

# Lab Assignment 1 - CSE 564 Spring 2011

Due: Tuesday, February 22 2010, 11:59pm

In this lab you will implement some of the image processing routines you learned about in class, using matlab (if you want to use another way to implement the lab, let me know). Submit all your work in a zip/rar/tar file on blackboard. Matlab is available in the CS department's UG and G labs. See me if you need an account. If you run it at home, you will need the image processing toolbox as well. Check the lab page for some example images, but look for more on the web. Also make sure you look at the web page for descriptions of the image processing tool box routines, the link is given on the lab page. Please start early.

Here is what you need to submit:

- a) All .m files
- b) A report that shows, for each question, the respective matlab code, the appropriate output (numbers, plots, images, spectra), and a narration of these (that is, a discussion of your solution and your findings, and any observations you may have made). Any questions asked in the text below should also be answered in the report. This report will form the basis for grading. Be professional about it.

First have a look at the matlab tutorials discussed in class and linked to on the class website to get an introduction to matlab. Try a few of the examples, and see if you understand the answers. There are a lot more tutorials on the web, if you have doubts. Finally, Matlab also has help facilities. Use them!

Make sure you understand the difference between a row vector, a column vector, and a 2D matrix. Also understand how you can create an input vector and use it to compute an output vector. For example, try to create a vector  $x = [0 \ 1 \ 2 \ 3 \ \dots \ 100]$  and compute another vector  $y$  from it that contains the squares of the  $x$ -elements. Here, make sure you understand the important meaning of the  $'.'$  when you generate  $y = x.^2$ . Make sure you understand when the  $'.'$  is used and when not. Check that this also works for 2D matrices.

Also, learn how you can graph the vectors and matrices you produced above. This should produce a graph of the function  $y=x^2$ . Make sure you can do this.

Now have a look at tutorial 4 and learn how to read in an image with  $my\_img=imread(filename, format)$ , where format is {gif, tif, jpg, ...}. Note, both the file name and the format must appear in single quotes, such as  $img=imread('some\_image.jpg', 'jpg')$ . Display it with  $imshow(img)$ . See also the section 'Expected results' at the end of this assignment.

If any greylevel image or spectrum appears oddly colored, first set the color map to grey-level using  $color\_map('gray')$ , then process. A window will pop up. This will be used to display the processed image.

Create functions for your matlab code. On the command line you first create and then go to your assignment directory where you want to save all your files. You need to change the current working directory in matlab. There is a box for this on the top bar of the application. To bring up the matlab editor type `'edit <filename.m>`

1. Write a function **gconv(image,sigma)** that applies Gaussian convolution to the 2D image for the given value of sigma, using a normalized gaussian. Matlab has a function  $fspecial()$  which you can use to create a 2D Gaussian. Note, for larger sigmas you need to specify a larger size. Just leave out the  $“;”$  to make sure that the filter matrix falls off sufficiently towards 0. For example, for  $g=fspecial('gaussian', [3 \ 3], 2)$  the matrix is not large enough, but for  $g=fspecial('gaussian', [10 \ 10], 2)$  it is. And if you choose Gaussians with larger sigmas you will need even greater ranges than  $[10, 10]$ . If you do not make the filter array large enough, you will not get the desired degree of blurring. Display  $g$  to see that it is sufficiently large. Use the matlab operator  $imfilter()$  for the convolution. Display the result. Try this with different sigmas. Practical hint: A Gaussian with  $\sigma=0.5$  will do some smoothing, keep doubling the sigma for more and more smoothing. Remember to make the filter matrix large enough.

2. Now write a function **bconv(image,L)** that convolves the image with a box filter. Use the  $fspecial()$  function again, now with the  $'average'$  filter, which is a normalized box. You can set different widths. Display the resulting images as well. Try this with different box sizes. A box filter of width = 3 will do a minimum amount of smoothing. Wider box filters will do more. Compare with the results you got with the Gaussian filter for a comparable amount of smoothing.

