0 & 0 & x_2^N x_1^N & x_2^N y_1^N & x_2^N \\ 0 & 0 & 0 & -x_1^N & -y_1^N & -1 & y_2^N x_1^N & y_2^N y_1^N & y_2^N \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

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

From lecture,

A trivial solution for h: $\begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$

The A can't be full rank.

For if A is full rank, for $\mathbf{Ah} = 0$ will only has one $\begin{bmatrix} 0 & \dots & 0 \end{bmatrix}^T$ solution(trivial solution)

For there is non-trivial solution to find, A can't be full rank.

Impact: The number of nonzero singular values of A is equal to rank of 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:

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

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 $\mathbf{A}\mathbf{A}^T$. The columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{A}^\mathbf{T}\mathbf{A}$. We can use this fact to solve for $\mathbf{h}$ in the equation $\mathbf{A}\mathbf{h} = 0$. Using this knowledge, let us reformulate our problem of solving $\mathbf{A}\mathbf{x} = 0$. We want to minimize the error in solution in the least-squares sense. Ideally, the product $\mathbf{A}\mathbf{h}$ should be 0. Thus, the sum-squared error can be written as:

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

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

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

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 $\mathbf{A}\mathbf{h} = 0$ 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 $\mathbf{x_1} \equiv \mathbf{H}\mathbf{x_2}$, 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

From lecture9:

Since for for camera 1, $\mathbf{x_1} = \mathbf{K_1}[\mathbf{I}\ \ \mathbf{0}]\mathbf{X}$, then $\begin{bmatrix}\mathbf{I}\ 0\end{bmatrix}\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}$

can be written as $\begin{bmatrix}\mathbf{I}\end{bmatrix}\begin{bmatrix}X\\Y\\Z\end{bmatrix}$ for last column is all 0

Then we get: $\mathbf{x_1} \equiv \mathbf{K_1}\begin{bmatrix}X\\Y\\Z\end{bmatrix}$, same for camera2, we get: $\mathbf{x_2} \equiv \mathbf{K_2 R}\begin{bmatrix}X\\Y\\Z\end{bmatrix}$

Then, $\begin{bmatrix}X\\Y\\Z\end{bmatrix} \equiv \mathbf{R}^{-1}{\mathbf{K_2}}^{-1}\mathbf{x_2}$

So $\mathbf{x_1} \equiv \mathbf{K_1}\mathbf{R}^{-1}{\mathbf{K_2}}^{-1}\mathbf{x_2}$

$\mathbf{x_1} \equiv \mathbf{H}\mathbf{x_2}$

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

From lecture 9 slide:

For it's rotation about camera center, the matrix can be presented like: $\mathbf{X_2} = \mathbf{R_x}\mathbf{Y_2}$

$\mathbf{R_x} = \begin{bmatrix}1 & 0 & 0\\0 & cos\theta & -sin\theta\\0 & sin\theta & cos\theta\end{bmatrix}$ $\mathbf{H} = \mathbf{R_x}$

Then for 2θ, $\mathbf{H}^2$ = $\begin{bmatrix}1 & 0 & 0\\0 & cos\theta & -sin\theta\\0 & sin\theta & cos\theta\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\0 & cos\theta & -sin\theta\\0 & sin\theta & cos\theta\end{bmatrix}$

= $\begin{bmatrix}1 & 0 & 0\\0 & cos^2\theta - sin^2\theta & -2sin\theta cos\theta\\0 & 2sin\theta cos\theta & cos^2\theta - sin^2\theta\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\0 & cos2\theta & -sin2\theta\\0 & sin2\theta & cos2\theta\end{bmatrix}$ according to Trigonometric Function Formula

It's the same as using 2θ and 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

From lecture slide 9 and helpful Concepts:
For mapping the scene image to another viewpoint, need special cases that the points in the 3D world lie on a plane, or there is pure rotation between the two views. These special cases can't apply to any arbitrary scene image, so it's not completely 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

Prove method1:
Assume:
$x_1 - x_2 = \lambda(x_1 - x_3)$
With projection $\mathbf{P}$ in $\mathbf{x} = \mathbf{PX}$: $PX_1 - PX_2 = \lambda(PX_2 - PX_3)$
$P(X_1 - X_2) = \lambda P(X_2 - X_3)$
$(X_1 - X_2) = \lambda P^{-1}P(X_2 - X_3) = \lambda I(X_2 - X_3) = \lambda(X_2 - X_3)$
So it preserves lines.
Prove method2:
For $\mathbf{P}$ in $\mathbf{x} = \mathbf{PX}$, For coordinate [X,Y,Z] in $\mathbf{X}$:
Present line in 3D, [X,Y,Z], equation can be written as: $X = X_0 + ta; Y = Y_0 + tb; Z = Z_0 + tc$ (t is variable, a,b,c,X_0, Y_0, Z_0 are constant)
For coordinate [x,y] in $\mathbf{x}$, it can be written using [X,Y,Z] as:
$y = f\frac{Y}{Z}, x = f\frac{X}{Z}$
Then, $y = y = f\frac{Y_0+tb}{Z_0+tc} = f\frac{\frac{b}{c}(Z_0+tc)+Y_0-\frac{bZ_0}{c}}{Z_0+tc} = \frac{bf}{c} + f(Y_0 - \frac{bZ_0}{c})\frac{1}{Z_0+tc}$
Same for x: $x = \frac{af}{c} + f(X_0 - \frac{aZ_0}{c})\frac{1}{Z_0+tc}$
For t is variable, a,b,c,X_0, Y_0, Z_0 are constant, set $t' = \frac{1}{Z_0+tc}$, $y_0 = \frac{bf}{c}$,$b' = f(Y_0 - \frac{bZ_0}{c})$, $x_0 = \frac{af}{c}$,$a' = f(X_0 - \frac{aZ_0}{c})$
Then can write: $x = x_0 + t'a', y = y_0 + t'b'$($t'$ is variable, a', b', x_0, y_0 are constant),so it can also be written as $x = ky + b$.
So the projection $\mathbf{P}$ in $\mathbf{x} = \mathbf{PX}$ preserves lines.

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

From lecture and reference:
Difference: For FAST detector, it considers a circle of 16 pixels around the pixel under test. It checks if there exists a set of contiguous pixels which are all brighter than $I_p + threshold$, or darker than $I_p - threshold$. But for Harris corner detector, it shifts the image patch, computes horizontal and vertical gradients and get sum of squared differences between two patches.
Computational performance: the FAST detector is several times faster than Harris corner detector, but not so robust. It's fast for it has test that examines only the four pixels and can exclude a large number of non-corners, but may miss some corners and not robust to high levels of noise.

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

According to slide 7and8 and reference Introduction to BRIEF,7.2 Feature Descriptors 2:
BRIEF descriptor uses binary strings as feature point descriptor. It get random pointers in patch with center of corner and compare pixel values to create a binary feature vector out of patch.
Filterbanks are used to detect information like edges of whole image, smooth image and compute histograms and distribution of image gradients. It's based on the global image representation.
Yes, for it can detect the gradient and edge, we can use filter banks like Gabor filters in GIST as a descriptor.

