COMS 4731 – Computer Vision
Spring 2013
Homework #3
Due: Tue, March 26, 2013 by 10:10am

All solutions (e.g., code, README write-ups, output images) should be zipped up as **HW3_yourUNI.zip** and posted on CourseWorks, where 'yourUNI' is your Columbia UNI. Only basic MATLAB functions are allowed, unless otherwise specified. If you are unsure of an allowable function, please ask before assuming! You will get no credit for using MATLAB to answer any questions where you should be answering them. When in doubt, ASK. You may use: `rgb2gray`, `fspecial`, `conv2`, `filter2`, `sort`. Any other functions not listed here please feel free to ask.

Programming Assignment (75 Points Total)

1) In this project, you will write MATLAB code to detect discriminating features in an image and find the best matching features in other images. Your features should be reasonably invariant to translation, rotation, and illumination and you'll evaluate their performance on a suite of benchmark images. The project has three parts: feature detection, description, and matching.

   You will have one "driver" MATLAB script which calls all of the functions below, called **hw3.m**. The images you will test with are included on CourseWorks:
   - **bikes1.png, bikes2.png, bikes3.png**
   - **graf1.png, graf2.png, graf3.png**
   - **leuven1.png, leuven2.png, leuven3.png**
   - **wall1.png, wall2.png, wall3.png**

   First, load all 12 images into memory (at once). Then, we will compare the following images together (more details below on how to compare):

   - bikes1 ↔ bikes2
   - bikes1 ↔ bikes3
   - graf1 ↔ graf2
   - graf1 ↔ graf3
   - leuven1 ↔ leuven2
   - leuven1 ↔ leuven3
   - wall1 ↔ wall2
   - wall1 ↔ wall3

Feature Detection (25 points)

In this step, you will identify points of interest in the image using the Harris corner detection method. The steps are as follows (see the lecture notes for more details). For each point in the image, consider a window of pixels around that point. Compute the Harris matrix H for that point, defined as:

$$H = \sum_p w_p \nabla I_p (\nabla I_p)^T$$

where the summation over all pixels *p* in the window. The weights w_p should be chosen to be circularly symmetric (for rotation invariance). A common choice is to use a 3x3 or 5x5 Gaussian mask. Note that these weights were not mentioned in the lecture slides, but you should use them for your computation.

Note that H is a 2x2 matrix. To find interest points, first compute the corner strength function. There are small variations on the original Harris corner strength definition. For this assignment, we will use the following, which has been shown successful in the vision literature. Compute the first-order spatial gradients I_x and I_y using the Sobel kernels. Next, we compute the product images:

$$I_x{}^2 = I_x I_x$$

$$I_{xy} = I_x I_y$$

$$I_y{}^2 = I_y I_y$$

Next, to efficiently achieve the Gaussian weighting, construct a 2D Gaussian, **G,** by choosing some sigma (e.g., 1-3). Then, apply the circularly-symmetric Gaussian weights as a convolution to *each* of the gradient product images:

$$F_x{}^2 = G * I_x{}^2$$

$$F_{xy} = G * I_{xy}$$

$$F_y{}^2 = G * I_y{}^2$$

Finally, we compute the corner strength image, **CS**, as:

$$CS = \frac{F_x{}^2 F_y{}^2 - F_{xy}{}^2}{F_x{}^2 + F_y{}^2 + \epsilon}$$

where ε helps avoid divide by zero, and should be some small value ~ 1e-16. Once you've computed **CS** for every point in the image, choose points where **CS** is above a threshold. You also want **CS** to be a local maximum in at least a 3x3 neighborhood (or 5x5 neighborhoods work too). Since we did not go over any non-max suppression algorithms in class, a MATLAB function has been included which you may use. To use it, all you supply is your corner strength image, a window radius, and a threshold. For example:

```
[y, x] = nonmaxsuppts(CS, nonmax_radius, corner_thresh);
```

The radius can be values like 1 (for a 3x3 window) or 2 (for a 5x5 window). Now, given an image, you should be able to return the pixel locations of the corners in the image.

