

# 16720B: Computer Vision Homework 4

## Feature Descriptors, Homographies & RANSAC

Instructor: Kris Kitani

TAs: Nadine Chang(Lead), Alireza Golestaneh , Zhengyi Luo

Anand Bhoraskar , Rawal Khirodkar

Due: **Monday, November 2nd, 2020 11:59pm**

Total points: 165, EC: 40

Version: 1 (October 19, 2020)

### Instructions/Hints

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must submit their own work. Include the names of your collaborators in your write up. Code should **NOT** be shared or copied. Please **DO NOT** use external code unless permitted. Plagiarism is prohibited and may lead to failure of this course.
2. **Start early!** This homework will take a long time to complete.
3. **Questions:** If you have any question, please look at piazza first and the FAQ page for this homework.
4. All questions marked with a **Q** require a submission.
5. **For the implementation part, please stick to the headers, variable names, and file conventions provided.**
6. **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.
7. **Do not import external functions/packages other than the ones already imported in the files:** The current imported functions and packages are enough for you to complete this assignment.
8. **Submission:** We have provided a script `check_file.py` which will check if you have all the files needed for submission. The submission is on Gradescope, **you will be submitting both your writeup and code zip file.** The zip file, `<andrew-id.zip>` contains your code and any results files we ask you to save. **Note: You have to submit your writeup separately to Gradescope, and**

**include results in the writeup.** Do **not** submit anything from the `data/` folder in your submission.

9. Assignments that do not follow this submission rule will be **penalized up to 10% of the total score.**
10. Please make sure that the file paths that you use are relative and not absolute.

## Introduction

In this homework, you will implement an interest point (keypoint) detector, a feature descriptor and an image stitching tool based on feature matching and homography.

Interest point (keypoint) detectors find particularly salient points in an image. We can then extract a feature descriptor that helps describe the region around each of the interest points. SIFT, SURF and BRIEF are all examples of commonly used feature descriptors. Once we have extracted the interest points, we can use feature descriptors to match them (find correspondences) between images to do interesting things like panorama stitching or scene reconstruction.

We will implement the BRIEF feature descriptor in this homework. It has a compact representation, is quick to compute, has a discriminative yet easily computed distance metric, and is relatively simple to implement. This allows for real-time computation, as you have seen in class. Most importantly, as you will see, it is also just as powerful as more complex descriptors like SIFT for many cases.

After matching the features that we extract, we will explore the homography between images based on the locations of the matched features. Specifically, we will look at the planar homographies. Why is this useful? In many robotics applications, robots must often deal with tabletops, ground, and walls among other flat planar surfaces. When two cameras observe a plane, there exists a relationship between the captured images. This relationship is defined by a  $3 \times 3$  transformation matrix, called a planar homography. A planar homography allows us to compute how a planar scene would look from a second camera location, given only the first camera image. In fact, we can compute how images of the planes will look from any camera at any location without knowing any internal camera parameters and without actually taking the pictures, all using the planar homography matrix.

## 1 Keypoint Detector (25 pts)

We will implement an interest point detector similar to SIFT. A good reference for its implementation can be found in [4]. Keypoints are found by using the Difference of Gaussian (DoG) detector. This detector finds points that are extrema in both scale and space of a DoG pyramid. This is described in [1], an important paper in computer vision. Here, we will implement a simplified version of the DoG detector described in Section 3 of [4].



Figure 1: Example Gaussian pyramid for `model_chickenbroth.jpg`



Figure 2: Example DoG pyramid for `model_chickenbroth.jpg`

**HINT:** All the functions to implement here are located in `keypointDetect.py`.

**NOTE:** The parameters to use for the following sections are:

$$\sigma_0 = 1, k = \sqrt{2}, \text{levels} = [-1, 0, 1, 2, 3, 4], \theta_c = 0.03 \text{ and } \theta_r = 12.$$

## 1.1 Gaussian Pyramid

In order to create a DoG pyramid, we will first need to create a Gaussian pyramid. Gaussian pyramids are constructed by progressively applying a low pass Gaussian filter to the input image. This function is already provided to you in `keypointDetect.py`.

```
GaussianPyramid = createGaussianPyramid(im, sigma0, k, levels)
```

The function takes as input an image which is going to be converted to grayscale with intensity between 0 and 1 (hint: `cv2.cvtColor(...)`), the scale of the zeroth level of the pyramid `sigma0`, the pyramid factor `k`, and a vector `levels` specifying the levels of the pyramid to construct.