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 distance have over a more conventional Euclidean distance measure in our setting?

Q2.1.3

From reference: BRIEF: Binary Robust Independent Elementary Features, Hamming distance, Local Feature Matching
Hamming distance: it's a way to measure the distance and similarity. For two strings representing binary descriptors, Hamming distance compute between two strings the number of positions at which the corresponding symbols(0 or 1) are different.
Nearest Neighbor: The nearest neighbor distance ratio (NNDR), or ratio test, finds the nearest neighbor to the feature descriptor and second nearest neighbor to the feature descriptor and divides the two.
For the two strings are binary descriptors. Using Hamming distance can be done extremely fast on modern CPUs that often provide a specific instruction to perform a XOR or bit count operation. And Euclidean distance isn't appropriate to be used in binary strings.

Q2.1.4 (10 points): Feature Matching

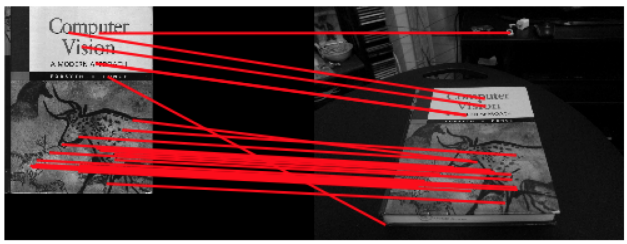


Figure 2.1: A few matched FAST feature points with the BRIEF descriptor.

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`.

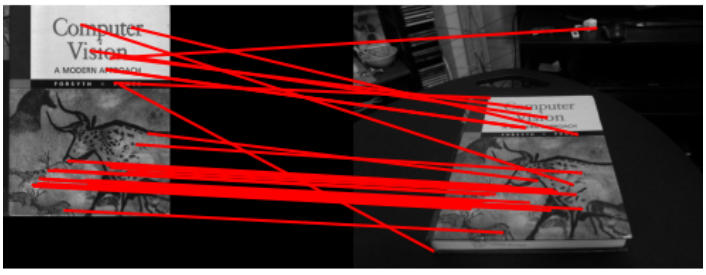
Use the provided helper function `corner_detection()` to compute the features, then build descriptors using `computeBrief()`, and finally compare them using `briefMatch()`. Use `plotMatches()` to visualize your matched points and include the result image in the first box below. An example is shown in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

```
import numpy as np
import cv2
import skimage.color
from helper import briefMatch
from helper import computeBrief
from helper import corner_detection
# Q2.1.4
def matchPics(I1, I2, opts):
        ratio = opts.ratio  #'ratio for BRIEF feature descriptor'
        sigma = opts.sigma  #'threshold for corner detection using FAST feature
        ↪  detector'


        # TODO: Convert Images to GrayScale
        I1_gray = skimage.color.rgb2gray(I1)
        I2_gray = skimage.color.rgb2gray(I2)

        # TODO: Detect Features in Both Images
        locs1 = corner_detection(I1_gray, sigma)
        locs2 = corner_detection(I2_gray, sigma)

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

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

        return matches, locs1, locs2

import cv2
from matchPics import matchPics
from helper import plotMatches
from opts import get_opts
def displayMatched(opts, image1, image2):
        matches, locs1, locs2 = matchPics(image1, image2, opts)
        #display matched features
        plotMatches(image1, image2, matches, locs1, locs2)
```

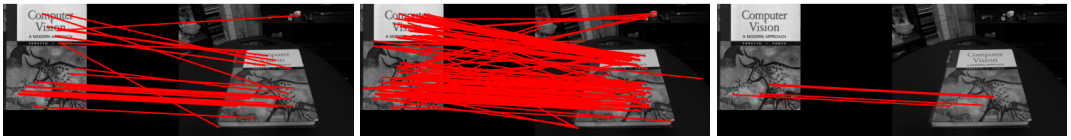
Q2.1.5 (10 points): Feature Matching and Parameter Tuning

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`.

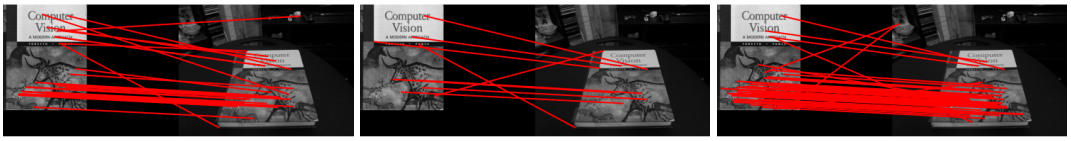
Conduct a small ablation study by running `displayMatch.py` with various `sigma` and `ratio` values. Include the figures displaying the matched features with various parameters in your writeup, and explain the effect of these two parameters respectively.

Q2.1.5

1. Change the ratio(keep sigma as 0.15):
The first image with ratio as 0.7, the second with ratio as 0.9, 3rd with ratio as 0.5



From result can see that with ratio increasing, the number of matched pointers increases. Because the ratio is max_ratio in skimage.feature.match_descriptors, with threshold of distances between first and second closest descriptor in the second set of descriptors increased, more matched pointers are detected.
2. Change the sigma(keep ratio as 0.7):
The first image with sigma as 0.15, the second with sigma as 0.2, 3rd with sigma as 0.1



From result can see that with sigma decreasing, the number of matched pointers increases. Because the sigma is $threshold$ in skimage.feature.corner_fast. This is threshold used in deciding whether the pixels on the circle are brighter, darker or similar. So if threshold is small, more pointers can be detected as corners. With corners increased, the matched points number also increased.

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:

