3 Planar Homographies Theory Questions Write-Up

3.1 FAST Detector

The FAST Detector (or, Features from Accelerated Segment Test) is a means of detecting corners, useful for feature detection. It works by checking a circular area around the center of a possible corner, and determines 'corner-ness' based on whether a contiguous section of pixels are brighter than the center plus a given contrast threshold, or darker than that center minus a threshold. If given parameters/quotas on the number of pixels to pass the threshold(s) are met, then the center is considered a corner. Harris Corner Detection relies more on gradients— it takes horizontal and vertical derivatives of zones across an image in order to identify regions where both are high, and defines those areas as corners.
Both algorithms serve a similar purpose, but the FAST Detector is more computationally performative (requiring significantly less computation time than Harris) as its rules preclude from expending unnecessary energy on regions certain to not contain corners, while the Harris Detector checks everything. However, it may also miss some points that Harris would not and it is more sensitive to noise, requiring combination with other algorithms for improvement.

3.2 BRIEF Descriptor

The BRIEF Descriptor (or, Binary Robust Independent Elementary Features Descriptor) functions differently from those feature descriptors (MOPS, GIST, SIFT) that we discussed in class because it "directly comput[es] binary strings from image patches", according to the original paper proposing it. That is, instead of relying on the pre-processing of filterbanks as MOPS and GIST do, it does not base 'descriptors' on the response of a bank of filters to a given patch. Instead, it computes bits directly by comparing the intensity of pairs of points, or neighbors to a chosen keypoint, without referencing an existing training database or other library. As a result, it may be quicker than others but more sensitive to noise, and requires an additional layer to link specific points to descriptors.
I would say that any of the filterbanks that we've discussed might be used as a descriptor (e.g. GIST's Gabor filters, MOPS's Haar Wavelets) if we used them in association with further layers of processing, accumulating the responses across an image to a specified bank. However, this might be significantly less efficient than using BRIEF as it requires an additional layer of calculation.

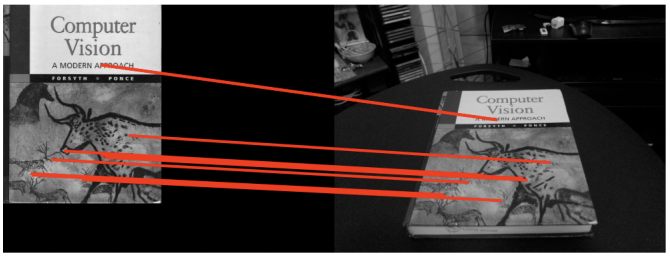
3.3 Matching Methods

Hamming distance is a means of comparing two strings of binary data, and it functions by comparing two strings of equal length and calculating the number of bit positions where the strings differ. It's used in error detection, and may be particularly helpful in the case of BRIEF to quantify similarity between image patches.
Nearest Neighbor is another method for matching data, using a query image patch and a set of reference patches to identify the closest matches. It may be used alongside BRIEF descriptors and Hamming distance to find the reference patch with the 'shortest distance' to the query, and to then establish that correspondence to indicate the best descriptor for that region.
First and foremost, given that our case uses binary descriptors, Hamming distance makes more logical sense given that it acts directly on the binary strings themselves. Furthermore, its efficiency and robustness to minute intensity variance in that process stand in stark contrast to Euclidean distance measurements. Euclidean distance is meant to operate on real-valued descriptors, and so is less-suited to string operations like this. It would have to consider bit-to-bit intensity values, while Hamming distance can simply (and quickly!) count across.
Additionally, Hamming distance is more computational efficient than Euclidean distance, so it improves the performance of the Nearest Neighbors algorithm. Rather than undergoing the complex calculations needed to determine Euclidean distance, Hamming distance is a bit-wise comparison operation and thus is significantly faster.

3.4 Feature Matching

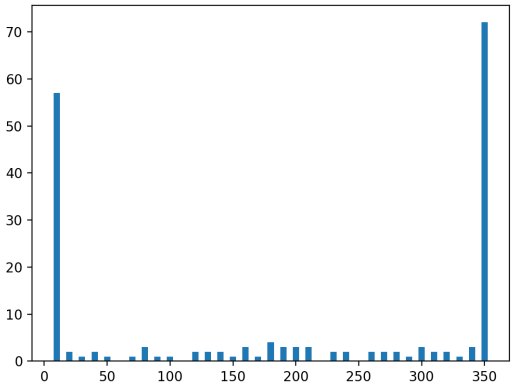
matchPics.py takes in two images and seeks to find distinct matching features between both images. To do this, we first convert the images to grayscale. We then use corner detection on each image to

identify feature points. These points and their respective gray-scaled signals undergo BRIEF Description to locate distinguish features on each, then use the given *briefMatch* to locate features on each image the correspond to each other. The indices of matching pairs that are returned can be used on the other returned values— two arrays of feature locations, one for each image— to note the points that match each other.
We can link matching points by plotting lines between them to see the success of this location. These images use the recommended ratio (0.65) and sigma (0.15). Increasing the ratio and decreasing sigma both generated more lines with less accuracy, but decreasing the ratio and increasing sigma seemed to ignore evident matches, so these values seemed like a sweet spot.

