16720 Computer Vision: Homework 3
BRIEF Feature Descriptors

Instructor: Martial Hebert
TAs: Xinlei Chen, Arne Suppé, Jacob Walker, Feng Zhou
Due Date: October 17^{th}, 2013, 11:59 PM

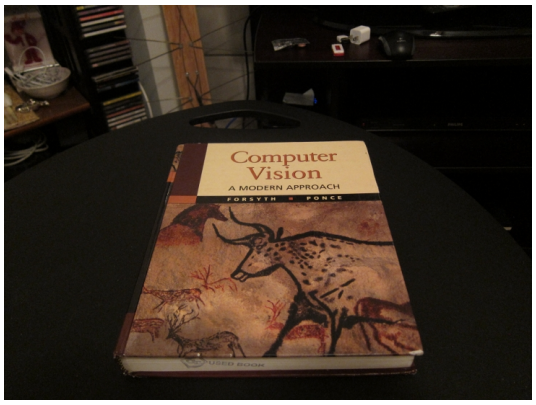

Figure 1: Your computer vision text book.

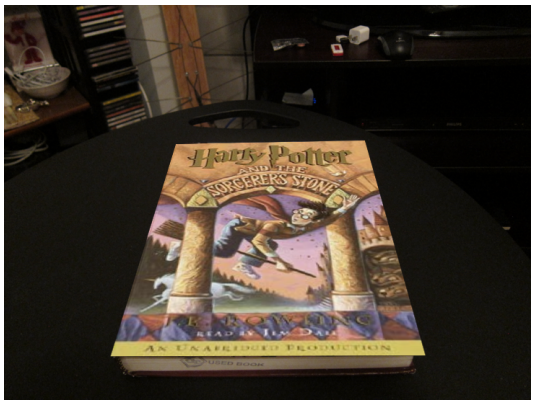

Figure 2: Your textbook HarryPotterized.

In this homework, you will implement an interest point detector and feature descriptor. Interest point detectors find particularly salient points in an image upon which we can extract a feature descriptor. SIFT [6], SURF [1], and BRIEF [3] are all examples of descriptors. In our case, we will be using BRIEF. Once we have extracted the interest points, we can use descriptors to match them between images to do neat things like panorama stitching or scene reconstruction.

BRIEF is one the simplest feature descriptors to implement. It has a very compact representation, is quick to compute, and has a discriminative yet easily computed distance metric. This allows for real-time computation, as you have seen in class.[1] Most importantly, as you will see, it is also just as powerful as more complex descriptors like SIFT, for many cases.

[1] Your MATLAB implementation will probably not be real time. Don't worry about this.

1 Keypoint Detector

For our implementation, we will be using the detector similar to the one introduced in class. A good reference for its implementation can be found in [6]. 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 [2], an important paper in our field.

Here, we will be implementing a simplified version of the DoG detector described in Section 3 of [6]. The parameters you will use for the following sections are $\sigma_0 = 1$, $k = \sqrt{2}$, $levels = \begin{bmatrix} -1 & 0 & 1 & 2 & 3 & 4 \end{bmatrix}$, $th_{contrast} = 0.03$, and $th_r = 12$.

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 lowpass filter to the input image. We provide you this function.

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

The function takes as input a grayscale image `im` with values between 0 and 1 (hint: `im2double`), 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 $\sigma_l = \sigma_0 k^l$. `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 size of `levels`. An example of a Gaussian pyramid can be seen in Figure 3.

Figure 3: Example Gaussian pyramid for `model_chickenbroth.jpg`

1.1.1 The DoG Pyramid

The DoG pyramid is obtained by subtracting successive levels of the Gaussian pyramid.

$$D_l(x, y, \sigma_l) = (G(x, y, \sigma_l) - G(x, y, \sigma_{l-1})) * I(x, y)$$

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

```
[DoGPyramid, DoG_levels] = createDoGPyramid(GaussianPyramid, levels)
```

The function should return `DoGPyramid`, a $R \times C \times (L-1)$ matrix of the DoG pyramid created using `GaussianPyramid`. Note that you will have one less level than the Gaussian pyramid. `DoG_levels` is an $(L-1)$ vector specifying the corresponding levels of the DoG pyramid. An example of the DoG pyramid can be seen in Figure 4.