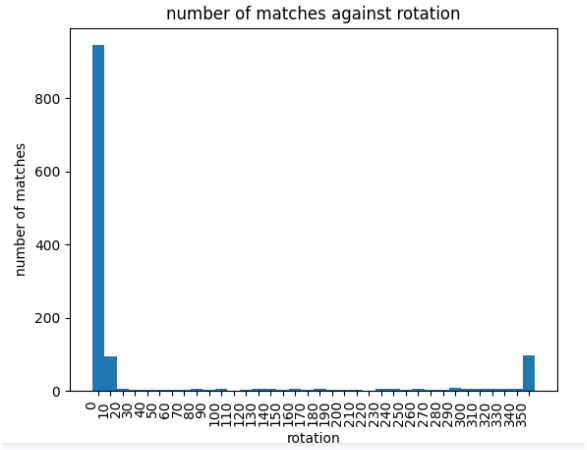
Q2.1.6 (a)

```
import cv2
from matchPics import matchPics
from opts import get_opts
from scipy import ndimage
import matplotlib.pyplot as plt
from helper import plotMatches
#Q2.1.6
def rotTest(opts):
        #Read the image and convert to grayscale, if necessary
        img = cv2.imread('../data/cv_cover.jpg')

        hist=[]
        rotations =[]
        for i in range(36):
                #Rotate Image
                rotation_img = ndimage.rotate(img, i*10)
                #Compute features, descriptors and Match features
                matches, locs1, locs2 = matchPics(img, rotation_img, opts)
                #Update histogram
                hist.append(matches.shape[0])
                rotations.append(str(i*10))
                print(matches.shape)
                if i==9 or i==18 or i==27:
                plotMatches(rotation_img, img, matches, locs1, locs2)

        #Display histogram
        plt.xticks(rotation=90, ha='right')
        plt.bar(rotations,hist,align='center',width = 1)
        plt.xlabel('rotation')
        plt.ylabel('number of matches')
        plt.title('number of matches against rotation')
        plt.show()

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

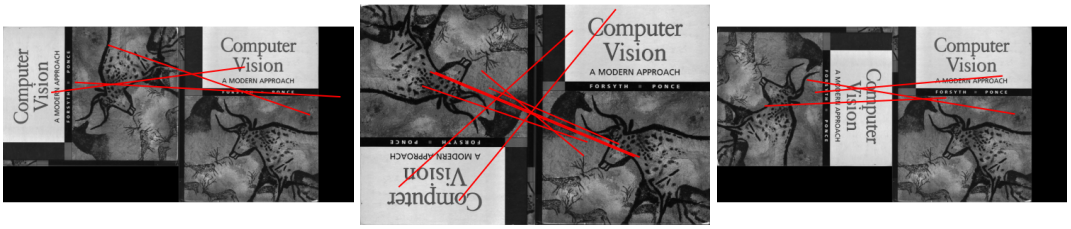
Visualize the histogram and the feature matching result at three different orientations and include them in your write-up. Explain why you think the BRIEF descriptor behaves this way.

Q2.1.6 (b)

histogram:



feature matching result at different orientations of 90,180,270:



From result we can see that the number of detected pointers decreases a lot if there is rotation. For the BRIEF descriptor is not stable under rotations. This may because that with rotation, the principal direction coordinate axis will change and when BRIEF uses binary features vector and Hamming distance to get similarity, the information in the patch will change with rotation and results bad matching result.

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(part1)

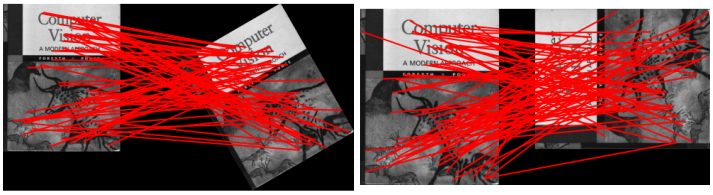
code:(explaining and result in next page)

```
def computeBrief_rotation1(img, locs):

        patchWidth = 9
        nbits = 256
        compareX, compareY = makeTestPattern(patchWidth,nbits)
        m, n = img.shape

        halfWidth = patchWidth//2

        locs = np.array(list(filter(lambda x: halfWidth <= x[0] < m-halfWidth and
        ↪  halfWidth <= x[1] < n-halfWidth, locs)))
        desc = np.array([list(map(lambda x: computePixel_rotation1(img, x[0],
        ↪  x[1], patchWidth, c), zip(compareX, compareY))) for c in locs])

        #print(desc.shape)
        #desc = np.array(desc.sum(axis=1))
        #print(desc.shape[1])
        return desc, locs


def rotate(pre, center, radiant):
        cx, cy = center
        px, py = pre
        qx = cx + math.cos(radiant) * (px - cx) - math.sin(radiant) * (py - cy)
        qy = cy + math.sin(radiant) * (px - cx) + math.cos(radiant) * (py - cy)
        return int(qx), int(qy)

def computePixel_rotation1(img, idx1, idx2, width, center):
        h,w=img.shape
        half = width//2
        brightest =0
        b_i=0
        b_j=0
        # get brightest pointer
        for i in range(center[0] - half,center[0]+half):
                for j in range(center[1]-half,center[1]+half):
                        value=img[i,j]
                        if value>brightest:
                                # record
                                b_i=i
                                b_j=j
                                brightest=value
        # get radians
        radians = math.atan2(b_i - center[0], b_j - center[1])
        halfWidth = width // 2
(see the rest in next page)
```

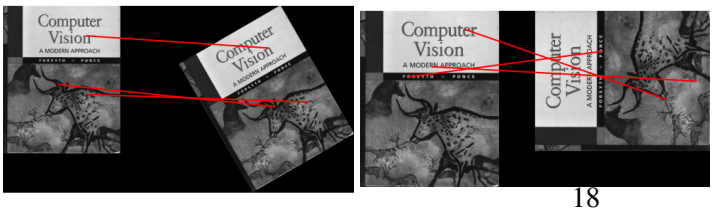
Q2.1.7.1(part2)

```
        col1 = idx1 % width - halfWidth
        row1 = idx1 // width - halfWidth
        col2 = idx2 % width - halfWidth
        row2 = idx2 // width - halfWidth

        pointer1_0=int(center[0]+row1)
        pointer1_1 = int(center[1]+col1)
        pointer2_0 = int(center[0]+row2)
        pointer2_1 = int(center[1]+col2)

        # rotate pointer around center
        pointer1_0,pointer1_1=rotate((pointer1_0,pointer1_1),(center[0],center[1]),radians)
        pointer2_0, pointer2_1 = rotate((pointer2_0, pointer2_1), (center[0],
        ↪  center[1]), radians)

        pointer2_0=min(pointer2_0,h-1)
        pointer1_0 = min(pointer1_0, h-1)
        pointer2_1 = min(pointer2_0, w-1)
        pointer1_1 = min(pointer1_0, w-1)