3. Let's use convolution to practice indexing into images. Re-implement the convolution with a box filter, but now using for-loops: **bconv2(image,L)**. Recall that an image is a 2D array with each pixel having 3 color channels, RGB. For example, you would obtain the G-channel value at location (y=2,x=3) from image *img* with *value=img(2,3,2)*. You'll need a 4-way nested loop: the first 2 levels are for the (x,y) positions of the image, the second 2 levels are for the (u,v) positions of the filter mask (the size should be flexible). You will convolve each color channel separately (just write three statements in the inner loop, one for each channel). Don't forget to normalize the result (that is, divide by the sum of filter weights which for the box is just the size of the filter). Ignore the image boundaries (just copy the original image to the target first and overwrite the non-boundary pixels in the convolution). Try different filter sizes (typically an odd number) and compare the results with those obtained with the matlab routine above. Use *imshow()* to display the result so you can see if you normalized correctly.

4. Now find the (2D) Fourier transforms of the original images, as well as of the blurred ones you computed above. You can use matlab's *fft2()* (2D Fast Fourier Transform) function for this (you may be asked to convert them first to *double()*). Then, use *fftshift()* to put the origin of the computed spectrum (the zero-frequency band, the DC component, the average term) into the center of the plot. Following, use the *abs()* function to compute the magnitude of each (complex) frequency term before plotting. Using *log()* will bring out smaller values better, or alternatively you could bracket the values using specialized parameters in the display command (see the matlab help files). Use the routine *imagesc()* for this display since your value range will likely be outside [0, 255]. In your report put the spectra images next to the corresponding spatial images.

Compare the Gaussian-filtered images with the box-filtered images, both in the spatial and in the frequency domain. Can you relate what you see in the frequency domain with what you see in the spatial domain (look at artifacts and patterns there)?

5. Now let's have a look at images with spot/speckle noise. There are a few provided in the image set available on the lab page. Smooth a noisy image using filters developed above and compare the result with that obtained using a median filter. Use matlab's function *medfilt2()* to achieve the latter. What is more effective here and why? Will median filtering be useful for general blurring? Note, *medfilt2()* expects a grey level image, so use *rgb2gray()* first.

6. Implement a routine **edgeDetect(image)** that performs edge detection with the (Sobel) mask described in the notes. You can use *fspecial()* to define this filter as well. You need to create 2 intermediate images, one for the x-derivative and one for the y-derivative (use the transpose operator ' for this, that is, if *sf* is your Sobel filter in *y*, then *sf'* is your Sobel filter in *x*). Now you need to combine the 2 (x and y edge) images using the absolute value mechanism (that is, compute *edge\_img=abs(imgx)+abs(imgy)*). Use *imagesc()* to display the result, since the values are out of the [0, 255] bounds again. Display all 3 images, the x-derivative (*imgx*), the y-derivative (*imgy*), and the combined image (*edge\_img*). Use the subplot feature of matlab to create this composite display. Note, matlab also has an *edge()* function, compare your results with the results obtained using that function.

7. Implement a routine called **unsharpMask(image, sigma, alpha)** that does unsharp masking with different blurring factors *sigma* and different weighting factors *alpha*, as discussed in the notes. Use your own function, and *not* the function matlab provides. Do you think the images reveal more information than the edge images, and in what sense? Use *Google Image* to find interesting images that show off the strength of this function. Submit the full-resolution images with your report.