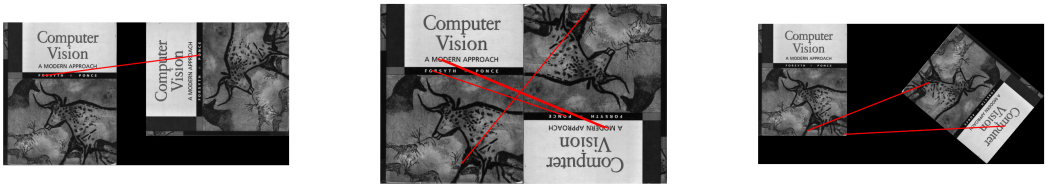


3.5 BRIEF and Rotations

We can further test these matches and the capacity of the functions that built them in *briefRotTest.py*. The results of this function— with calculates matches between the book on the desk and the book cover image at various different rotated points— demonstrate the above-stated known: that BRIEF descriptors are not robust to rotation. We can chart them on a pyplot bar graph as pictured below, with the x-axis corresponding to each degree of rotation and the y corresponding to the number of matches found at that rotation. The graph goes from 10 degrees to 350, so does not depict the maximum number (at angle 0 and angle 360, when no visible rotation has occurred). This decision was made because including those values significantly skewed the scale of the y-axis such that the few matches that the rotated images successfully made were not even visible.



For greater insight into the matches made at differing angles, see the representations below.



Rotated 90 degrees, 180 degrees, and 230 degrees.

3.6 Computing the Homography

computeH calculates the homography by first creating a matrix A of zeros with a row for each x and another for each y (alternating x-y) and nine columns. We then loop through the number of rows and fill them in two-by-two with the necessary vector to calculate x and then, in the following row, y. We use these to compute the singular value decomposition of A and get the smallest value (last column) of V, reshape it to a 3-by-3 homography matrix, and return it.
This matrix, although correctly calculated, is not normalized. So...

3.7 Homography with Normalization

...we use it on the original values after normalizing them such that the points are centered about the origin and the further distance is the square root of two. This function, *computeH_norm* takes in the same values (match locations) for $x1$ and $x2$. We calculate the mean for each and subtract all positions by that translation factor, then divide by the furthest respective distance from the origin and multiply by $\sqrt{2}$ to ensure that the new greatest distance is root-2. These values are used by *computeH* to get the normalized homography matrix. Finally, to denormalize, we use the calculated similarity transforms (built in the following format)...

$$\begin{bmatrix} scale & 0 & -scale * center_x \\ 0 & scale & -scale * center_y \\ 0 & 0 & 1 \end{bmatrix}$$

...within the given equation to calculated the denormalized homography matrix. More specifically, because $x_1 = T_1^{-1} H T_2 x_2$, we can re-structure as follows.

$$x_1 = T_1^{-1} H T_2 x_2$$
$$x_1 = (T_1^{-1} H T_2) x_2$$
$$H' = T_1^{-1} H T_2$$

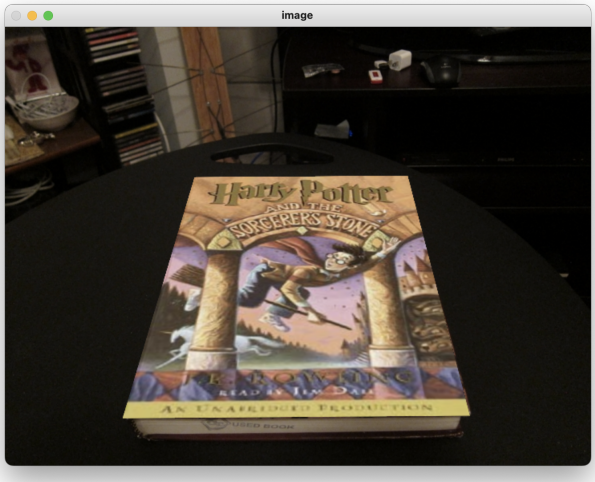
3.8 RANSAC

To find the best possible homography matrix, we then take varying random selections from the given images and calculate homography matrices using RANSAC to find the 'closest' warping of the matrix to get the desired projection. Since, as we calculated above, only four point pairs are needed to calculate the homography, we must randomly select four different points from the set of matches and use them to find the homography. We use $computeH_norm$ to calculate the homography, then check how close the homography is by calculating what *should* be x_1 and seeing how close the homography renders the points to their goal locations. We keep track of the indices of the points that are within a given threshold range of the goal, and if it's larger than the existing greatest count, we update that count and the best matrix. At the end, we recalculate the matrix outside of the loop to tune it to best fit of *all* the data. This ensures the most accurate result possible for the best fit. To ensure that— when normalizing the results— we do not divide by zero, we replace any zero values with ones in the z-level row of the matrix before dividing.
This implementation assumes that there are at least four matches. If an instance occurs where there are fewer than four— as could be brought about by rotation, for example— it is not anticipated within RANSAC. However, we do account for it in other functions that use $computeH_ransac$ like *ar.py* by skipping over images wherein *matchPics* returns a match count of less than four.