        return 1 if img[pointer1_0][pointer1_1] < img[pointer2_0][pointer2_1]
        ↪  else 0
```

Explaining:
For BRIEF, need to find the rotation firstly, then rotate the chosen pointers in patch and use these rotated pointers to get binary string.
To get the rotation, need to get feature like edge or special pointers. According to suggestion from TA, I choose to use the brightest pointer in the patch as feature pointer. Then I obtain the angle between the line (which is from that special pointer to the center) and the x-axis.
Then ,just rotate every random pointer with that angle. So that make sure every patch's main axis is the same.
Effectiveness of algorithm:(with default parameter sigma=0.15,ratio=0.7) rotation with 30 and 90 degree:



Compared with original algorithm result(same parameter):



A lot more pointers are matched, so it's efficient.

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.

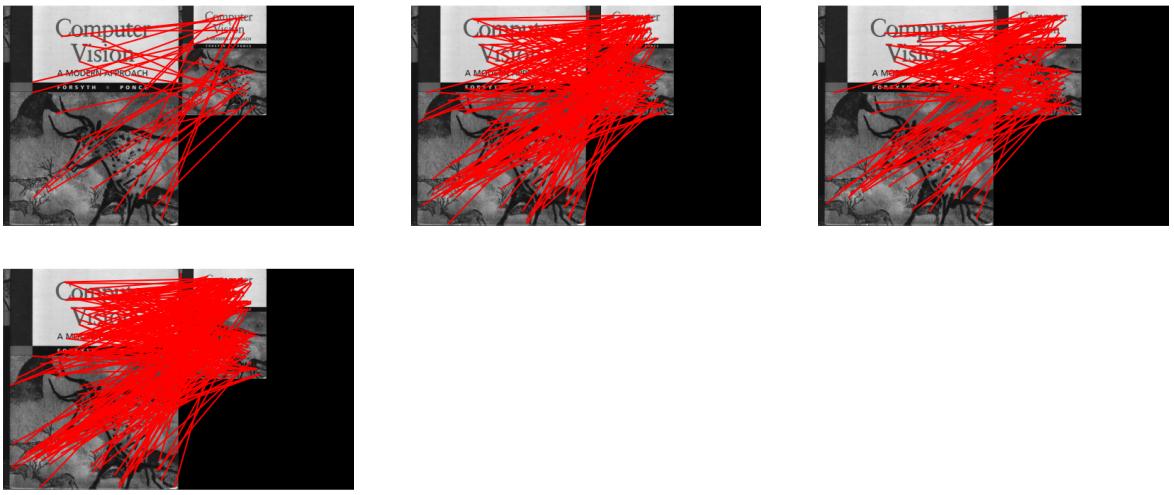
Q2.1.7.2(part1)

code:(explaining and result in next page)

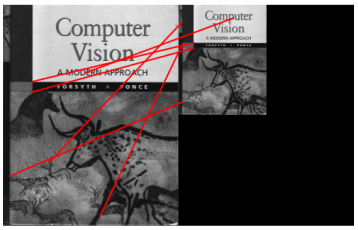
```
def computePixel_scale(img, idx1, idx2, width, center,scale):
        halfWidth = width // 2
        col1 = idx1 % width - halfWidth
        row1 = idx1 // width - halfWidth
        col2 = idx2 % width - halfWidth
        row2 = idx2 // width - halfWidth
        p1x=int(center[0]+row1*scale)
        p1y=int(int(center[1]+col1*scale))
        p2x=int(center[0]+row2*scale)
        p2y=int(center[1]+col2*scale)
        h, w = img.shape
        p1x = min(p1x, h - 1)
        p2x = min(p2x, h - 1)
        p1y= min(p1y, w - 1)
        p2y = min(p2y, w - 1)
        return 1 if img[p1x][p1y] < img[p2x][p2y] else 0

def computeBrief_scale(img, locs):
        patchWidth = 9
        nbits = 256
        compareX, compareY = makeTestPattern(patchWidth, nbits)
        m, n = img.shape
        ori_locs = np.copy(locs)
        #print(locs.shape)
        halfWidth = patchWidth // 2
        locs = np.array(
        list(filter(lambda x: halfWidth <= x[0] < m - halfWidth and halfWidth <=
        ↪  x[1] < n - halfWidth, locs)))
        # set layer number
        img= gaussian_filter(img, sigma=1.6)
        layer_num = 6
        scale_num = 1/layer_num
        scale = scale_num
        desc=np.array([list(map(lambda x: computePixel_scale(img, x[0], x[1],
        ↪  patchWidth, c,scale), zip(compareX, compareY))) for c in locs])
        for i in range(layer_num):
                scale += scale_num
                desc_one =np.array([list(map(lambda x: computePixel_scale(img,
                ↪  x[0], x[1], patchWidth, c,scale), zip(compareX, compareY)))
                ↪  for c in ori_locs])
                #print(desc_one.shape)
                desc=np.vstack((desc,desc_one))
                locs = np.vstack((locs, ori_locs))
        return desc, locs
(see the rest in next page)
```

Q2.1.7.2(part2)

Explaining:
For scale stability, according to reference and TA, need to build a scale pyramid. To build it firstly build gaussian kernels(from reference the best is 1.6), use a loop with different scale. Test result with layer number.
Test for image half size of original one with default parameters(sigma=0.15,ratio=0.7)
Result for(layer number as 2,4,6,8):



Compared with original algorithm result(same parameter):



More matched pointers can be found, so there is the improvement.

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

```
def computeH(x1, x2):
        #Q2.2.1
        #Compute the homography between two sets of points
        N = x1.shape[0]
        # A:2N * 9
        A = np.zeros((N*2, 9))
        for i in range(N):
                px1 = x1[i, 0]
                py1 = x1[i, 1]
                px2 = x2[i, 0]
                py2 = x2[i, 1]
                A[i*2, :] = np.array([px2, py2, 1, 0, 0, 0, -px1*px2, -px1*py2,
                ↪  -px1])
                A[i*2+1, :] = np.array([0, 0, 0, px2, py2, 1, -px2*py1, -py2*py1,
                ↪  -py1])
        u, s, vh = np.linalg.svd(A)
        # s in descending order, so choose the smallest eigenvalue of ATA(in
        ↪  column 9 of the matrix V)
        #print(vh.shape)
        H2to1 = vh[8, :]
        H2to1 = H2to1.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(part 1)

```
def computeH_norm(x1, x2):
        #Q2.2.2
        x1=x1.astype(float)
        x2 = x2.astype(float)
        #Compute the centroid of the points
        # for x1 moving transform
        p1_center_x = np.mean(x1[:, 0])
        p1_center_y = np.mean(x1[:, 1])
        # for x2
        p2_center_x = np.mean(x2[:, 0])
        p2_center_y = np.mean(x2[:, 1])
        #Shift the origin of the points to the centroid
        # for x1 moving transform
        for i in range(x1.shape[0]):
                x1[i, 0] = x1[i, 0] - p1_center_x
                x1[i, 1] = x1[i, 1] - p1_center_y
        # for x2
        for i in range(x2.shape[0]):
                x2[i, 0] = x2[i, 0] - p2_center_x
                x2[i, 1] = x2[i, 1] - p2_center_y
        #print(x2)
        #Normalize the points so that the largest distance from the origin is
        ↪  equal to sqrt(2)
        sqrt2 = np.sqrt(2)
        p1_max_dis=0
        p2_max_dis = 0
        for i in range(x1.shape[0]):
                dis_p1 = np.linalg.norm(x1[i, :])
                p1_max_dis = np.maximum(p1_max_dis, dis_p1)
        for i in range(x2.shape[0]):
                dis_p2 = np.linalg.norm(x2[i, :])
                p2_max_dis = np.maximum(p2_max_dis, dis_p2)
        #Similarity transform 1
        # for have done moving, so use matrix for scale
        T1 = np.zeros((3,3))
        T1[0,0] = sqrt2/p1_max_dis
        T1[1, 1] = sqrt2 / p1_max_dis
        T1[2, 2] = 1
        new_col = np.ones((x1.shape[0], 1))
        new_x1 = np.hstack((x1, new_col))
        for i in range(x1.shape[0]):
                new_loc1 = (T1@new_x1[i, 0:3].T)
                x1[i, :] = new_loc1[0:2]
                #print(x1[i,:])
        #Similarity transform 2
```