Feature Description (25 points)

Now that you've identified points of interest, the next step is to come up with a *descriptor* for the feature centered at each interest point. This descriptor will be a representation you will use to compare features in different images to see if they match.

You will implement two descriptors for this assignment. The first is just a small image window around each feature location (e.g., ~ 11x11 or 15x15 window size). During the matching stage, this will be used to compute normalized cross-correlations between different window descriptors.

The second feature will be something akin to SIFT, but simplified for this assignment (we'll call this *Simple-SIFT*). Recall that in SIFT, once the feature location is chosen, we grid the region around the feature into a 4x4 grid, and within each grid we construct an 8-element gradient orientation histogram, where each vote in the histogram is weighted on the gradient magnitude at that pixel. This yields a 4*4*8=128-length feature descriptor vector. In other words, when giving a vote to a gradient orientation angle in the histogram, instead of adding +1 we add the value of the gradient magnitude at that pixel. The 8-element histograms from each grid are concatenated to form the final descriptor for the entire feature region. Because we aren't using the scale-invariant feature detection method used in SIFT (and hence we don't know the actual size of the feature automatically), we will use a fixed-size 41x41 window radius around every feature location detected from the Harris corner detector. Be sure to be careful of features by the image boundary!

Make sure that the *Simple-SIFT* descriptors that we are creating are rotation invariant. To do this, take the 41x41 image patch, and first compute the dominant orientation. One way to do this is to make a denser gradient orientation histogram (e.g., 45-bins representing every 8 degrees), and find the bin with the most hit counts, again using the gradient magnitude as the weight for the votes. Take the center of the bin as the *dominant orientation*. Then, subtract this value from all gradient orientations when updating each of the 4x4 grids to create the final descriptor.

The last step (not discussed in detail in class) is to normalize the descriptor to (somewhat) account for lighting differences. The way Lowe handles this in the paper is as follows:

- Normalize the 128-dimensional feature descriptor to unit length
- Then threshold it at 0.2, so that no value in the feature can be greater than 0.2
- Re-normalize to unit length

For each corner location, we now also have a 128-dimensional feature descriptor which we will use in the feature matching step, next.

Feature Matching (25 points)

Now that you've detected and described your features, the next step is to write code to match them (e.g., given a feature in one image, find the best matching feature in one or more other images). This part of the feature detection and matching component is mainly designed to help you test out your feature descriptor.

The simplest approach is the following: write a procedure that compares two features and outputs a *distance* between them. For example, you could compute the normalized cross-correlation (NCC) between the image patch around two feature locations. You could then use this distance to compute the best match between a feature in one image and the set of features in another image by finding the one with the best distance score (e.g., minimize normalized difference, maximize normalized correlations). Two possible distances are:

- **Mutual Marriages**: Suppose image I_1 yields matches $d = \{(x_1,y_1), \ldots, (x_n,y_n)\}$ and image I_2 yields matches $s = \{(x_1,y_1), \ldots, (x_m,y_m)\}$. When looking at feature $d_i = (x_i,y_i)$, we look at all of the features in *s* and take the feature with the best NCC match score. Then, we do the same for all features in *s* to find the best matches in *d*. To retrieve good matches, they must agree, such that if d_i finds s_j as its best match, when searching for matches in *s* then s_j should *also* choose d_i as its best match. In this way, they agree on each other and we can reduce ambiguous matches and arbitrary distance thresholding.
- Compute (NCC score of the best feature match) / (NCC score of the second best feature match). This is called the *ratio test*, and you can apply this with different distance functions (e.g., Sum-of-Squared Distances (SSD))

For *Simple-SIFT*, the distance function should be a Euclidean distance. When comparing the ratio of the best and second-best features, a good ratio is typically 0.6-0.75. We are basically trying to prune bad matches such that if we match two features too closely in feature space, they are likely too ambiguous and so we can't tell for sure which is the correct match. However, if the best and second best matches are separated from each other enough in feature space, we can say we have a good match beyond an ambiguity.