3.9 HarryPotterize

HarryPotterize.py takes in the necessary images, then matches features and computes homography from the computer vision book cover ("cv_cover.jpg") to the image of the same book on a desk ("cv_desk.png"). This is then used on the cover of the first Harry Potter book (rescaled to fit the same dimensions as the initial "x2" of the homography) to superimpose it onto the image of the desk. To do that, *compositeH* is used. This makes a matrix to the scale of the desk image, then masks in

the shape of the book cover. The book cover and mask are both warped, and then the mask is used to determine what parts of the resulting image should come from the desk and which should come from the Harry Potter cover.



Results of HarryPotterize.py, superimposing the Sorcerer's Stone cover into the textbook on a desk.

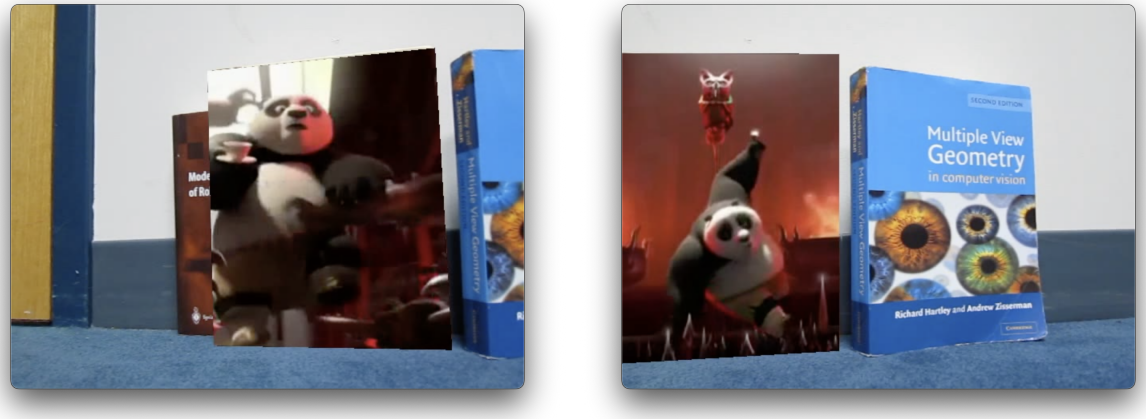
Because the file "hp_cover.jpg" is a smaller scale than the original "cv_cover.jpg", it doesn't 'fit' on top of the book on the desk in the image "cv_desk.png". Evidently, the homography matrix is scale-dependent— to fit another cover to the image, the new cover must have the same dimensions as the one used to create the homography matrix. Therefore, in order to fully cover the book, it stands to reason that "hp_cover.jpg" must have the same dimensions as "cv_cover.jpg". To do this, we can use *cv2.resize* on "hp_cover.jpg" to scale it to fit the picture needed for the homography.

3.10 AR

The augmented reality portion of the assignment was super similar to HarryPotterize— the only differences lay in the following areas:

- *loadVid* returns an array of images, so a loop is required to modify each frame of one video using the corresponding frame of the other.
- To save this to a playable video, we need to use cv2.VideoWriter.
- Because the video to be superimposed has a different aspect ratio from the book, we can't just resize it— we also need to crop it to fit so that warping does not also warp the piece of the video that we see.

The basic flow of the code is as follows: after bringing in the necessary files (the cover of the computer vision book, the frames of the book video, and the frames of the source video). Then we begin writing the video and start looping through the length of the number of frames. For every source frame, we clip off the black edges, rescale to keep the aspect ratio and match the height of the book cover image, and crop once more to isolate an image about the center at the correct width of the book cover.
We then find the matches and homography from book cover to the respective frame and apply the resulting H-matrix to the cropped source image. Along the way, a few checks are necessary— to make sure that enough matches are found to compute RANSAC, and to ensure that we never divide by zero— but, at the end, we can write the final frame to the video. After the loop we release the result and produce an .avi file!



Some screenshots from the video result of ar.py.

3.11 Panorama

I also implemented the extra credit panorama function, but as I had issues importing functions from the python folder to ec, I conducted this part of the assignment in a panorama script within the python folder (as it was handed in). The process was nearly identical to the compositing process, with some small adjustments to how I structured the shape of the image. Given more time, I would've worked through the process of removing any rows and columns with only black pixels to crop to the right scale— however, I did this a bit last minute and did not have time. Still, please enjoy my masterpiece: the DALI lab!