(see the rest in next page)

Q2.2.2(part 2)

```
# Similarity transform 2
# matrix for scale
T2 = np.zeros((3, 3))
T2[0, 0] = sqrt2 / p2_max_dis
T2[1, 1] = sqrt2 / p2_max_dis
T2[2, 2] = 1
new_col = np.ones((x2.shape[0], 1))
new_x2 = np.hstack((x2, new_col))
for i in range(x2.shape[0]):
        new_loc2 = (T2 @ new_x2[i, 0:3].T)
        x2[i, :] = new_loc2[0:2]
#Compute homography
H2to1 = computeH(x1,x2)
#print(H2to1)
#Denormalization
# moving matrix @ scaling matrix for x1
# moving matrix
T1_cen = np.zeros((3, 3))
T1_cen[0, 0] = 1
T1_cen[1, 1] = 1
T1_cen[2, 2] = 1
T1_cen[0, 2] = -p1_center_x
T1_cen[1, 2] = -p1_center_y
T1 = T1 @ T1_cen
# moving matrix @ scaling matrix for x2
T2_cen = np.zeros((3, 3))
T2_cen[0, 0] = 1
T2_cen[1, 1] = 1
T2_cen[2, 2] = 1
T2_cen[0, 2] = -p2_center_x
T2_cen[1, 2] = -p2_center_y
T2 = T2 @ T2_cen
H2to1 = np.linalg.inv(T1)@H2to1@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 homography.

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.**

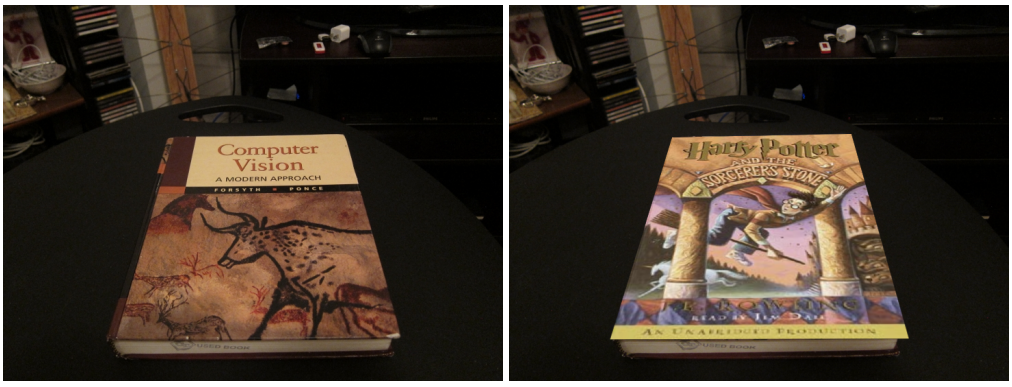


Figure 2.2: An ordinary textbook (left). Harry-Potterized book (right)

Q2.2.3(part1)

```
def computeH_ransac(locs1, locs2, opts):
        #Q2.2.3
        #Compute the best fitting homography given a list of matching points
        max_iters = opts.max_iters  # the number of iterations to run RANSAC for
        inlier_tol = opts.inlier_tol # the tolerance value for considering a
        ↪  point to be an inlier

        times = 0
        n_choice = 4
        N = locs2.shape[0]

        bestH2to1 = None
        best_inliers = None
        best_in_num = 0
        np.random.seed(2020)
        if N <= n_choice:
                max_iters = 1
        while times < max_iters:
                # Draw n points uniformly at random (from left image)
                # use length for a must be 1-D
                if N>n_choice:
                        random_indexs=np.random.choice(N, n_choice,
                        ↪  replace=False)
                else:
                        random_indexs = np.arange(N)
                #print(random_indexs)
                random_x1 = locs1[random_indexs]
                #print(random_x1)
                # Fit (homography) warp to these n points and their
                ↪  correspondences
                random_x2 = locs2[random_indexs]
                H = computeH_norm(random_x1,random_x2)
                # Find inliers among the remaining left-image points(warped
                ↪  positions land close to right-image)
                new_col = np.ones((N, 1))
                new_loc2 = np.hstack((locs2, new_col))
                inlier_num = 0
                inliers = np.zeros((N))
                for i in range(N):
                        warp_x1 = H @ new_loc2[i,0:3].T
                        # divide by z ro get x,y
                        z = warp_x1[-1]
                        if z!=0:
                                warp_x1_new = warp_x1[0:2]/z
                        diff = warp_x1_new - locs1[i,:]
                        dis = np.linalg.norm(diff)
```

(see the rest in next page)

Q2.2.3(part2)

```
#print(dis)
                        if dis < inlier_tol:
                                inlier_num +=1
                                inliers[i] = 1
                        else:
                                inliers[i]=0
                        #print(H)
                        # largest inlier set
                if inlier_num > best_in_num:
                        bestH2to1 = H
                        best_in_num = inlier_num
                        best_inliers=inliers
                times += 1
        inliers=best_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(part1)

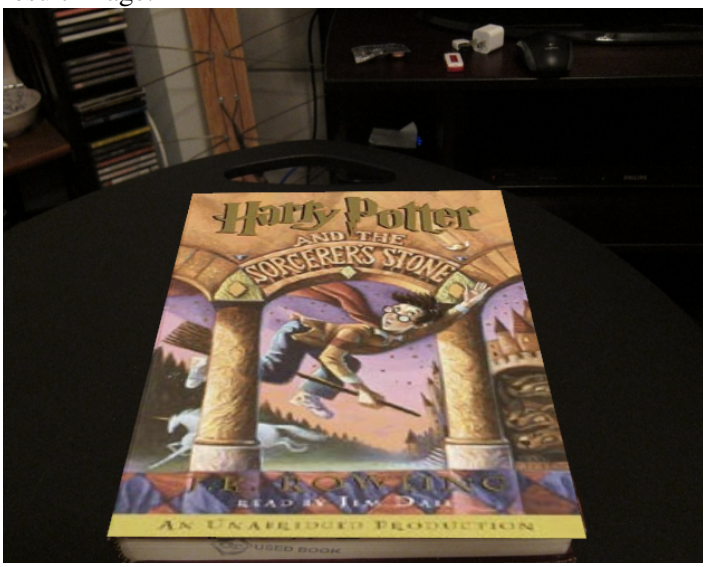
code:

```
def compositeH(H2to1, template, img):
        #For warping the template to the image, we need to invert it.
        H2to1_inv = np.linalg.inv(H2to1)
        #Create mask of same size as template
        mask = np.full(template.shape,True).astype(template.dtype)
        #print(img.shape)
        #Warp mask by appropriate homography
        h,w,c = img.shape
        mask_warp=cv2.warpPerspective(mask,H2to1_inv,(w,h))
        #Warp template by appropriate homography
        template_warp = cv2.warpPerspective(template, H2to1_inv,(w,h))
        #Use mask to combine the warped template and the image
        composite_img = np.zeros(img.shape).astype(img.dtype)
        #print(template_warp.shape)
        for i in range(c):
                for j in range(h):
                        for k in range(w):
                                if mask_warp[j,k,i] !=0:
                                        composite_img[j,k,i] =
                                        ↪  template_warp[j,k,i]
                                else:
                                        composite_img[j, k, i] = img[j, k, i]
        return composite_img

def warpImage(opts):
        cv_c = cv2.imread('../data/cv_cover.jpg')
        cv_d = cv2.imread('../data/cv_desk.png')
        hp_c = cv2.imread('../data/hp_cover.jpg')
        matches, locs1, locs2 = matchPics(cv_c, cv_d, opts)
        # get matched locs
        locs1 = locs1[matches[:, 0]]
        locs2 = locs2[matches[:, 1]]
        # solve: it is not filling up the same space as the book
        # for two size are different
        h_diff = hp_c.shape[0]/cv_c.shape[0]
        w_diff = hp_c.shape[1] / cv_c.shape[1]
        # x,y swap
        locs2[:,[0,1]] = locs2[:,[1,0]]
        locs1[:, [0, 1]] = locs1[:, [1, 0]]
        locs1[:,1] = locs1[:,1]*h_diff
        locs1[:, 0] = locs1[:, 0] * w_diff
        H,inliers = computeH_ransac(locs1,locs2,opts)
        #print(H)
        composite_img = compositeH(H,hp_c,cv_d)
        # show
        cv2.imshow("composite img", composite_img)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
```


