EN.600.461/661 – Computer Vision
Fall 2017
Homework #2
Due: 11:59 PM, Tuesday, October 24, 2017
(10% off for each late day)

Changes:

10/8 7:21pm: clarified return values of nonmaxsuppts()

10/3 12:21pm: change cv2.createGaussianKernel() (N/A from python) to cv2.getGaussianKernel()

All solutions (e.g., code, README write-ups, output images) should be zipped up as **HW2_yourJHED.zip** and posted on Blackboard. Only basic Python library functions are allowed, unless otherwise specified. If you are unsure of an allowable function, please ask on Piazza before assuming! You will get no credit for using a "magic" function to answer any questions where you should be answering them. When in doubt, ASK!

For this assignment, you may use: `cv2.cvtColor, cv2.getGaussianKernel, cv2.filter2d, sort, np.linalg.inv`. For any functions not listed here, please feel free to ask.

IMPORTANT: All the material required to complete the assignment will be found in the following files. The skeleton scripts detect_features.py, match_features.py, compute_affine_xform.py, compute_proj_xform.py, ssift_descriptor.py, nonmaxsuppts.py define the structure of each respective function. You are to implement these empty functions as you progress through the assignment. More specifically, the argument parameters and return types are specified in the documentation. Use these to help guide you through the implementation. You are expected to strictly follow the specifications. Please do not edit **nonmaxsuppts.py** or change any of the parameter/return types.

Programming Assignment

In this project, you will write Python code to detect discriminating features in an image and find the best matching features in other images. Then you will estimate a global transformation between the pair of images using your matched features and then warp one image towards the other to stitch them together as a panorama. You'll evaluate their performance on a suite of benchmark images. The project has multiple parts: feature detection, matching, alignment, and warping.

You must include a README file with discussion of your design choices **(5 points)**.

You will have one "driver" script which calls all of the functions below, called **hw2.py (5 points)**. The images you will test with are:

- bikes1.png, bikes2.png, bikes3.png
- graf1.png, graf2.png, graf3.png
- leuven1.png, leuven2.png, leuven3.png
- wall1.png, wall2.png, wall3.png

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

(20 points)

In this step, you will identify points of interest in the images using the Harris corner detection method (see the lecture notes for details). Recall that the result of this process is an image at which each pixel has a value related to the "cornerness" at that pixel. Call this image **CS**. The final step to compute corner locations is a process called non-maximal suppression. Since we did not go over any non-max suppression algorithms in class, a Python function has been included which you may use. To use it, all you supply is your corner strength image **CS**, a window radius, and a threshold. For example:

```
nonmaxsuppts(CS, nonmax_radius, corner_thresh

# returns list of (row, col)
```

The radius can be values such as 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. You may experiment with different corner thresholds. This should return a **list** of (row, col) locations of your corners.

Feature Matching

(30 points)

Now that you've detected 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 the other image for each pair listed

above). This part of the feature detection and matching component is essential towards estimating the global transformation between the pair of images.

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, or a Sum-of-Squared-Distances (SSD)—**you may choose which distance to use -- you don't need both!**. You can 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 SSD, maximize NCC). Then we do the following:

- **Mutual Marriages**: Suppose image I_1 yields corners $d = \{(x_1,y_1), \ldots, (x_n,y_n)\}$ and image I_2 yields corners $s = \{(x_1,y_1), \ldots, (x_m,y_m)\}$. When looking at corner $d_i = (x_i,y_i)$, we look at all of the corners in s and take the corner with the best NCC (or SSD) 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, then when searching for matches in s then s_j *better also* choose d_i as its best match or this wasn't a good match. In this way, they agree on each other and we can reduce ambiguous matches and arbitrary distance thresholding.

The result is a set of potential feature matches between the pair of images

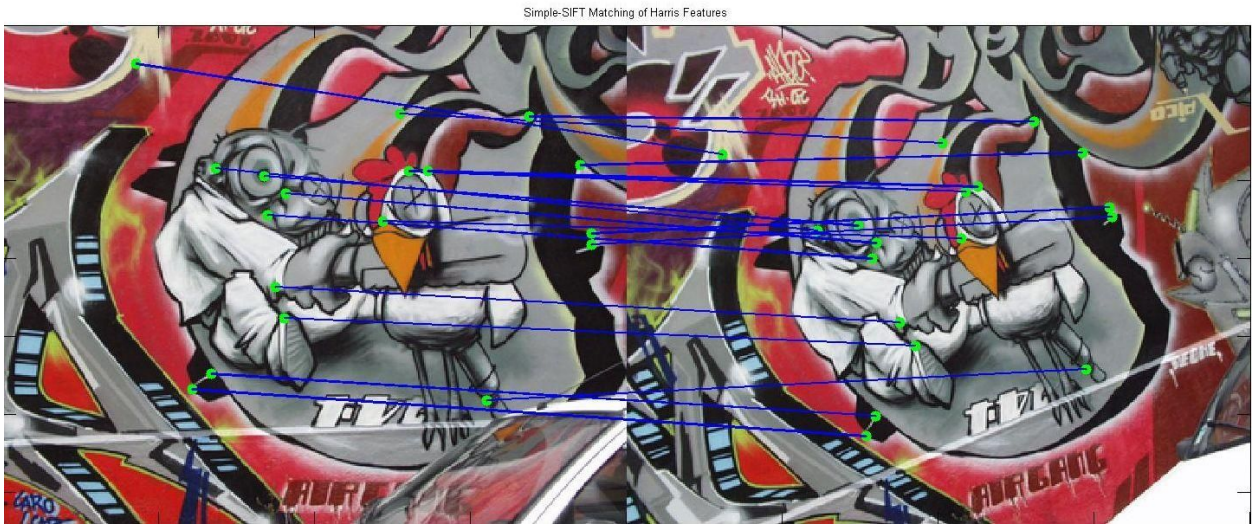
Displaying Matches

(10 points)

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

$$SbS = [I_1 \; I_2]$$

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.



Go through all of the image pairs listed above (8 of them), run the feature detection/matching and display your side-by-side results.

Alignment and Stitching

(30 points)

Next, 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** use the `cv2.warpAffine()` function. Show this for all 8 pairs of images and comment on how well the alignment worked for each in your README. If the alignment looks poor, you should state why you believe this happened (*hint: look at the geometry of the scenes)*.

GRAD STUDENTS REQUIREMENT: (10 points) Repeat this process by computing a fully-projective (8 degrees of freedom) alignment model.

Feature Description (Grad Students Only)

(10 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.

The 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 (ignore those that are too close to the boundary such that a 41x41 window cannot fully fit around it)!

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 each "bin" at the value 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. Repeat the feature matching process above, with the following difference: instead of using the "Mutual Marriage" approach (described above), use the "Ratio Test" described in class. In short, we take the top match and the second best match (according to the smallest L2-distances of the Simple-SIFT descriptors) and only accept this as a match if the ratio of the distance of the best/second-best is <= some threshold (typically 0.6-0.7 is a good choice here). Align the images using an Affine model only and compare the results to the Harris corner + SSD/NCC + Affine method in your README. *Which gave better matches?*

Total points: 100 for undergrads, 120 for grads