At level  $l$  in the pyramid, the image is smoothed by a Gaussian filter with  $\sigma_l = \sigma_0 k^l$ . The output `GaussianPyramid` is a  $R \times C \times L$  matrix, where  $R \times C$  is the size of the input image `im` and  $L$  is the number of levels. An example of a Gaussian pyramid can be seen in Figure 1. You can visualize this pyramid with the provided function `displayPyramid(pyramid)`.

## 1.2 The DoG Pyramid (5 pts)

The DoG pyramid is obtained by subtracting successive levels of the Gaussian pyramid. Specifically, we want to find:

$$\mathcal{D}_l(x, y, \sigma_l) = (\mathcal{G}(x, y, \sigma_{l-1}) - \mathcal{G}(x, y, \sigma_l)) * \mathcal{I}(x, y) \quad (1)$$

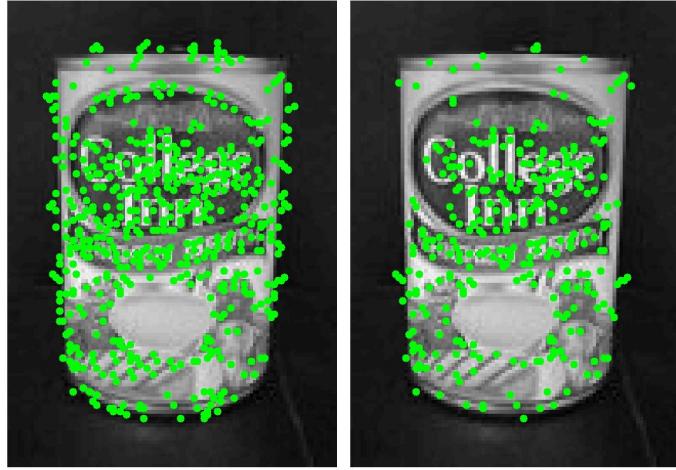


Figure 3.a: Without edge suppression      Figure 3.b: With edge suppression

Figure 3: Interest Point (keypoint) Detection without and with edge suppression for `model_chickenbroth.jpg`

where  $\mathcal{G}(x, y, \sigma_l)$  is the Gaussian filter used at level  $l$  and  $*$  is the convolution operator. Due to the distributive property of convolution, this simplifies to

$$\mathcal{D}_l(x, y, \sigma_l) = \mathcal{G}(x, y, \sigma_{l-1}) * \mathcal{I}(x, y) - \mathcal{G}(x, y, \sigma_l) * \mathcal{I}(x, y) \quad (2)$$

$$= \mathcal{GP}_l - \mathcal{GP}_{l-1} \quad (3)$$

where  $\mathcal{GP}_l$  denotes level  $l$  in the Gaussian pyramid.

**Q 1.2:** Write the following function to construct a Difference of Gaussian pyramid:

```
DoGPyramid, DoGLevels = createDoGPyramid(GaussianPyramid, levels)
```

The function should return `DoGPyramid` an  $R \times C \times (L - 1)$  matrix of the DoG pyramid created by differencing the `GaussianPyramid` input. Note that you will have one level less than the Gaussian Pyramid. `DoGLevels` is an  $(L - 1)$  vector specifying the corresponding levels of the DoG Pyramid (should be the last  $L - 1$  elements of `levels`). An example of the DoG pyramid can be seen in Figure 2.

### 1.3 Edge Suppression (5 pts)

The Difference of Gaussian function responds strongly on corners and edges in addition to blob-like objects. However, edges are not desirable for feature extraction as they are not as distinctive and do not provide a substantially stable localization for keypoints. Here, we will implement the edge removal method described in Section 4.1 of [4], which is based on the principal curvature ratio in a local neighborhood of a point. The paper

presents the observation that edge points will have a “large principal curvature across the edge but a small one in the perpendicular direction.”

