16-720 Computer Vision: Homework 2 (Spring 2022)
Augmented Reality with Planar Homographies

Instructor: Deva Ramanan
TAs: Gautam Gare, Tarasha Khurana, Neehar Peri
Due: Thursday, Feb 17, 2022 23:59:59

Instructions/Hints

1. You will submit both a pdf writeup and a zip of your code to Gradescope. Zip your code into a single file named <**AndrewId**>**.zip**. See the complete submission checklist at the end to ensure you have everything. Handwritten writeups will not be accepted.

2. Each question (for points) is marked with a **Q**.

3. **Start early!** This homework may take a long time to complete.

4. **Attempt to 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.

5. **In your PDF, start a new page for each question, and indicate the answer/page(s) correspondence carefully when submitting on Gradescope.** For some questions, this may leave a lot of blank space. If you skip a written question, just submit a blank page for it. This makes your work much easier to grade.

6. **Some questions will ask you to "Include your code in the writeup".** For those questions, you can either copy/paste the code into a `verbatim` environment, or include screenshots of your code.

7. If you have any questions or need clarifications, please post in Slack or visit the TAs during the office hours.

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 Representing the World with Visual Words

1.1 Planar Homographies as a Warp

Recall that a planar homography is an warp operation (which is a mapping from pixel coordinates from one camera frame to another) that 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.

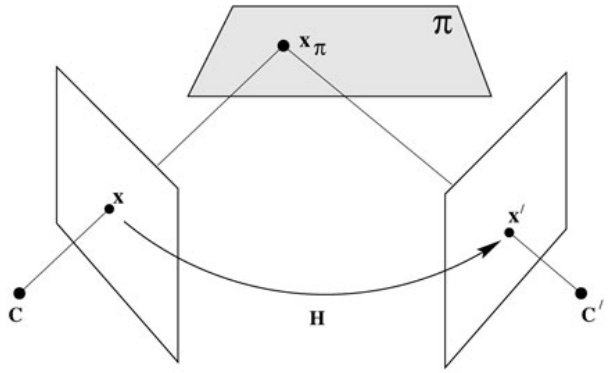

Figure 1: A homography **H** links all points $\mathbf{x}_\pi$ lying in plane π between two camera views **x** and **x**' in cameras C and C_0 respectively such that **x**' = **Hx**. [*From Hartley and Zisserman*]

Q1.1 (5 points): Prove that there exists a homography **H** that satisfies equation 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**.

$$x_1 \equiv H x_2 \tag{1}$$

The $\equiv$ symbol stands for identical to. The points x_1 and x_2 are in *homogenous coordinates*, which means they have an additional dimension. If $\mathbf{x}_1$ 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. You can ignore this special case in your answer. **Answer in your write-up.**

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 **x** and **y** are known vectors, and **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* (which means that the set of equations are of the form $\mathbf{x} = \lambda \mathbf{H}\mathbf{x}'$), which 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 **H** matrix, it's a *transform* of the problem into a set of *linear* equation, thus giving it its name.

Q1.2 (15 points): Correspondences

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 **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 **H** is a 3×3 matrix. For each point pair, this relation can be rewritten as

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

where **h** is a column vector reshaped from **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 **H** from the given point correspondences.

1. How many degrees of freedom does **h** have? (3 points)
2. How many point pairs are required to solve **h**? (2 points)
3. Derive $\mathbf{A_i}$. (5 points)
4. When solving $\mathbf{Ah} = 0$, in essence you're trying to find the **h** that exists in the null space of **A**. What that means is that there would be some non-trivial solution for **h** such that the product **Ah** turns out to be 0. What will be a trivial solution for **h**? Is the matrix **A** full rank? Why/Why not? What impact will it have on the eigen values? What impact will it have on the eigen vectors? (5 points)

1.3 Using Matrix Decompositions to calculate the homography

A homography **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 **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 **V** are the eigenvectors and each corresponding element in **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:

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

Here, U is a matrix of column vectors called the *left singular vectors*. Similarly, V is called the *right singular vectors*. The matrix Σ 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 U are eigenvectors of $\mathbf{A}\mathbf{A}^T$. The columns of V are the eigenvectors of $\mathbf{A^T A}$. 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.

Q1.4.2 (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).

Q1.4.3 (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

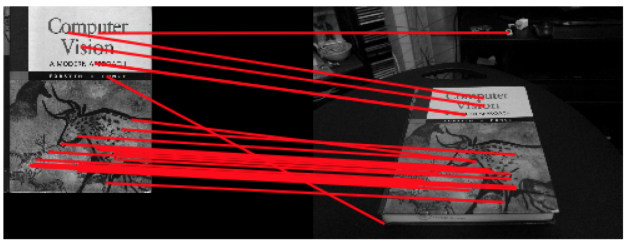

Figure 2: A few matched FAST feature points with the BRIEF descriptor.

one or two sentences.

