CMSC 426
Problem Set 6
Assigned, Wednesday, March 26, 2014
Due, Wednesday, April 9, 2014

Pencil and Paper Exercises

1. **Segmentation with Graph Cut (25 points)**
   Suppose you are using graph cuts to interactively segment an image discussed in class and in the paper by Boykov and Jolly linked to on the class web page. Given an image and user provided information, you must construct a graph so that the mincut of the graph corresponds to a segmentation of the image.

   Suppose the image is:

0	10	40
10	30	50

   and the user has indicated that the pixel with intensity of 0 is background, and the pixel with intensity of 50 is foreground.

   We will be creating a graph in which every pixel corresponds to a node, and there is an edge between two pixels' nodes if they are 4-connected neighbors (that is, if they are neighbors horizontally or vertically; diagonal neighbors do not count). For example, I(x,y) and I(x+1,y) are neighbors, while I(x,y) and I(x+1,y+1) are not neighbors. I'll call these neighboring nodes A and B, and refer to their intensities as I(A) and I(B).

   The graph will have two kinds of costs. First, if A and B are neighbors, we will define the edge weight between them as:

$$w(A,B) = e^{-\frac{(I(A)-I(B))^2}{2\sigma^2}}$$

   σ is a parameter, use a value of 20 for it. Notice that I(A)-I(B) is essentially the image derivative as we go from pixel A to B (in this example, in the x direction).

   Second, since we know that one foreground pixel has intensity of 50, and one background pixel has an intensity of 0, we can estimate that the foreground is generally lighter than the background. For each pixel we use a weight based on the probability that a pixel with that intensity would be produced by the Foreground or Background. Specifically, we use: $-\lambda \log\left(\Pr\left(I(A) \mid Foreground\right)\right)$ and $-\lambda \log\left(\Pr\left(I(A) \mid Background\right)\right)$. Here λ is another parameter. You can just ignore it, assuming it to be 1. Let's suppose we have:

$Pr(I(A) = 10 \mid Foreground) = .05$, $Pr(I(A) = 30 \mid Foreground) = .2$,
$Pr(I(A) = 40 \mid Foreground) = .3$.
and
$Pr(I(A) = 10 \mid Background) = .3$, $Pr(I(A) = 30 \mid Background) = .1$,
$Pr(I(A) = 40 \mid Background) = .05$.

Draw a graph that the graph cut segmentation algorithm would produce in order to segment this image, using the costs described above. What would be the result of running mincut on this graph? What would be the image segmentation that would be produced?

Mosaics

The goal of this problem set is to write code to form a mosaic of two images. We begin with two images that have overlapping fields of view (see below) and stitch them together into a single, larger image. Our strategy is to use SIFT to locate feature points and their descriptors. Each feature in one image is matched to a feature in the second image with the most similar descriptor. Note that many, but not all of these matches will be correct. To find a good set of matches we use RANSAC. We randomly pick three matches, compute the affine transformation that relates these matches, and then apply these to all the feature points. We repeat many times and pick the transformation that maps the most feature points in one image near their matching feature point in the other image. Finally, we use this transformation to combine the images.


We will give you code that computes SIFT features and descriptors for a single image. (Actually, we give you instructions on retrieving this code). This code includes a Matlab wrapper you can use to call SIFT. We also give you two test images, LR1 and LR2 (above), to use in testing your code.

DOWNLOAD SIFT CODE:

The code for finding SIFT features is due to Andrea Vedaldi. Go to www.vlfeat.org and download the latest version of vlfeat code (0.9.18). Download the binary package. Then follow the instructions under Matlab install on the menu on the left. Note that there is a lot of documentation for everything, including all functions in VLfeat. You should just need to execute the command:

```
run('VLFEATROOT/toolbox/vl_setup')
```

Note that this code contains a mix of Matlab and C++, so you may need to install a C++ compiler. This can be found at:
http://www.mathworks.com/support/compilers/R2012a/win64.html
Following this link, you should be able to find the supported compilers for windows, linux, and mac.

We include some skeleton code and a helper function, in the file ps6.m.

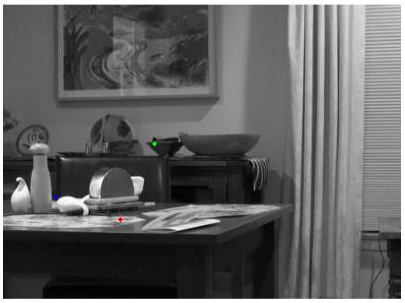
Problems:

1. **10 points:** Run the skeleton code to detect SIFT features in a white square on a black background, and in the two images above. The results should look like this:

Essentially all the code you need to do this is given to you. The point of this problem is just to get you started running SIFT. Include the resulting images in your write-up.

2. **10 Points: Find Best Match.** vl_sift returns two values, F and D. Each column of F contains the x and y coordinates, orientation and scale of each feature. Each corresponding column of D contains a descriptor for that feature. Write code that takes every SIFT feature in image 2 and finds the feature in image 1 with the most similar descriptor. You should compare two SIFT descriptors (which are histograms) using SSD. That is, just compute the SSD of the appropriate columns of the D that is computed for each image. If there are *n* features in image 2, the output of this might be an *n*x5 array, in which each row contains the (x,y) coordinates of a point in image 2, the point in image 1 that matches it best, and the score of this match.