**Q 1.3:** The following function is provided to you because of its similar nature of the Harris corner implementation you’ve previously done:

```
PrincipalCurvature = computePrincipalCurvature(DoGPyramid)
```

The function takes in `DoGPyramid` generated in the previous section and returns `PrincipalCurvature`, a matrix of the same size where each point contains the curvature ratio  $R$  for the corresponding point in the DoG pyramid:

$$R = \frac{\text{Tr}(H)^2}{\text{Det}(H)} = \frac{(\lambda_{min} + \lambda_{max})^2}{\lambda_{min}\lambda_{max}} \quad (4)$$

where  $H$  is the Hessian of the Difference of Gaussian function (i.e. one level of the DoG pyramid) computed by using pixel differences as mentioned in Section 4.1 of [4]. (hint: use Sobel filter `cv2.Sobel(...)`).

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix} \quad (5)$$

This is similar in spirit to but *different* than the Harris corner detection matrix [3] you saw in class. In your report, please **explain** the two methods’ similarity and dissimilarity.

We can see that  $R$  reaches its minimum when the two eigenvalues  $\lambda_{min}$  and  $\lambda_{max}$  are equal, meaning that the curvature is the same in the two principal directions. Edge points, in general, will have a principal curvature significantly larger in one direction than the other. To remove edge points, we simply check against a threshold  $R > \theta_r$ . Fig. 3 shows the DoG detector with and without edge suppression.

## 1.4 Detecting Extrema (10 pts)

To detect corner-like, scale-invariant interest points, the DoG detector chooses points that are local extrema in both scale and space. Here, we will consider a point’s eight neighbors in space and its two neighbors in scale (one in the scale above and one in the scale below).

**Q 1.4:** Write the function :

```
locsDoG = getLocalExtrema(DoGPyramid, DoGLevels, PrincipalCurvature,
                           th_contrast, th_r)
```

This function takes as input `DoGPyramid` and `DoGLevels` from Section 1.2 and `PrincipalCurvature` from Section 1.3. It also takes two threshold values, `th_contrast` and `th_r`. The threshold  $\theta_c$  should remove any point that is a local extremum but does not have a Difference of Gaussian (DoG) response magnitude above this threshold (i.e.  $|D(x, y, \sigma)| > \theta_c$ ). The

threshold  $\theta_r$  should remove any edge-like points that have too large a principal curvature ratio specified by `PrincipalCurvature`.

The function should return `locsDoG`, an  $N \times 3$  matrix where the DoG pyramid achieves a local extrema in both scale and space, and also satisfies the two thresholds. The first and second column of `locsDoG` should be the  $(x, y)$  values of the local extremum and the third column should contain the corresponding level of the DoG pyramid where it was detected. (Try to eliminate loops in the function so that it runs efficiently.)

**NOTE:** In all implementations, we assume the *x* coordinate corresponds to columns and *y* coordinate corresponds to rows. For example, the coordinate  $(10, 20)$  corresponds to the (row 21, column 11) in the image.

## 1.5 Putting it together (5 pts)

**Q 1.5:** Write the following function to combine the above parts into a DoG detector:

```
locsDoG, GaussianPyramid = DoGdetector(im, sigma0, k, levels,
                                         th_contrast, th_r)
```

The function should take in a gray scale image, `im`, scaled between 0 and 1, and the parameters `sigma0`, `k`, `levels`, `th_contrast`, and `th_r`. It should use each of the above functions and return the keypoints in `locsDoG` and the Gaussian pyramid in `GaussianPyramid`. Figure 3 shows the keypoints detected for an example image. Note that we are dealing with real images here, so your keypoint detector may find points with high scores that you do not perceive to be corners.

Save the image with the detected keypoints as `results/1_5.jpg` and include it in your report (similar to the one shown in Fig. 3 (b)). You can use any of the provided images.

## 2 BRIEF Descriptor (35 pts)

Now that we have interest points that tell us where to find the most informative feature points in the image, we can compute descriptors that can be used to match to other views of the same point in different images. The BRIEF descriptor encodes information from a  $9 \times 9$  patch  $p$  centered around the interest point at the *characteristic scale* of the interest point. See the lecture notes to refresh your memory.