(see the rest (result image) in next page)

Q2.2.4(part2)

result image:

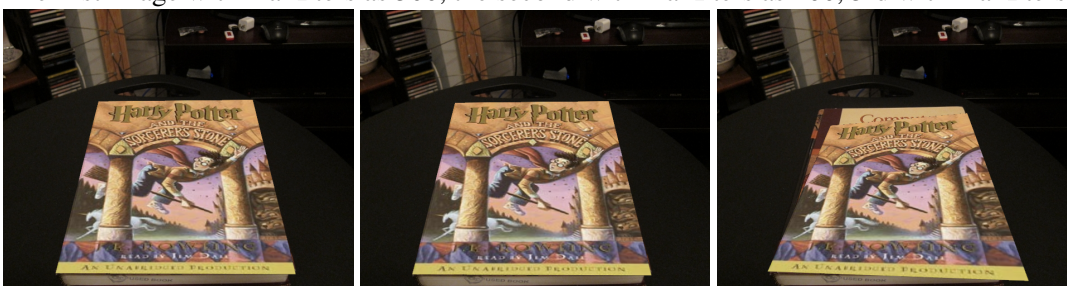


Q2.2.5 (10 points): RANSAC Parameter Tuning

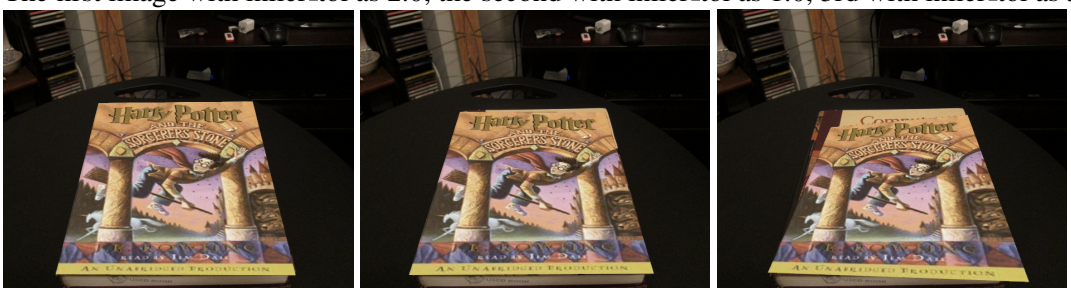
Just like how we tune parameters for feature matching, there are two tunable parameters in RANSAC as well. You can change the values by changing their default fields or by command-line arguments. For example, `python HarryPotterize.py --max_iters 500 --inlier_tol 2.0`. Conduct a small ablation study 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

1. Change the max_iters(keep inlier_tol as 2.0):
The first image with max_iters as 500, the second with max_iters as 100, 3rd with max_iters as 50



From result can see that with max_iters decreasing, the result becomes bad for in computeH_ransac, there isn't enough loops to find the best result with largest number of inliers.
2. Change the inlier_tol(keep max_iters as 500):
The first image with inlier_tol as 2.0, the second with inlier_tol as 1.0, 3rd with inlier_tol as 5.0



From result can see that with inlier_tol decreasing or increasing, the result isn't as good as 2.0. For if it's too large, some points with large distance that shouldn't be inliers may counted. If it's too small, some true inliers won't be counted. Both os these may cause the result becomes bad.

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

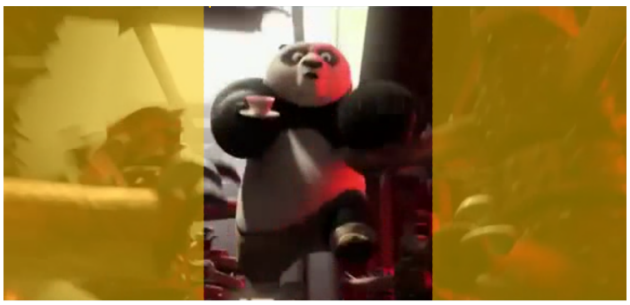


Figure 3.2: Rendering video on a moving target

`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).

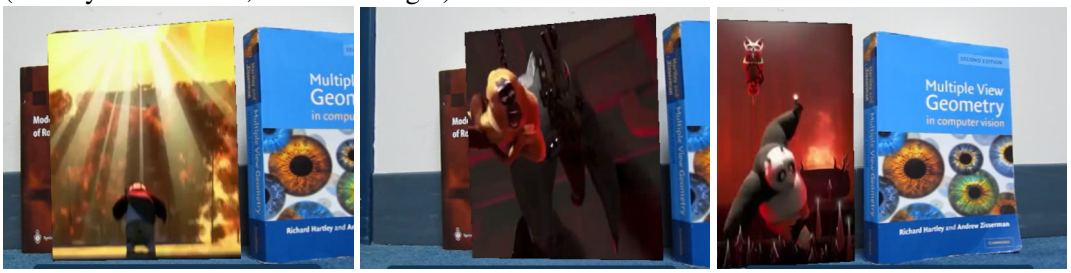
Finally, note that this is a very time-intensive job and may take many hours on a single core (parallel processing on 8 cores takes around 30 minutes). Debug before running your full script (e.g. by saving a few early AR frames and verifying that they look correct).