Figure 4: Example DoG pyramid for `model_chickenbroth.jpg`

1.1.2 Edge Suppression (10 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 distinctive and do not provide stable localization of the keypoint. Here, we will implement the edge removal method described in Section 4.1 of [6], which is based on the principal curvature ratio in a local neighborhood of a point. The paper makes the observation that edge points will have a "large principal curvature across the edge but a small one in the perpendicular direction."

Implement the following function:

```
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:

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

Here, `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 [6]. (hint: Matlab function gradient). This is similar in spirit but different from the Harris matrix you saw in class. Both methods examine the eigen values of a matrix, but the method in [6] performs a test that does not require direct computation of the values.

We can see that `R` reaches its minimum when the two eigenvalues λ_{min} and λ_{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. Thus, to remove edge points, we can check that $R < thr$ for some thr. Figure 5 shows the DoG detector with and without edge suppression.

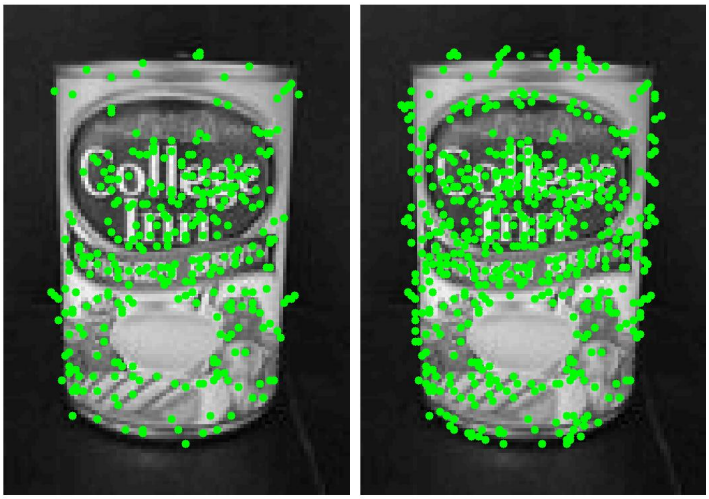

Figure 5: Left, keypoints with edge suppression, right, without for `model_chickenbroth.jpg`

1.1.3 Detecting Extrema (10 pts)

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

```
locs = getLocalExtrema(DoGPyramid, DoG_levels, PrincipalCurvature,
                       th_contrast, th_r)
```

This function takes as input `DogPyramid` and `DoG_levels` from Section 1.1.1 and `PrincipalCurvature` from Section 1.1.2. It also takes two threshold values, `th_contrast` and `th_r`. The threshold `th_contrast` should remove any point that is a local extremum but does not have a Difference of Gaussian response magnitude above this threshold (i.e. $|D(x, y, \sigma)| > th_{contrast}$). The threshold `th_r` should remove any edge-like points that have too large a principal curvature ratio specified by `PrincipalCurvature`.

The function should return `locs`, 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 `locs` 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.)

1.2 Putting it together (5 pts)

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

```
[locs, 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 call each of the above functions and return the keypoints in `locs` and the Gaussian pyramid in `GaussianPyramid`. Figure 5 left 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.

2 BRIEF Descriptor

Now that we have keypoints that tell us where to find the most informative 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 9×9 patch p centered around the interest point *at the characteristic scale of the interest point*. See the lecture notes for Point Feature Detectors if you need to refresh your memory.

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:

$$\tau(p; x, y) := \begin{cases} 1 & p(x) < p(y) \\ 0 & otherwise \end{cases}$$

Set n to 256 bits. 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. The authors describe and test some of them in [4]. 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. Please include this file in your submission so that we do not need to regenerate your test pattern.