**HINT:** All the functions to implement here are located in `BRIEF.py`.

### 2.1 Creating a Set of BRIEF Tests (5 pts)

The descriptor itself is a vector that is  $n$ -bits long, where each bit is the result of the following simple test:

$$\tau(\mathbf{P}; \mathbf{x}, \mathbf{y}) := \begin{cases} 1, & \text{if } \mathbf{P}[\mathbf{x}] < \mathbf{P}[\mathbf{y}]. \\ 0, & \text{otherwise.} \end{cases} \quad (6)$$

Set  $n$  to 256 bits ( $\mathbb{R}^{256}$ ). Thus there are 256  $\tau$ . There is no need to encode the test results as actual bits. It is fine to encode them as a 256 element vector.

There are many choices for the 256 test pairs  $\{\mathbf{x}, \mathbf{y}\}$  (remember  $\mathbf{x}$  and  $\mathbf{y}$  are 2D vectors relating to discrete 2D coordinates within the 2D image patch matrix  $\mathbf{P}$ ) used to compute  $\tau(\mathbf{P}; \mathbf{x}, \mathbf{y})$  (each of the  $n$  bits). The authors describe and test some of them in [2]. Read Section 3.2 of that paper and implement one of these solutions. You should generate a static set of test pairs and save that data to a file. You will use these pairs for all subsequent computations of the BRIEF descriptor.

**Q 2.1:** Write the function to create the  $x$  and  $y$  pairs that we will use for comparison to compute  $\tau$ :

```
compareX, compareY = makeTestPattern(patchWidth, nbBits)
```

`patchWidth` is the width of the image patch (usually 9) and `nbBits` is the number of tests  $n$  in the BRIEF descriptor. `compareX` and `compareY` are linear indices into the  $patchWidth \times patchWidth$  image patch and are each  $nbBits \times 1$  vectors. Run this routine for the given parameters `patchWidth = 9` and  $n = 256$  and save the results in `testPattern.npy`. **Include this file** in your submission.

## 2.2 Compute the BRIEF Descriptor (10 pts)

Now we can compute the BRIEF descriptor for the detected keypoints.

**Q 2.2:** Write the function:

```
locs, desc = computeBrief(im, GaussianPyramid, locsDoG, k, levels,
                           compareX, compareY)
```

Where `im` is a grayscale image with values from 0 to 1, `locsDoG` are the keypoint locations returned by the DoG detector from Section 1.5, `levels` are the Gaussian scale levels that were given in Section 1, and `compareX` and `compareY` are the test patterns computed in Section 2.1 and were saved into `testPattern.npy`.

The function returns `locs`, an  $m \times 3$  vector, where the first two columns are the image coordinates of keypoints and the third column is the pyramid level of the keypoints, and `desc` is an  $m \times n$  bits matrix of stacked BRIEF descriptors.  $m$  is the number of valid descriptors in the image and will vary. You may have to be careful about the input DoG detector locations since they may be at the edge of an image where we cannot extract a full patch of width `patchWidth`. Thus, the number of output `locs` may be less than the input `locsDoG`. **Note:** Its possible that you may not require all the arguments to this function to compute the desired output. They have just been provided to permit the use of any of some different approaches to solve this problem.

## 2.3 Putting it all Together (5 pts)

**Q 2.3:** Write a function :

```
locs, desc = briefLite(im)
```

which accepts a grayscale image `im` with values between 0 and 1 and returns `locs`, an  $m \times 3$  vector, where the first two columns are the image coordinates of keypoints and the third column is the pyramid level of the keypoints, and `desc`, an  $m \times n$  bits matrix of stacked BRIEF descriptors.  $m$  is the number of valid descriptors in the image and will vary.  $n$  is the number of bits for the BRIEF descriptor.

This function should perform all the necessary steps to extract the descriptors from the image, including

- Compute DoG pyramid.
- Get keypoint locations.
- Compute a set of valid BRIEF descriptors.

## 2.4 Check Point: Descriptor Matching (5 pts)

A descriptor's strength is in its ability to match to other descriptors generated by the same world point despite change of view, lighting, etc. The distance metric used to compute the similarity between two descriptors is critical. For BRIEF, this distance metric is the Hamming distance. The Hamming distance is simply the number of bits in two descriptors that differ. (Note that the position of the bits matters.)