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(par1)

three screenshots of ar.avi at three distinct timestamps
(overlay near the left, center and right)



Result video: AR video
code:

```
def crop_center(source,template):
        t_h,t_w, t_c = template.shape
        s_h, s_w,s_c = source.shape
        # Crop black edges
        source = source[int(t_h//8*0.8):int(t_h//8*5.75), :, :]
        # keep center part
        ratio = t_h/t_w
        s_w_new = int(s_h//ratio)
        center_left = int((s_w-s_w_new)//2)
        crop_source = source[:,center_left:center_left+s_w_new,:]
        return crop_source

def process_one(opts,cv_c,cv_book,source):
        source = crop_center(source,cv_c)
        matches, locs1, locs2 = matchPics(cv_c, cv_book, opts)
        # get matched locs
        locs1 = locs1[matches[:, 0]]
        locs2 = locs2[matches[:, 1]]
        # solve: it is not filling up the same space as the book
        # for two size are different
        h_diff = source.shape[0] / cv_c.shape[0]
        w_diff = source.shape[1] / cv_c.shape[1]
        # x,y swap
        locs2[:, [0, 1]] = locs2[:, [1, 0]]
        locs1[:, [0, 1]] = locs1[:, [1, 0]]
        locs1[:, 1] = locs1[:, 1] * h_diff
        locs1[:, 0] = locs1[:, 0] * w_diff
        #print(locs1.shape, locs2.shape)
        H, inliers = computeH_ransac(locs1, locs2, opts)
        # print(H)
        composite_img = compositeH(H, source, cv_book)
        # show
        #cv2.imshow("composite img", composite_img)
        # cv2.waitKey(0)
        # cv2.destroyAllWindows()
        return composite_img
```


(see the rest in the next page)

Q3.1(part2)

```
if __name__ == "__main__":
        # load
        book_frames=loadVid("../data/book.mov")
        cv_c=cv_c = cv2.imread('../data/cv_cover.jpg')
        ar_frames=loadVid("../data/ar_source.mov")
        # get FPS
        cap = cv2.VideoCapture("../data/ar_source.mov")
        fps = int(cap.get(cv2.CAP_PROP_FPS))
        print(fps)
        opts = get_opts()
        # make sure same size
        h, w = book_frames[0].shape[:2]
        out = cv2.VideoWriter('result_ar.avi',
        ↪  cv2.VideoWriter_fourcc(*'MJPG'),fps, (w, h))
        length = np.minimum(len(book_frames),len(ar_frames))
        for i in range(length):
        # get img
                composite_img=process_one(opts,cv_c,book_frames[i],ar_frames[i])
                out.write(composite_img)
                print("process: "+str(i+1)+"/"+str(length))
        out.release()
```

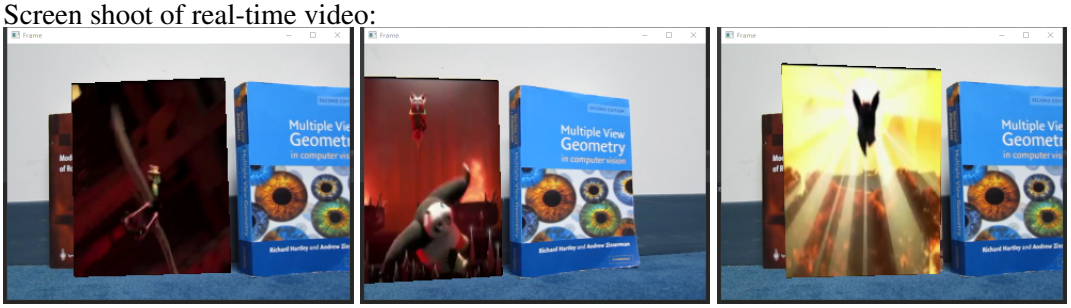
3.2 Make Your AR Real Time (Extra Credit - 15 points)

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(part1)

Result:(mean FPS): `mean fps: 30.032512669621358`

Screen shoot of real-time video:



To make it real-time, I firstly tried to record the time of each step:

```
processing: 3
corner: 0.1260361671447754
loc: 0.0009968280792236328
desc: 2.2009615898132324
loc: 0.0020003318786621094
desc: 2.2689976692199707
brief: 4.472956418991089
match: 0.1970064640045166
matchPic: 4.805994749069214
ransac: 0.16303014755249023
composite: 2.304965019226074
7.308000802993774
fps: 0.13683632869749315
```

I find that the most time costing part are computeBrief, computeH_ransac and cv2.imshow('Frame', composite_img),compositeH

Methods:

1. Resize the images to make all of them half of original size and use small image for processing. And resize it back to original size in showing image.
2. For image of cv_cover is not changed, and used in every loop so just get Brief descriptor of cv_cover in the begining, then use the locs and des1 of it in each loop. So only need to get the BRIEF of frame of video in each loop.
3. Change parameters of sigma, ratio, max_iters, nbits... and find ones that can make processing fast without making the result of showing image looks very bad.
4. Optimize my code. For example , in compositeH, I find better way to get the result without using loops as before.
5. Optimize the original code. For example, in computeBrief, not using so many % to get random x and y, but get random direction at first.

(See the rest in next page)

Q3.2(part2)

6. Use the precious H. Get the hashing value of previous frame and frame for now using hashing. Then get difference of hashing value. If the hashing value difference is more than threshold, update H, or just use old one. Also, make sure to update H every 6 steps.
7. Other changes like use cv2.waitKey(1) to make cv2.imgshow() vey fast.
Code:

```
def process_one(opts, locs1, desc1, cv_c, cv_book, source, H, flag):
        source = crop_center(source, cv_c)
        time_match = time.time()
        if flag == True:
                matches, locs2 = matchPics(desc1, cv_book, opts)
                #print("matchPic: "+str(time.time() - time_match))
                # get matched locs
                locs1_new = locs1[matches[:, 0]]
                locs2 = locs2[matches[:, 1]]
                # solve: it is not filling up the same space as the book
                # for two size are different
                h_diff = source.shape[0] / cv_c.shape[0]
                w_diff = source.shape[1] / cv_c.shape[1]
                # x,y swap
                locs2[:, [0, 1]] = locs2[:, [1, 0]]
                locs1_new[:, [0, 1]] = locs1_new[:, [1, 0]]
                locs1_new[:, 1] = locs1_new[:, 1] * h_diff
                locs1_new[:, 0] = locs1_new[:, 0] * w_diff
                # print(locs1.shape, locs2.shape)
                time_rans = time.time()
                H, inliers = computeH_ransac(locs1_new, locs2, opts)
        #H= computeH_norm(locs1_new, locs2)
        #H, mask = cv2.findHomography(locs1, locs2, cv2.RANSAC, 2.0)
        #print("ransac: "+str(time.time()-time_rans))
        # print(H)

        time_comp = time.time()
        composite_img = compositeH(H, source, cv_book)
        #print("composite: "+str(time.time() - time_comp))

        return composite_img,H
```

(see the rest in next page)

