**Lab 7 - Images**

**Due: end of class**

**OVERVIEW**

For this lab, you will do some basic manipulations of gray-scale images. The obvious data structure for an image is a 2-D array, so you will be writing functions that manipulate 2-D arrays.

**TASKS**

**For all tasks:**

There are many image formats (jpg, png, etc.), and we will use PGM (Portable GrayMap) since it is a simple grayscale format.

You may need to use the following linux programs while developing your programs.

* Viewing a PGM image: eog and gimp are standard Linux program for viewing images in various formats.
* Converting between image formats: The Linux utility convert will convert between all standard formats, where its 2 arguments are the input and output files, with format determined by filename suffix. You won't need this unless you want to play with other image formats.

We need a way to convert gray-scale images to arrays, and output arrays into image files. Your instructor will supply you two functions as follows:

* writeImage: writes the argument image (a 2-D array) to outImage.pgm in PGM grayscale format.
* readImage: reads a PGM grayscale format from inImage.pgm into an array, and updates the height/width arguments appropriately (since they are passed by reference).

You may use these functions as is.

**Task 0:**

First, we need a gray-scale image to work on. Save the one supplied by your instructor to $PWD. This file format has 8-bit images - i.e., each pixel is a gray scale value between 0 and 255 inclusive.

Since all our tasks will involve reading an image file, processing the image, and writing the processed image back to a different file, we will first get the basics setup. Write a small program that uses the given readimage/writeimage functions to input the PPM file into an array, copies the image to a 2nd array, and writes the 2nd array back. View the resulting image to make sure its the same as the original image.

**Task 1:**

One way to highlight objects in an image is to make all pixels below a threshold (T1) 0, and all pixels above a threshold (T2) 255. Write a function to highlight the image, using the following prototype:

**void highlight(int image[][MAXHEIGHT],int width, int height, int t1, int t2)**

Write a main program that inputs t1 and t2 from the user, highlights the image, and then writes the image.

This highlighting is called segmentation, though most segmentation algorithms are much more complicated.

**Task 2:**

One way to pixelate an image is to effectively make every nXn *non-overlapping* window contain the same value (the obvious value is the average of the window). For example, if n=2, the following image:

10 20 30 40

11 21 31 41

12 22 32 42

13 23 33 43

may be transformed to:

16 16 36 36

16 16 36 36

18 18 37 37

18 18 37 37

where the 16 was computed by averaging 10, 20, 11, and 21 (after rounding), and so on.

Repeat the earlier tasks, except with a function named pixelate. Your function should work for arbitrary values of n. and you should state in comments how you handle any problematic values for n.

**Task 3:**

A *sliding* window operator replaces each pixel with some function of its 8 neighbors (and itself). Consider the following 3X3 window:

a b c

d e f

g h i

Then, pixel e would be replaced by some function f(a,b,c,d,e,f,g,h,i). One way to detect vertical edges is to use the function (c+2f+i)-(a+2d+g). Note that this is a *sliding window* operator unlike the non-overlapping windows in the previous task - that is, the window is always a window around the pixel whose value is being computed.

Write a vertical edge detection function, named vedgeDet. For simplicity, you can ignore the left/right columns and top/bottom rows of the image.