Image Processing
Spring 2015
Prof. George Wolberg
Homework 2

Due: Wednesday, March 25
Objective: This assignment requires you to implement dithering, halftoning, and neighborhood operations.

1) **unordered_dither** *in n gamma out*

Function *unordered_dither* reads the input in file *in* and applies unordered (random) dither to it before quantizing it to *n* quantization levels. The output is stored in file *out*.

Use the same quantization rule as function *qntz* in homework #1. Recall that this requires the midpoints of the uniform intensity intervals to be assigned to the output. The bias used in *qntz* is to be used here as the dither amplitude. For instance, if $n=8$, the bias is (256/8)*.5, or 16. For every visited pixel, get a random number between 0 and 16 and alternatively add/sub it from the pixel value (to jitter it) before indexing the quantization lookup table created for 8 gray levels. Make sure to clip the jittered pixel intensity to the [0,255] range before indexing the lookup table. Set *gamma* appropriately so that the input image may be properly gamma corrected *before* dithering. Compare these results with those of *qntz* from homework #1.

2) **ordered_dither** *in m gamma out*

Function *ordered_dither* reads the input in file *in* and applies ordered dither to it, storing the output in file *out*. Use a square dither matrix with dimension $m \times m$. This will simulate m^2+1 gray levels at the output. The dither matrix is derived from the matrices given below.

$$D^{(2)} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} \qquad D^{(3)} = \begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix}$$

Use the recurrence relation given below to generate larger dither matrices of dimension $n \times n$, where n is a power of 2.

$$D^{(n)} = \begin{bmatrix} 4\,D^{(n/2)} + D^{(2)}_{00}U^{(n/2)} & 4\,D^{(n/2)} + D^{(2)}_{01}U^{(n/2)} \\ 4\,D^{(n/2)} + D^{(2)}_{10}U^{(n/2)} & 4\,D^{(n/2)} + D^{(2)}_{11}U^{(n/2)} \end{bmatrix}$$

where $U^{(n)}$ is an $n \times n$ matrix of 1's. Note that m must be 3, or it can be any power of 2. Implement $m=3$, 4, and 8. Set *gamma* appropriately so that the input image may be properly gamma corrected *before* dithering.

3) **halftone** *in m gamma out* (EXTRA CREDIT)

Function *halftone* reads the input image from file *in* and halftones it with a $m \times m$ spiral dither matrix. The output is stored in *out*. This is often referred to as clustered-dot ordered dithering because it replaces each input pixel with an $m \times m$ group of (clustered) output pixels to simulate gray values. Note that if the dimensions of the input is $n \times n$ then the output will have dimensions $nm \times nm$ (unlike the previous two algorithms which produce $n \times n$ output). Set *gamma* appropriately so that the input image may be properly gamma corrected *before* dithering.

It is recommended that you initialize an array with a large spiral pattern, i.e., 10×10. Then, when you are asked to use an $m \times m$ spiral ($m \le 10$) you can simply extract it from the larger spiral in the array. That is, all smaller spirals

are embedded in the larger one.

4) **error_diffusion** *in mtd serpentine gamma out*

Function *error_diffusion* reads the input image from file *in* and converts it into a pseudo gray-scale image using the error diffusion algorithm. The particular set of weights used is specified by *mtd*. The user may select from those weights given by Floyd-Steinberg and Jarvis-Judice-Ninke by selecting *mtd* to be 0 or 1, respectively.

In order to avoid the systematic patterns that may appear when scanning all scanlines in the same direction, it is common to use a serpentine scan. In this manner, even lines are processed in left-to-right order while odd lines are processed in right-to-left order. If *serpentine* is set to one, a serpentine scan is used; otherwise ordinary raster scan is used where all lines are processed from left-to-right.

Set *gamma* appropriately so that the input image may be properly gamma corrected *before* dithering. Also, use zero padding to pad the image along the borders. Note that you should first cast the image to datatype *short* to avoid under/overflow of pixel values while distributing error to neighboring pixels. Use a 3-row circular buffer to store the properly cast and padded scanlines necessary to compute the current row of output pixels.

5) **blur** *in xsz ysz out*

Function *blur* reads input image *in* and blurs it with a box filter (unweighted averaging) using a separable implementation. The filter has dimensions $xsz \times ysz$. That is, it has *ysz* rows and *xsz* columns (where *xsz* and *ysz* are odd numbers and not necessarily equal). The output is stored in *out*. Try directional blurs in which one of the filter dimensions is much larger than the other. Make sure to use a circular buffer to store the properly padded scanlines necessary to compute the output without border problems. Implement with pixel replication, but experiment with other modes as well.

Note that this is a separable implementation. This means that you first blur the rows alone, putting the results in a temporary image. Then, do a second pass which blurs each of the columns in the temporary image, putting the results in the output image. Realize that you can define a variable *sum* which must only add the incoming pixel and subtract the outgoing pixel as the window moves across the scanline. The output image is assigned sum/N, where N is the length of the window.

6) **sharpen** *in sz fctr out*

Sharpen *in* by subtracting a blurred version of *in* from its original values and adding the scaled difference back to *in*. The blurred version is obtained by invoking *blur* with filter dimensions $sz \times sz$. The difference between *in* and its blurred version is multiplied by *fctr* and then added back to *in* to yield the output image stored in *out*. Make sure you properly clip values to the range [0, 255].

7) **median** *in sz avg_nbrs out*

Median filter applied to *in* over a neighborhood size of $sz \times sz$. The input values in that neighborhood must be sorted. Then, the median is averaged with *avg_nbrs* pixels below and *avg_nbrs* pixels above it in the sorted list, and the result is stored in *out*. Note that for $avg\_nbrs = 0$, the output is simply the median. If $avg\_nbrs = sz^2/2$, then the output is identical to what *blur* would have computed. Pad the input image using pixel replication before starting in order to avoid border problems. Use a circular buffer to store the padded rows necessary to compute each output scanline. Pad the scanlines using the pixel replication method.

8) **convolve** *in kernel out*

Convolve input image *in* with *kernel* and store the result in *out*. The convolution kernel is stored in file *kernel* as a

2-D array of ASCII numbers, with the first line consisting of two numbers: width and height of the filter kernel.

The input image must be padded to avoid problems at the borders where the convolution kernel falls off the edge of the image. For example, a 5×5 kernel will require two extra rows and columns on each side of the input image so that when the kernel is centered on a border pixel there will be enough image values with which to multiply. Use a circular buffer to store the padded rows necessary to compute each output scanline. Pad the scanlines using the pixel replication method. This problem does not require you to flip the kernel before convolution.

In the C program, you will want to use the following data structure to represent the kernel:

```
typedef struct {
        int width ;          /* width  of kernel */
        int height;          /* height of kernel */
        float *kernel;       /* pointer to floating point kernel */
} kernelS, *kernelP;
```

Make sure to try this with unweighted averaging and compare the execution time with that of *blur*. Try several blurring and edge detection kernels, including those shown below.

```
.11 .11 .11          -1 -1 -1        -10 -10 -10  -10 0 10
.11 .11 .11          -1  8 -1          0   0   0    -10 0 10
.11 .11 .11          -1 -1 -1         10  10  10    -10 0 10
```