To perform the descriptor matching mentioned above, we have provided the function in `BRIEF.py`:

```
matches = briefMatch(desc1, desc2, ratio)
```

Which accepts an  $m_1 \times n$  bits stack of BRIEF descriptors from a first image and a  $m_2 \times n$  bits stack of BRIEF descriptors from a second image and returns a  $p \times 2$  matrix of matches, where the first column are indices into `desc1` and the second column are indices into `desc2`. Note that  $m_1$ ,  $m_2$ , and  $p$  may be different sizes and  $p \leq \min(m_1, m_2)$ .

**Q 2.4:** Write a test script or utilize the code provided in the `main` function of `BRIEF.py` to load two of the `chickenbroth` images and compute feature matches. Use the provided `plotMatches` and `briefMatch` functions to visualize the result.

```
plotMatches(im1, im2, matches, locs1, locs2)
```

where `im1` and `im2` are two colored images stored as `uint8`, `matches` is the list of matches returned by `briefMatch` and `locs1` and `locs2` are the locations of keypoints from `briefLite`.

**Save the resulting figure** as `results/2_4.jpg` and submit it in your PDF report. Also, present results with the two `incline` images and with the computer vision textbook cover page (template is in file `pf_scan_scaled.jpg`) against the other `pf_*` images. Briefly discuss any cases that perform worse or better. See Figure 4 for an example result.

Suggestion for debugging: A good test of your code is to check that you can match an image to itself.

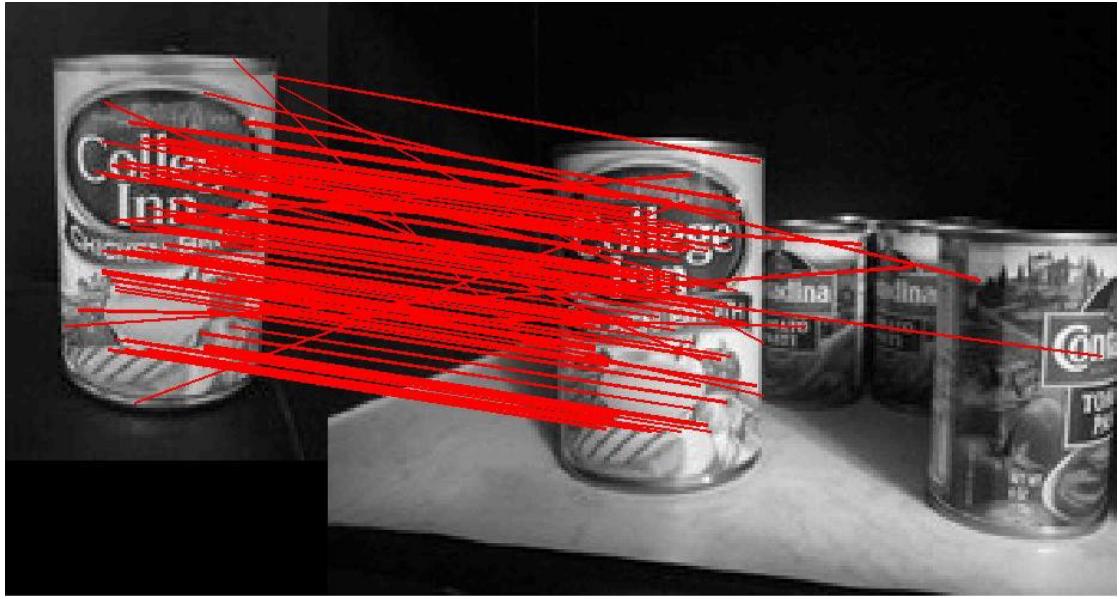


Figure 4: Example of BRIEF matches for `model_chickenbroth.jpg` and `chickenbroth_01.jpg`.

## 2.5 BRIEF and rotations (10 pts)

You may have noticed worse performance under rotations. Let's investigate this!

