CSCI 5561: Computer Vision
Spring 2009
Prof. Paul Schrater
Homework #3, Due Mar. 24th

Do the problem set below for credit.

Submit homework as pdf, except for matlab programs which should be submitted as a .m file. Commands should be in order and the file should be executable.

1) Filtering:

Load einstein128.mat;
To view it:
image(einstein128); colormap(gray(256))

- Implement a 5x5 Gaussian filter with sigma^2 =1.5 pixels. Write a function to perform 2D convolution. Convolve einstein128 with your Gaussian filter.
- Grab his eye: eye = einstein128(44:51,69:76); image(eye);

We will try to use this eye image to form a simple eye detector. The basic idea is that convolution implements at each point a dot product operation between the 8x8 eye template and the image. Simple convolution does not work well:

- Convolve the einstein image with the eye image. Note that there are lots of regions that are brighter than the eye region. The problem is that brighter regions in the image respond better no matter what the filter is. This is due to the fact that the dot product between two patterns is $a \cdot b = \|a\|\|b\|\cos(\theta)$.
- We only really care about the cos theta term. Thus if normalize our convolution by the 'length' of the filter and by the 'length' of each 8x8 image patch, we will have a viable eye detector. Length of a filter H means $\|H\| = \sqrt{\sum_{m=1}^{M}\sum_{n=1}^{N} H_{mn}^2}$. Define a new eye filter: eye/||eye||. Convolve einstein with the new eye filter and save the result into an image A. Next we need to compute the length of einstein at each 8x8 region. This can be done via convolution with an 8x8 filter of ones if you make the right changes. Compute the length of einstein at each 8x8 patch and save it into a new image B. Then cos theta at each pixel can be computed as: $\cos(\theta) = A./B$. You should find that the right eye has the maximum value =1. You can find likely

eye positions by thresholding. To visualize thresholding, you can do imagesc(max(A./B,*t*)), for threshold *t* between 0 and 1. About how many locations are better matches to the right eye than the left eye?

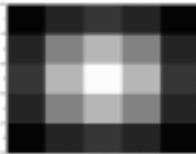
2) Edge detection

Implement an edge detector, with non-maximal suppression. Your result should have the form: function [edgeim,edgepointlists]=edgedetect(im,filt,threshold)

The user should be able to pass to your function the image to be filtered (im), the filt (or filter parameters) you will use to produce candidate points, and a threshold value for thresholding the gradient magnitude. Your function should return two things, an image that is black on edge points and white everywhere else, and an array of edgepoint lists, in which each list contains the indices of a set of points forming an unbroken edge chain. You will probably benefit from knowing about Matlab's structure capabilities, and the different kinds of indexing you can perform within images.

For example, consider the following 5x5 image.

```
im = [   7   28   42   28    7
        28  112  168  112   28
        42  168  252  168   42
        28  112  168  112   28
         7   28   42   28    7]
```



```
1) using find()
indices = find(im>100)
% returns: indices' =[   7    8    9   12   13   14   17   18   19]
 % such that im(indices) = [112   168   112   168   252   168   112   168   112]
% are the values of the pixels that are greater than 100.
% these 1-D indices correspond to 2-D points according to the following ordering
   1    6   11   16   21
   2    7   12   17   22
   3    8   13   18   23
   4    9   14   19   24
   5   10   15   20   25
 % You see then that find has selected the middle 3x3 block.
% Matlab calls this 1-D indexing,, index
% But it calls 2-D indexing (e.g. im(3,5)) subscripts.
% to convert to subscripts, use ind2sub() for instance: [I,J] = ind2sub(size(im),indices);
So that im(I(1),J(1)) is the first pixel, im(I(2),J(2)) is the second pixel, etc.
```

In general it is more convenient to work with indices, because you quickly grab all the indices with some properties, and quickly return the values at those indices, and you can readily store those pixel locations in a 1-D list.

For instance, chain = [1 2 3 4 9 14 13 12 11] represents a U in the image.

To visualize this: imnew = zeros(5,5);
imnew(chain) = 256;
image(imnew)
Let a second chain =[21 22 23 24 25 20 15];

Then we can store both chains in a matlab structure:

edgepointlists(1).chain = [1 2 3 4 9 14 13 12 11] % length 9
edgepointlists(2).chain = [21 22 23 24 25 20 15] % length 7

Structure fields are easy to add:
edgepointlists(1).originalvalues = im(edgepointlists(1).chain); (matlab takes care of all the memory management for you.)

Finally, structures can be quickly accessed for their contents if all the indices are stored in row format:

Alledgepointindices = [edgepointlists(:).chain]

Evaluating your edge detector

Use imread() to load housewtree.jpg.
Run your edge detector at multiple scales and thresholds.
How well does your detector find the tree boundary?
Flower boundaries?
House boundaries?