Write the function:

```
[compareX, compareY] = makeTestPattern(patchWidth, nbits)
```

Where `patchWidth` is the width of the image patch (usually 9) and `nbits` is the number of tests in the BRIEF descriptor. `compareX` and `compareY` are linear indices into the $patchWidth \times patchWidth$ image patch and are each $nbits \times 1$ vectors. Run this routine for the given parameters and save the results in `testPattern.mat`.

2.2 Compute the BRIEF Descriptor (10 pts)

It is now time to compute the BRIEF descriptor for the detected keypoints. Write the function:

```
[locs,desc] = computeBrief(im, locs, levels, compareX, compareY)
```

Where `im` is grayscale image with values from 0 to 1, `locs` are the keypoint locations returned by the DoG detector from Section 1.2, `levels` are the scale levels that were given in Section 1, and `compareX` and `compareY` are the test patterns computed in Section 2.1.

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`, an $m \times nbits$ matrix of stacked BRIEF descriptors. m is the number of valid descriptors in the image and will vary.

2.3 Putting it together (5 pts)

Write a function:

```
[locs, desc] = brief(im)
```

Which accepts a grayscale image `im` with values between zero and one 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 $m \times nbits$ matrix of stacked BRIEF descriptors. m is the number of valid descriptors in the image and will vary.

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

- Load parameters and test patterns
- Get keypoints locations
- Compute a set of valid BRIEF descriptors

2.4 Descriptor Matching (10 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 percentage of bits in two descriptors that differ. (Note that the position of the bits matters.) Write the function

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

Which accepts an $m \times nbits$ stack of BRIEF descriptors from a first image and a $n \times nbits$ 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, n, and p may be different sizes.

Section 7.2 of [6] introduces a ratio test to suppress matches between descriptors that are not particularly discriminative. Implement this test and select a value for `ratio`. In your PDF, mention how you designed this test and how you selected `ratio`.

Write a test script `testMatch` to load two of the chickenbroth images, compute feature matches, and use the provided `plotMatches` function to visualize the result.

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

Where `im1` and `im2` are grayscale images from 0 to 1, `matches` is the list of matches returned by `briefMatch` and `locs1` and `locs2` are the locations of keypoints from `brief`.

Save the resulting figure and submit it in your PDF. Figure 6 is an example result. A good test is to check that you can match an image to itself.