**Q 2.5:** Take the `model_chickenbroth.jpg` test image and match it to itself while rotating the second image (hint: `cv2.getRotationMatrix2D(...)`, `cv2.warpAffine(...)`) in increments of 10 degrees. Count the number of matches at each rotation and construct a bar graph showing rotation angle vs the number of correct matches. Include this in your PDF and explain why you think the descriptor behaves this way. **Write your code in `briefRotTest.py`.**

## 3 Planar Homographies: Theory (30 pts)

Suppose we have two cameras looking at a common plane in 3D space. Any 3D point  $\mathbf{w}$  on this plane generates a projected 2D point located at  $\tilde{\mathbf{u}} = [u_1, v_1, 1]^T$  on the first camera and  $\tilde{\mathbf{x}} = [x_2, y_2, 1]^T$  on the second camera. Since  $\mathbf{w}$  is confined to a plane, we expect that there is a relationship between  $\tilde{\mathbf{u}}$  and  $\tilde{\mathbf{x}}$ . In particular, there exists a common  $3 \times 3$  matrix  $\mathbf{H}$ , so that for any  $\mathbf{w}$ , the following condition holds:

$$\lambda \tilde{\mathbf{x}} = \mathbf{H} \tilde{\mathbf{u}} \quad (7)$$

where  $\lambda$  is an arbitrary scalar weighting. We call this relationship a *planar homography*. Recall that the  $\sim$  operator implies a vector is employing homogenous coordinates such

that  $\tilde{\mathbf{x}} = [\mathbf{x}, 1]^T$ . It turns out this homography relationship is also true for cameras that are related by pure rotation without the planar constraint.

**Q 3.1** (20 pts) We have a set of  $N$  2D homogeneous coordinates  $\{\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_N\}$  taken at one camera view, and  $\{\tilde{\mathbf{u}}_1, \dots, \tilde{\mathbf{u}}_N\}$  taken at another. Suppose we know there exists an unknown homography  $\mathbf{H}$  between the two views such that,

$$\lambda_n \tilde{\mathbf{x}}_n = \mathbf{H} \tilde{\mathbf{u}}_n, \quad \text{for } n = 1 : N \quad (8)$$

where again  $\lambda_n$  is an arbitrary scalar weighting.

- (a) Given the  $N$  correspondences across the two views and using Equation 8, derive a set of  $2N$  independent linear equations in the form:

$$\mathbf{A}\mathbf{h} = \mathbf{0} \quad (9)$$

where  $\mathbf{h}$  is a vector of the elements of  $\mathbf{H}$  and  $\mathbf{A}$  is a matrix composed of elements derived from the point coordinates. Write out an expression for  $\mathbf{A}$ .

*Hint:* Start by writing out Equation 8 in terms of the elements of  $\mathbf{H}$  and the homogeneous coordinates for  $\tilde{\mathbf{u}}_n$  and  $\tilde{\mathbf{x}}_n$ .

- (b) How many elements are there in  $\mathbf{h}$ ?

- (c) How many point pairs (correspondences) are required to solve this system?

*Hint:* How many degrees of freedom are in  $\mathbf{H}$ ? How much information does each point correspondence give?

- (d) Show how to estimate the elements in  $\mathbf{h}$  to find a solution to minimize this homogeneous linear least squares system. Step us through this procedure.

*Hint:* Use the Rayleigh quotient theorem (homogeneous least squares).

### **Q 3.2** Understanding homographies under rotation (5 points)

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

### **Q 3.3** Limitations of the planar homography (2 points)

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.

### **Q 3.4** Behavior of lines under perspective projections (3 points)

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  $P$  in  $\mathbf{x} = P\mathbf{X}$  preserves lines.

## 4 Planar Homographies: Implementation (10 pts)

**Note:** Implement the method in `planarH.py`.

Now that we have derived how to find  $\mathbf{H}$  mathematically in **Q 3.1**, we will implement it in this section.

**Q 4.1 (10pts)** Implement the function

```
H2to1 = computeH(X1,X2)
```

Inputs:  $\mathbf{X}_1$  and  $\mathbf{X}_2$  should be  $2 \times N$  matrices of corresponding  $(x, y)^T$  coordinates between two images.

Outputs:  $\mathbf{H2to1}$  should be a  $3 \times 3$  matrix encoding the homography that best matches the linear equation derived above for Equation 8 (in the least squares sense). *Hint:* Remember that a homography is only determined up to scale. The `numpy.linalg` function `eigh()` or `svd()` will be useful. Note that this function can be written without an explicit for-loop over the data points.

## 5 RANSAC (15 pts)

**Note:** Implement the method in `planarH.py`.

The least squares method you implemented for computing homographies is not robust to outliers. If all the correspondences are good matches, this is not a problem. But even a single false correspondence can completely throw off the homography estimation. When correspondences are determined automatically (using BRIEF feature matching for instance), some mismatches in a set of point correspondences are almost certain. RANSAC (Random Sample Consensus) can be used to fit models robustly in the presence of outliers.