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.

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

Q2.1.2 (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?

Q2.1.3 (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?

Q2.1.4 (10 points): Feature Matching

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 your write-up. An example is shown in Fig. 2.

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 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. **Please include the code snippet in your writeup.**

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.

Q2.1.6 (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).

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. **Please include the code snippet in your writeup.**

Q2.1.7 Improving Performance - (Extra Credit - 10 points)

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.

1. (**5 pts**) 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). Include the code in your PDF, 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.

2. (**5 pts**) 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 your code and the test images in your PDF. You may simply rescale some of the test images we have given you.

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). **Please include the code snippet in your writeup.**

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. item 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. **Please include the code snippet in your writeup.**

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. **Please include the code snippet in your writeup.**

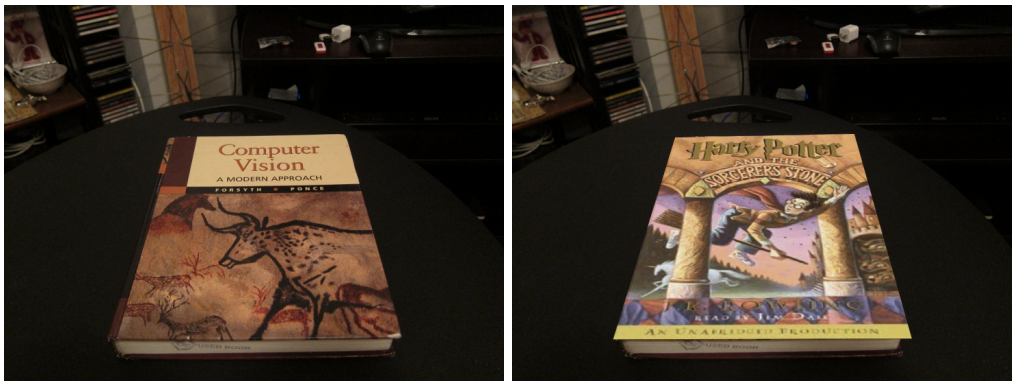

Figure 3: An ordinary textbook (left). Harry-Potterized book (right)

Q2.2.4 (10 points): Automated Homography Estimation and Warping
Write a script `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 3
6. Include your **code** and **result** in your write-up.

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. Include the result images in your writeup, and explain the effect of these two parameters respectively.

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 4 for an example frame of what the final video should look like.

Figure 4: 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 5 for an example.
The two videos have different lengths, you can use the shorter length for your video. Also, the video `book.mov`

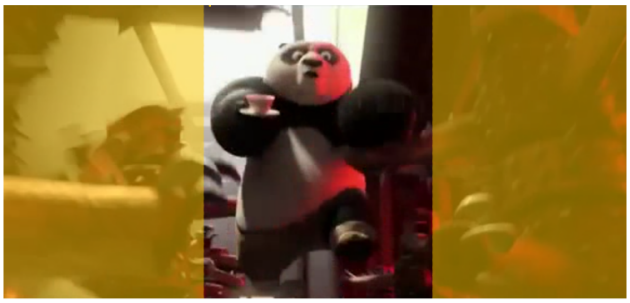

Figure 5: Rendering video on a moving target

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 4 as an example of where the overlay is in the center of the video frame. Please also include the code snippet in your writeup

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.

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. **Make sure to note the achieved fps in your write-up in addition to all the steps taken to achieve real-time performance. Please also include the code snippet in your writeup**

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. Submit the original images, the final panorama, and the script `panorama.py` that loads the images and assembles a panorama. 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. **In your writeup, include your code, original images and the panorama result image.** See Figure 6 below for example.

Figure 6: Original Image 1 (top left). Original Image 2 (top right). Panorama (bottom).

5 HW2 Distribution Checklist

After unpacking `hw2.zip`, you should have a folder `hw2` 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
- `panorama.py`: script to create panorama
- `planarH.py`: script to estimate planar homography

6 HW2 submission checklist

Submit your write-up and code to Gradescope.

- **Writeup.** The write-up should be a pdf file named <**AndrewId**>**_hw2.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`>`_hw2.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:

$$H_{norm} = computeH(x1\_normalized, x2\_normalized)$$
$$H = T_1^{-1} @ H_{norm} @ T_2$$

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 of *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 $\mathbf{H}$ 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 HarryPotterize 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.