8. Compute the histogram of an image using matlab's *imhist()* function. Use a grey level image for this (or convert a color image before you go on). Take an image with a narrow intensity histogram (that is, an image that is too dark, too bright, or has little contrast). Then widen its intensity range using a simple transfer function that scales the dominant histogram intensity range [*I\_min*, *I\_max*] to the range [0, 255]. For this, use a linear function that maps *I\_min* to 0 and *I\_max* to 255 and all intensities *I* within that range to  $255 * (I - I_{min}) / (I_{max} - I_{min})$ . Now use this transfer function to map the pixel values of the image (representing the x-axis values of the function) to pixel values in the result function (representing the y-values of the function). You need a double for loop (the outer for *y* and the inner for *x*) to translate each pixel. You want the transfer function to map input image values to the desired output image values. The range in which you convert is [*I\_min*, *I\_max*]. First set up an array *x=(0 : 1 : 255)*. Then set up the transfer function *y* to  $y(I_{min}+1 : I_{max}+1) = (255*(x(I_{min}+1 : I_{max}+1) - I_{min})/(I_{max} - I_{min}))$ . Note, we add a factor of 1 to the indices in *x* and *y* since in matlab arrays start at 1, not at 0. Now set all values below *I\_min* to 0 and those above *I\_max* to 255, by writing *y(1 : I\_min)=0* and *y(I\_max+2 : 256)=255*. Plot the

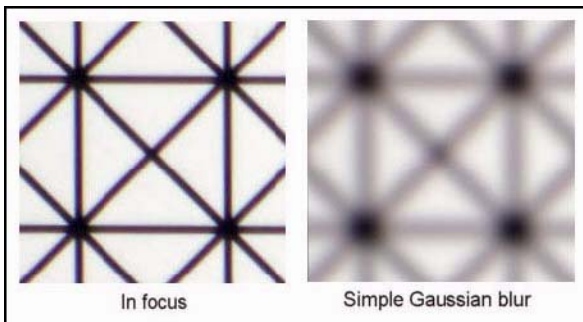
transfer function to make sure it looks OK. Now you can use `y` as the `transfer_function` as stated above. Compare this what you get with the matlab function `imadjust()`.

You can use this function also for *windowing* for enhancing a certain selected intensity range in order to make small detail clearer in this range. Compare the result obtained using histogram equalization (you could use *Google Image* to find good images that show off the strength of the contrast enhancement and windowing). Submit the full-resolution images with your report.

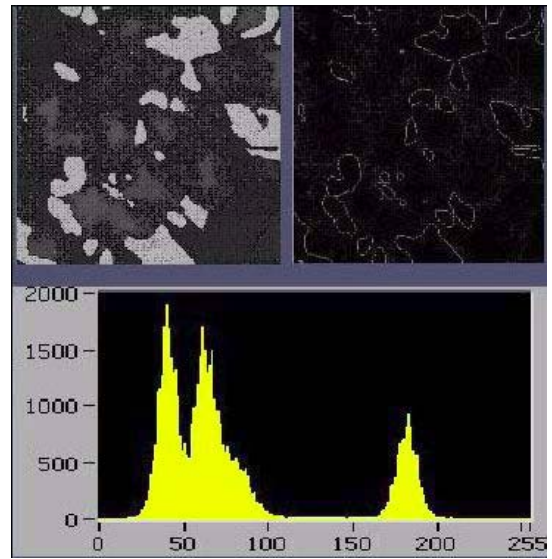
Some hints on expected results:

- (a) Larger boxes and larger sigmas should produce blurrier images and narrower frequency spectra (that is, the higher components are attenuated).
- (b) The blurriness and narrowing should be isotropic (that is, in all directions, not just along the x- or the y-axis).
- (c) In the edge-detected image you should see just the edges, not much else.

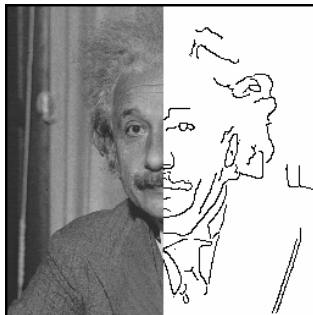
Some examples (for gray level images -- you can convert color images to gray via `rgb2gray()`):



Gaussian blurring



histogram and edge detection



Edge detection / filtering



unsharp masking (left: original, center and right: more detail added back in)