Problem Set 5
CMSC 426
Assigned April 1 2010
Due April 8, 2010

Blob Detection

This problem set deals with some of the key components needed to perform blob detection. This was discussed in class and is also described in: David G. Lowe, **"Distinctive image features from scale-invariant keypoints,"** *International Journal of Computer Vision,* 60, 2 (2004), pp. 91-110, which is available at: http://www.cs.ubc.ca/~lowe/keypoints/. You will be given some sample code on the class web page.

A big part of this process involves smoothing images with a Gaussian filter. Unfortunately, Java's representation of images as unsigned bytes is not adequate for this assignment. To detect blobs, it is necessary to represent images as double (or floating point) numbers, because rounding off values to integers destroys the local extrema that indicate blobs. Therefore, I have supplied you with my own convolution code, in a function called myConvolve. This takes an image, represented as an array of doubles, and a filter, also represented as an array of doubles. So to use this code, you must read in an image and convert it to an array of doubles, as shown in the code. (You are free to use Java's convolution code in completing part 1 of the assignment, for partial credit. But if you want to also complete part 2, you'll need to perform convolution using doubles). One disadvantage of this is that to display the smoothed images, you will need to convert them back to BufferedImages (code isn't provided for that).

1. **40 Points: Gaussian Smoothing.** To find Blobs, you must first generate a multiscale representation of images, by smoothing them repeatedly. We choose the magic numbers sigma = 1.6 and $k = 2^{1/3}$. Given these numbers, we want to smooth an input image by sigma, sigma*k, sigma*k*k, sigma*k*k*k, …. To do this, implement the functions gauss_width and gauss. Gauss_width provides the appropriate size for a Gaussian filter that will ensure that we do not lose important information (a good rule of thumb is to make the filter an odd width, so that at least all information within 3sigma of the mean is included). Gauss returns an array of doubles that represents the filter.

   Next, use these routines to smooth the dog image provided on the class web page. Smooth the dog six times, with Gaussians that have standard deviations ranging from sigma to sigma*k*k*k*k*k. Include these six images when you turn in your assignment.

   Note that there are two ways to do this. Suppose you have smoothed an image with a Gaussian that has a standard deviation of A, and you want to obtain the image smoothed with a Gaussian that has a standard deviation of B, with B>A. You can just smooth the image with a Gaussian with std dev. of B. Or, you can

take the smoothed image, and smooth it again with a Gaussian whose std dev. is sqrt(B*B-A*A). This second approach will use a smaller filter and be faster. Either is acceptable, but I include some code to help you do it the second way.

2. **20 points: Blob Detection.** To detect blobs, you must repeatedly smooth your images. Then you subtract the results at each scale from the next scale. This gives you a set of difference images, which are equivalent to convolving the image with Difference of Gaussian (DoG) filters at multiple scales. Then identify local extrema by comparing a pixel in a DoG image to all neighboring pixels at that scale and at the scales that are the next larger and smaller. So to be a local maximum, a pixel must be bigger than 26 neighbors (8 at the same scale, 9 each at two adjacent scales). Similarly, a local minima must be smaller than all neighbors. In addition, an extrema will only be salient if its absolute value is bigger than .03 (this assumes that the image values range between 0 and 1, not 0 and 255.).

   I have to say, while these basic steps aren't too hard, it took me quite a while to get this to work (and my code may still have some bugs.) So watch out for subtle issues in implementing this.

   You should run your code on the dog image. You should detect all extrema after smoothing 13 times (that means you can compute 12 DoG images, and check 10 of them for extrema. The first and last DoG images needn't be checked, because you can't compare them to scales above and below them). I am including my program's output. There might be a bug in my code affecting the precise position of the extrema, which are shown as red circles. They look like they are displaced a little. So don't worry if your output doesn't look exactly like mine.

3. **20 points: Challenge Problem.** As described in David Lowe's paper, this process can be made much more efficient by subsampling the image, every time it has been smoothed with a standard deviation of 1.6. This would correspond to reducing the size of the image by half after every three smoothings. This is a little subtle, of course, because you need to compare each scale to neighboring scales, and because you need to recover the positions of the extrema in the original image. But this is a challenge problem.

   In addition to including your code, show the results that you obtained in an image, and include a brief description of how much of a speed-up you found.