   Now, using this code, write a function, find_best_match, which finds the three matches that have the lowest SSD. Display both images with these points shown in three different colors. That is, the images might each have a red disk, indicating the location of the two points that match best, a green disk, showing the points that match second best, and a blue disk, showing the points that match 3rd best. Run this on LR1 and LR2. Include the resulting images in your write-up. The results should look like this (it's a little hard to see the blue dot without enlarging the image):



3. **10 points: Affine Transform.** Write a function that takes three matching points as input, and computes an affine transformation that maps three points from one image to the matching three points in the second image. So executing:

```
A = affine_transformation(p1,p2);
```

   computes a 2x3 affine transformation, A, that maps the points in p2, represented as a 2x3 matrix in which each column is a point, so that they match the points in p1. Test this by running the code in ps6. You might get results like:

   >> ps6(3)

   ans =

1.0e-15 *

0.3331 0.2220 0.1110
-0.0555 -0.2220 -0.4441

4. **30 Points: RANSAC.** Now, perform RANSAC to find the best affine transformation that maps the most points from image 2 to image 1. To do this, perform the following steps:
    a. Randomly pick three matching points, in the format computed in part 2.
    b. Compute the affine transformation, A, that relates them, using part 3.
    c. Apply A to all points in image 2.
    d. For each point, p, in image 2, that has been matched to point q in image 1, find out whether Ap is close to q ("close" might mean their Euclidean distance is less than 2).
    e. Count the number of points that are mapped by A to be close to their matching point.
    f. Repeat steps a-e many times (maybe a thousand?) and choose the A with the highest total.

    Test your code using part 4 of ps6. In this code, I construct a test case with random points in p1 and p2, some of which are related by an affine transformation. For example, you might get:

    A =

    70.1550 77.2314 14.4582
    36.9972 9.5638 9.4105

    Ares =

    70.1550 77.2314 14.4582
    36.9972 9.5638 9.4105

5. **20 Points: Stitching.** Now write a function, `stitch(J, K, A)`, that will stitch two images, J and K, together, using the affine transformation A. For this problem, just do this in a very simple way. For example, for every pixel in image 2, apply A to find its transformed location. Round off this location to an integer value. Place this value in the new image at this location. At the same time, place all values from image 1 into the new image at their original location. If a pixel doesn't get a value from image 1 or image 2, make it black. If a pixel gets two or more values, you can combine them in any way you want. That is, if image 1 and image 2 overlap, you might use the average, or always use image 1's value, or always use image 2's value. Doing this may produce some artifacts, since there may be pixels in the middle of the new, target image, that don't get any value assigned from either of the original images. You can fix this for extra credit as part of the challenge problem. Test your code on LR1 and LR2 using two affine

transformations. First, create an affine transformation that will translate LR2 100 pixels in the x direction, to the right of LR1. Next, create an affine transformation that will rotate LR2 by pi/16, scale it by .8, and translate it 100 in the x direction and 50 in the y direction. The results of my program are shown below:

Hint: Suppose I and J are the first and second image, and A is the affine transformation. For a point, (x,y), let (x',y') = round(A(x,y)). Then we could just set I(x',y') = J(x,y). One of the things that might make this tricky is that x' or y' might be less than 1. However, the examples in this project are constructed so that you won't have to worry about that. This is something you might want to handle as part of the challenge problem.

Note: Combining two images can be done much more efficiently using the Matlab function imtransform. I'm not recommending that, because I think it will be trickier, though. However, you are free to use imtransform if you want.

6. **10 points** Now write a mosaic program that puts this all together. It runs SIFT on each image, finds the best matching points, runs RANSAC to compute an affine transformation, and uses this transformation to stitch the images together. The results of my program are shown below.

7. **10 points:** Now, take your own photos and run your program on them. Notice that the affine transformation you computed in part 4 is almost a pure translation. This is because I moved the camera very little between pictures. If you move the camera too much, even an affine transformation won't work well. See if you can take two pictures that are well aligned by a more complicated affine transformation. Show the pictures you started with, the result, and the affine transformation that was computed.

8. **Up to 20 Points:** Challenge problem. Enhance this in a significant way. Suggestions:

    a. Improve your program so that you can provide it with many images, and it will stitch them all together.
    b. Improve the program so that you use bilinear interpolation to fill up all the pixels in the image. Show an example where this improves the results. For example, if the second image is smaller than the first image, this will be necessary.

c. Often, due to automatic gain control or other factors, the overall lightness of the images will be different. This looks bad when they are stitched together. Even in the result in Part 5 this is a factor. You can see a slight line where the images overlap, because the left image is a little lighter. Find a way to normalize the images to remove this effect.
d. If you know about projective transformations, enhance the program so that it uses a full homography to align the images.
e. Make up your own improvement. You can check the paper by Brown and Lowe (http://cvlab.epfl.ch/~brown/papers/ijcv2007.pdf) for ideas.