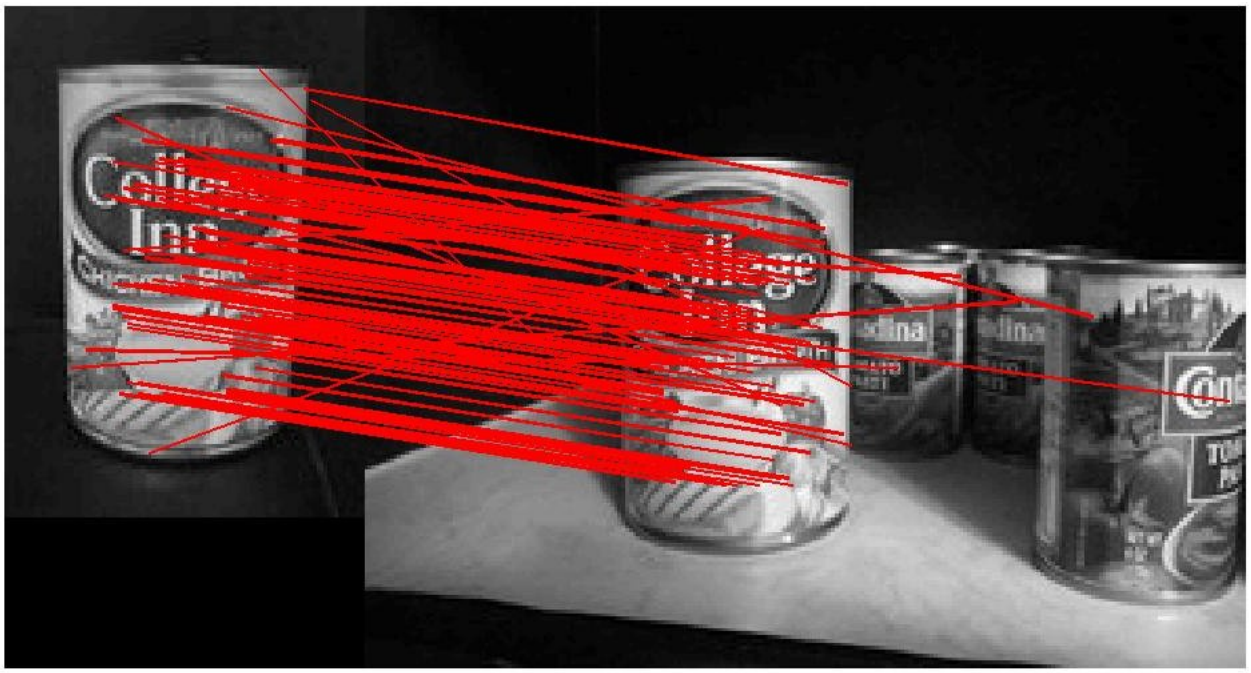

Figure 6: Example of BRIEF matches for `model_chickenbroth.jpg` and `chickenbroth_01.jpg`.

2.5 BRIEF and Rotations (10 pts)

Take one of the test images and match it to itself while rotating the second image in increments of 10 degrees. Count the number of correct matches at each rotation and construct a histogram. Include this in your PDF and explain why you think the descriptor behaves this way. Create a script `briefRotTest` that performs this task.

3 Fun With Homography

Suppose you have a picture with the computer vision text book and you are so thoroughly disgusted with this book that you want to replace it with your favorite Harry Potter

novel's cover. (Figures 1 2) You now *almost* have all the tools necessary to do this automatically. If you can compute the planar homography between the computer vision textbook in a known pose to a picture of the book in an unknown pose using BRIEF features matches, you should be able to warp an image of Harry Potter (in the same, known pose) onto the image you wish to modify. The problem is that outliers will make your last homework's least-squares solution unstable. Here is where RANSAC comes in.

3.1 RANSAC (10 pts)

The RANSAC algorithm [5] can generally to fit any model to data. You will implement it for (planar) homographies between images. Remember that 4 point-pairs are required at a minimum to compute a homography.

The function should have the following fingerprint:

`[bestH2to1, bestError, inliers]=ransacH2to1(matches, locs1, locs2).`

`bestH2to1` should be the homography with least error found during RANSAC, along with this error `bestError`. `H2to1` will be a homography such that if p_2 is a point in `locs2` and p_1 is a corresponding point in `locs1`, then $p_1 \equiv Hp_2$.

`locs1` and `locs2` are produced by `brief`. `matches` is the same matches as produced by `briefMatch`.

`inliers` is a vector of length `size(matches,1)` with a 1 at those matches that are part of the consensus set, and 0 elsewhere. We have provided `computeH_norm` to compute the homography, but feel free to use your own implementation from Homework 1.

3.2 Putting it together (5 pts)

Write a script `HarryPotterize` that

1. Reads `harrypotter.jpg`, `pf_scan_scaled.jpg` and `pf_desk.jpg`.
2. Computes a homography that maps pixels on `pf_scan_scaled.jpg` to `pf_desk.jpg` using matched features generated by your BRIEF implementation and your RANSAC implementation.
3. Warps `harrypotter.jpg` onto the cover of the book in `pf_scan_scaled.jpg`, as in Figure 2 using the provided `warpH` function.