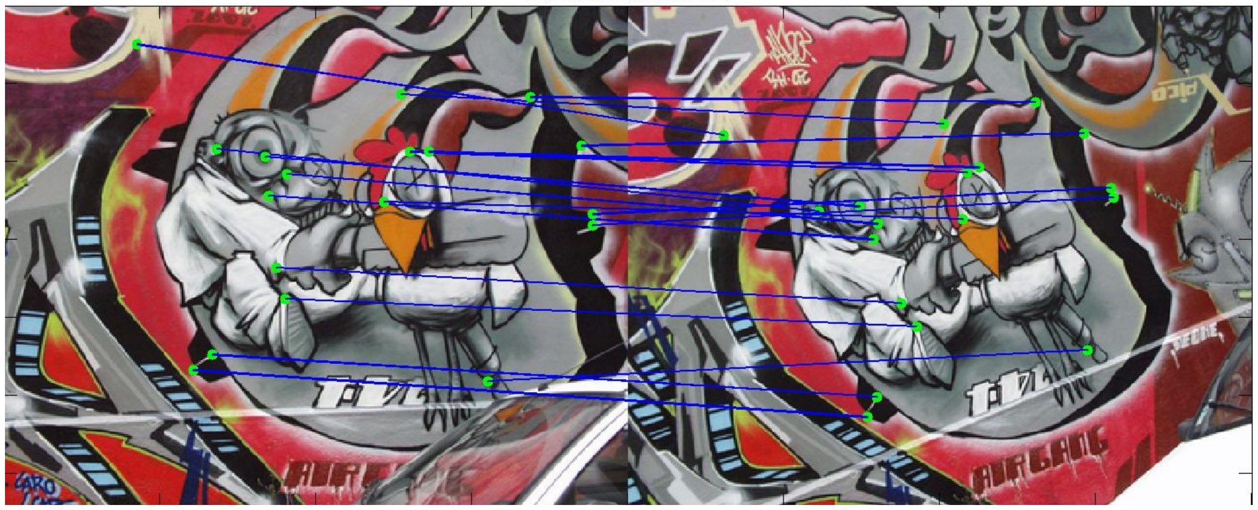
Displaying Matches

To display your matches in MATLAB, given images I_1 and I_2, we construct a side-by-side image:

$$\text{SbS} = [I_1 \;\; I_2]$$

Then, we go through all of our detected matches, and draw a line from the feature in the left image (I_1) to its location in the right image (I_2). Remember that when drawing this line, the feature location in the right image (I_2) should be offset by the width of the left image (I_1) in the side-by-side view. Below is an example.

Simple-SIFT Matching of Harris Features



Go through all of the image pairs listed above (8 of them), run the feature detection/description/matching and display your side-by-side results. To save a figure with overlaid graphics, there are two ways: (1) [manual method] when the figure pops up, go to File → Save As and choose some Image File Format, such as PNG or JPEG; (2) [automatic method] in code, when instantiating a figure, store the handle to it, such as:

```
h = figure(101); % Grab the handle to the figure
% … fill the figure with various graphics
% Now save the figure from the handle
f = getframe(h);
imwrite(f.cdata, 'my_figure.png', 'PNG');
```

Extra Credit (15 points)

For extra credit, given a set of feature matches from two images, compute the Affine Transformation that best aligns the points together using the RANSAC algorithm to remove outliers. Plot the inliers and outliers in different colors (so we can see if the RANSAC algorithm worked well). Then warp one image to the other (either direction is fine: 1 → 2 or 2 → 1) and display them to see how well the transformation was computed. In this question, you may **not** use the `imwarp` function. You must implement the image warping code yourself (you can use `imwarp` to verify your warping function). To do this, you should use the inverse transform to go through pixels in the destination image and find where each comes from in the source image using the inverse transformation. Do a nearest neighbor (floating-point rounding) to deal with sub-pixel locations. One good way to evaluate is to subtract the warped image from the original image to see how well they align. Show this for all 8 pairs of images and comment on how well the alignment worked for each.