**Q 5.1 (15pts):** Write a function that uses RANSAC to compute homographies automatically between two images:

```
bestH = ransacH(matches, locs1, locs2, nIter, tol)
```

The inputs and output of this function should be as follows:

- Inputs: `locs1` and `locs2` are matrices specifying point locations in each of the images and `matches` is a matrix specifying matches between these two sets of point locations. These matrices are formatted identically to the output of the provided `briefMatch` function.
- Algorithm Input Parameters: `nIter` is the number of iterations to run RANSAC for, `tol` is the tolerance value for considering a point to be an inlier. Define your function so that these two parameters have reasonable default values.
- Outputs: `bestH` should be the homography model with the most inliers found during RANSAC.

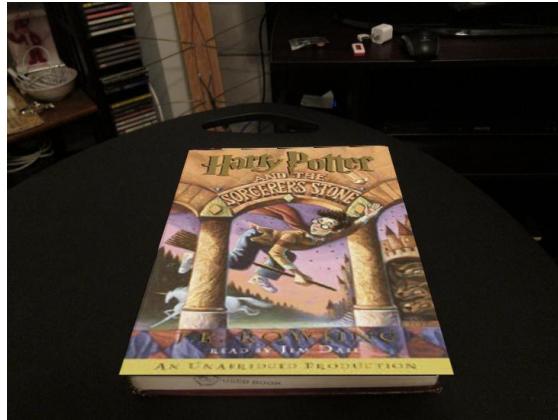


Figure 5: Harry potter cover warped to cv text book cover. This result is reproduced with ransac max iter 20000 and tol 3.

## 6 Automated Homography Estimation and Warping (20 pts)

**Q 6.1 Putting it together (10 points):** Write a script HarryPotterize.py that

1. Reads pf\_desk.jpg, pf\_scan\_scaled.jpg, and hp\_cover.jpg.
2. Computes a homography automatically using briefLite, matchBrief, and ransacH.
3. Warps hp\_cover.jpg to the dimensions of the pf\_desk.jpg 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 in planarH.py:

```
composite img = compositeH(H, template, img)
```

to now compose this warped image with the desk image as in in Figure 5.

6. **Save your image** as results/6\_1.jpg and include your resulting image in your report. Please also print the final H matrix in your report.

**Q 6.2 RANSAC Parameter Tuning (10 points)** There are two tunable parameters in RANSAC which will we be exploring. 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. What is a good selection for the number of random samples and why?



Figure  
incline\_L.jpg (img1)

6.a: Figure  
incline\_R.jpg (img2)

6.b: Figure  
6.c: img2 warped to  
img1's frame

Figure 6: Example output for **Q 7.1**: Original images `img1` and `img2` (left and center) and `img2` warped to fit `img1` (right). Notice that the warped image clips out of the image. We will fix this in **Q 7.2**

## 7 Stitching it together: Panoramas (30 pts)

**NOTE:** All the functions to implement here are in `panoramas.py`.

We can also use homographies to create a panorama image from multiple views of the same scene. This is possible for example when there is no camera translation between the views (e.g., only rotation about the camera center). First, you will generate panoramas using matched point correspondences between images using the BRIEF matching in Section 2.4. **We will assume that there is no error in your matched point correspondences between images (Although there might be some errors, and even small errors can have drastic impacts).** In the next section you will extend the technique to deal with the noisy keypoint matches.

You will need to use the perspective warping function from OpenCV: `warp_im = cv2.warpPerspective(im, H, out_size)`, which warps image `im` using the homography transform `H`. The pixels in `warp_im` are sampled at coordinates in the rectangle  $(0, 0)$  to  $(\text{out\_size}[0]-1, \text{out\_size}[1]-1)$ . The coordinates of the pixels in the source image are taken to be  $(0, 0)$  to  $(\text{im.shape}[1]-1, \text{im.shape}[0]-1)$  and transformed according to `H`. To understand this function, you may review Homework 0.