Hand in your version of Figure 2 for each test image as a JPG and include it in your PDF. Use `plotMatches` to show the inliers that RANSAC returned. Include this figure in your PDF. Perform the same task, substituting `pf_stand.jpg` and `pf_floor.jpg` for `pf_scan_scaled.jpg`.

3.3 Theory

Suppose you wished to recover the pose of the camera from the homography matrix. Assume there is a plane with a known equation in 3D space.

1. **(5 pts)** Write the relation between P_i and the imaged points in our target image, $(u_i, v_i, 1)^T$, in terms of the camera matrix K, and the camera rotation R and translation t. Use homogeneous coordinates and projective equivalence.

2. **(5 pts)** Write an expression for the columns of a matrix A such that $H \equiv KA$. This should be in terms of the rows and/or columns of R and t and the plane equation.

3. **(10 pts)** Devise a way to recover R and t from H given K and the plane equation. There may be more than one solution that satisfies the relationship. Explain the ambiguities and how they arise.

4 Extra Credit

1. 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.

2. We will provide on Blackboard two video sequences of the computer vision book. One sequence contains just panning and tilting (`bookPan.mp4`), the other contains large rotations (`bookRot.mp4`). Use your tools to replace the cover of the book in all the frames to produce a new video. You may want to consider ways to reduce the space that RANSAC searches for matches based on knowledge you have of where the book is in prior frames. Explain your design decisions and parameter selections in your PDF and provide documented code for us to run.

5 Files in this distribution

- `model_chickenbroth.jpg`: An image of a can.
- `chickenbroth_0x.jpg`: More cans to match against for fun. Feel free to use this in your PDF.
- `plotMatches.m`: A Matlab function to draw lines between matched features.
- `harrypotter.jpg`: A rectified Harry Potter book cover.
- `pf_scan_scaled.jpg`: A rectified Forsyth and Ponce book cover.
- `pf_desk.jpg`: The Ponce and Forsyth book on a desk.
- `pf_stand.jpg`: The Ponce and Forsyth book standing up.
- `pf_floor.jpg`: The Ponce and Forsyth book on the floor.

- `pf_floor_rot.jpg`: The Ponce and Forsyth book on the floor, rotated. Use this for extra credit problems.
- `pf_pile.jpg`: The Ponce and Forsyth book on the floor, rotated in a pile. Use this for extra credit problems.
- `computeH.m`: An un-normalized least-squares homography estimator.
- `computeH_norm.m`: A normalized least-squares homography estimator.
- `normCoord.m`: A coordinate normalization routine
- `warpH.m`: An image warping routine.
- `createGaussianPyramid.m`: Generates a Gaussian pyramid of an image.

6 What to Submit

You should hand in code so that we have a functional version of your system. Comment your code and provide any detailed explanations in your PDF, especially for the problems where we ask for explanations, explicitly.

1. Do not modify the files distributed with the homework; we will use our version for grading. Do not include these files with your submission.
2. Before submitting, ensure all your files are in a directory named with your Andrew ID (e.g. bovik/)
3. Please name the written part of your assignment as andrewid.pdf (e.g., bovik.pdf). (There is a scanner in the Engineering and Science library if you wrote parts by hand.)
4. Archive the directory into a .zip (not 7z, gz, tar, bz2, etc.) that is also named with your Andrew ID (e.g. bovik.zip)
5. Follow the assignment link (or click the assignment header) to upload your .zip solution
6. We have attached a LaTeX template that you may use. It is not required that you use this template.

References

[1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded up robust features. In *In ECCV*, pages 404–417, 2006.

[2] Peter J. Burt, Edward, and Edward H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31:532–540, 1983.

[3] M. Calonder, V. Lepetit, M. Ozuysal, T. Trzcinski, C. Strecha, and P. Fua. BRIEF: Computing a Local Binary Descriptor Very Fast. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1281–1298, 2012.

[4] Michael Calonder, Vincent Lepetit, and Pascal Fua. Brief: Binary robust independent elementary features, 2010.

[5] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[6] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.