Q3.2(part3)

```
if __name__ == "__main__":
        # load
        cap1 = cv2.VideoCapture("../data/book.mov")
        cv_c = cv_c = cv2.imread('../data/cv_cover.jpg')
        cap2 = cv2.VideoCapture("../data/ar_source.mov")
        opts = get_opts()
        if (cap1.isOpened() == False) or (cap2.isOpened() == False):
                print("Error opening video stream or file")
        i=0
        prev_frame_time = 0
        # used to record the time at which we processed current frame
        new_frame_time = 0
        # Read until video is completed
        ratio = opts.ratio  # 'ratio for BRIEF feature descriptor'
        sigma = opts.sigma  # 'threshold for corner detection using FAST feature
        ↪  detector'
        cv_c = cv2.resize(cv_c, (175, 220), interpolation=cv2.INTER_AREA)
        I1_gray = skimage.color.rgb2gray(cv_c)
        locs1 = corner_detection(I1_gray, sigma)
        desc1, locs1 = computeBrief_fast(I1_gray, locs1)
        fps_count =0
        H=None
        time_all_process = time.time()
        pre_frame = None
        while (cap1.isOpened()) and (cap2.isOpened()):
                i += 1
                # Capture frame-by-frame
                ret1, book_frame = cap1.read()
                ret2, ar_frame = cap2.read()
                if ret1 and ret2:
                        cv_c = cv2.resize(cv_c, (175,220),
                        ↪  interpolation=cv2.INTER_AREA)
                        book_frame=cv2.resize(book_frame, (320,240),
                        ↪  interpolation=cv2.INTER_AREA)
                        ar_frame=cv2.resize(ar_frame, (320,180),
                        ↪  interpolation=cv2.INTER_AREA)
                        if(i>1):
                                hash_now=imagehash.average_hash(Image.fromarray(book_frame,
                                ↪  'RGB'))
                                hash_pre =
                                ↪  imagehash.average_hash(Image.fromarray(pre_frame,
                                ↪  'RGB'))
                                diff = np.abs(hash_now - hash_pre)
                        #time_all = time.time()
                        if(i%6==1 or diff>40):
                                composite_img, H = process_one(opts,
                                ↪  np.copy(locs1), np.copy(desc1),cv_c,
                                ↪  book_frame, ar_frame, H, True)
```

(see rest in next page)

Q3.2(part4)

```
                        else:
                                composite_img, H = process_one(opts,
                                ↪  np.copy(locs1), np.copy(desc1), cv_c,
                                ↪  book_frame, ar_frame, H,
                                False)
                        new_frame_time = time.time()
                        fps = 1 / (new_frame_time - prev_frame_time)
                        fps_count+=fps
                        prev_frame_time = new_frame_time
                        print("fps: " + str(fps))
                        composite_img=cv2.resize(composite_img, (640, 480),
                        ↪  interpolation=cv2.INTER_AREA)
                        cv2.imshow('Frame', composite_img)
                        # Press Q on keyboard to  exit
                        temp = cv2.waitKey(1)
                        pre_frame = book_frame
                else:
                        break
        # When everything done, release the video capture object
        print("mean fps: "+str(i/(time.time()-time_all_process)))
        cap1.release()
        cap2.release()
        cv2.destroyAllWindows()
    def computeBrief_fast(img, locs):

def computeBrief_fast(img, locs):
        patchWidth = 9
        nbits = 68
        compareX_row, compareX_col, compareY_col, compareY_row =
        ↪  makeTestPattern_fast(patchWidth,nbits)
        #print(compareX_row.shape)
        m, n = img.shape
        halfWidth = patchWidth//2
        time_loc = time.time()
        locs = np.array(list(filter(lambda x: halfWidth <= x[0] < m-halfWidth and
        ↪  halfWidth <= x[1] < n-halfWidth, locs)))
        time_desc = time.time()
        desc=[]
        for c in locs:
                desc_one=[]
                for i in range(len(compareX_row)):
                        desc_one.append(computePixel_fast(img,
                        ↪  compareX_row[i][0], compareX_col[i][0],
                        compareY_col[i][0], compareY_row[i][0], patchWidth, c))
                desc.append(desc_one)
        desc=np.array(desc)
        return desc, locs
```

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(part1)

original images(left and right image):

the final panorama result image:

code:

```
def get_move(img,H):
        h, w, c = img.shape
        x=np.arange(w)
        y=np.arange(h)
        xv, yv = np.meshgrid(x, y)
        new_x = xv*H[0][0]+yv*H[0][1]+H[0][2]
        new_y = xv * H[1][0] + yv * H[1][1] + H[1][2]
        return new_x.min(),new_y.min()
```

(see the rest in the next page)

Q4(part2)

```
def compositeH_pano(H2to1, template, img):
        H2to1_inv = np.linalg.inv(H2to1)
        # Create mask of same size as template
        mask = np.full(template.shape, True).astype(template.dtype)
        # Warp mask by appropriate homography
        # move with matrix
        l,d=get_move(template,H2to1_inv)
        l=int(-l)-100
        d=int(-d)-100
        move = np.array([[1.0, 0, l], [0, 1, d], [0, 0, 1]])
        H2to1_inv=move@H2to1_inv
        img=cv2.copyMakeBorder(img, 0, d, 0, l, cv2.BORDER_CONSTANT, None, value
        ↪  = 0)
        h, w, c = img.shape
        img=cv2.warpPerspective(img, move, (w, h))
        mask_warp = cv2.warpPerspective(mask, H2to1_inv, (w, h))
        # Warp template by appropriate homography
        template_warp = cv2.warpPerspective(template, H2to1_inv, (w, h))
        # Use mask to combine the warped template and the image
        composite_img = np.zeros(img.shape).astype(img.dtype)
        # print(template_warp.shape)
        for i in range(c):
                for j in range(h):
                        for k in range(w):
                                if mask_warp[j, k, i] != 0:
                                        composite_img[j, k, i] = template_warp[j,
                                        ↪  k, i]
                                else:
                                        composite_img[j, k, i] = img[j, k, i]
        return composite_img

left = cv2.imread('../data/d1_l.jpg')
right = cv2.imread('../data/d1_r.jpg')
ratio = left.shape[0]/ left.shape[1] # w/h
left= cv2.resize(left, (640,int(640*ratio)), interpolation = cv2.INTER_AREA)
right= cv2.resize(right, (640,int(640*ratio)), interpolation = cv2.INTER_AREA)
opts = get_opts()
matches, locs1, locs2 = matchPics(left, right, opts)
# get matched locs
locs1 = locs1[matches[:, 0]]
locs2 = locs2[matches[:, 1]]
# x,y swap
locs2[:,[0,1]] = locs2[:,[1,0]]
locs1[:, [0, 1]] = locs1[:, [1, 0]]
H,inliers = computeH_ransac(locs1,locs2,opts)
composite_img = compositeH_pano(H,left,right)
cv2.imwrite("../img/d1_result.jpg", composite_img)
cv2.imshow("composite img", composite_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

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.

References