**Q 7.1 (15pts)** In this problem you will complete the **partially implemented** function:

```
panoImg = imageStitching(img1, img2, H2to1)
```

on two images from the Dusquesne incline. Please fill in the TODOs in the script. This function accepts two images and the output from the homography estimation function. This function:

- (a) Warps `img2` into `img1`'s reference frame using the aforementioned perspective warping function
- (b) Blends `img1` and warped `img2` and outputs the panorama image.

For this problem, use the provided images `incline_L` as `img1` and `incline_R` as `img2`. The point correspondences `pts` are generated by your BRIEF descriptor matching.

Apply your `ransacH()` to these correspondences to compute `H2to1`, which is the homography from `incline_R` onto `incline_L`. Then apply this homography to `incline_R` using `warpH()`.

**Note:** Since the warped image will be translated to the right, you will need a larger target image.

Visualize the warped image and save this figure as `results/7_1.jpg` using OpenCV's `cv2.imwrite()` function and save *only* `H2to1` as `results/q7_1.npy` using Numpy `np.save()` function.

**Q 7.2 (15pts)** Notice how the output from Q 7.1 is clipped at the edges? We will fix this now. A **partially implemented** function is provided:

```
[panoImg] = imageStitching_noClip(img1, img2, H2to1)
```

that takes in the same input types and produces the same outputs as in Q 7.1.

Please fill in the TODOs in the script.

To prevent clipping at the edges, we instead need to warp *both* image 1 and image 2 into a common third reference frame in which we can display both images without any clipping. Specifically, we want to find a matrix  $M$  that *only* does scaling and translation such that:

```
warp_im1 = cv2.warpPerspective(im1, M, out_size)
warp_im2 = cv2.warpPerspective(im2, np.matmul(M,H2to1), out_size)
```

This produces warped images in a common reference frame where all points in `im1` and `im2` are visible. To do this, we will only take as input either the width or height of `out_size` and compute the other one based on the given images such that the warped images are not squeezed or elongated in the panorama image. For now, assume we only take as input the width of the image (i.e., `out_size[0]`) and should therefore compute the correct height(i.e., `out_size[1]`).

*Hint:* The computation will be done in terms of `H2to1` and the extreme points (corners) of the two images. Make sure  $M$  includes only scale (find the aspect ratio of the full-sized panorama image) and translation.

Test your method by passing `incline_L` as `img1` and `incline_R` as `img2`.

**Q 7.3 (0pts):** This function has been provided to you. You now have all the tools you need to automatically generate panoramas. This function accepts two images as input, computes keypoints and descriptors for both the images, finds putative feature correspondences by matching keypoint descriptors, estimates a homography using RANSAC and then warps one of the images with the homography so that they are aligned and then overlays them.

```
im3 = generatePanorama(im1, im2)
```

Run your code on the image pairs `{data/incline_L.jpg, data/incline_R.jpg}` and `{data/hi_L.jpg, data/hi_R.jpg}`. However during debugging, try on scaled down versions of the images to keep running time low. Save the resulting panorama on the full sized images as `results/q7_3.jpg`. (see Figure 7 for example output). Include the figure in your writeup.



Figure 7: Final panorama view. With homography estimated using RANSAC.

## 8 Extra Credit (40 pts)

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. Host an online study group with at least 4 students. Up to 2 study groups are counted for EC for each assignment. Each hosting is worth 5 points. (10 pts max)
2. You have attended Ce Liu’s VASC seminar or Tali Dekel’s AI seminar. Please provide a half page report on what you learned from the seminar. (10 pts)
3. As we have seen, BRIEF is not rotationally invariant. Design a simple fix to solve this problem using the tools you have developed so far. Explain in your PDF 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. Submit your code as `ec_1.py`. (10pts)
4. 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 [4] 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. You may simply rescale some of the test images we have given you. Submit your code as `ec_2.py`. (10pts)

For each question, plot a graph of the number of correct matches, changing rotation from 0 to 180 degrees and scale from  $1x$  to  $20x$  (or some other set of appropriate scale values) and analyze your results in your report.

## References

- [1] P. Burt and E. Adelson. The Laplacian Pyramid as a Compact Image Code. *IEEE Transactions on Communications*, 31(4):532–540, April 1983.
- [2] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. BRIEF : Binary Robust Independent Elementary Features.
- [3] Christopher G Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.
- [4] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004.