Image Processing
Spring 2024
Prof. George Wolberg
Homework 2

Due: Thursday, March 28
Objective: This assignment requires you to implement dithering and neighborhood operations.

1) **HW_errDiffusion** (ImagePtr I1, int method, bool serpentine, double gamma, ImagePtr I2)

Function *HW_errDiffusion* reads the input image from I1 and converts it into a pseudo gray-scale image using the error diffusion algorithm. The output is stored in I2. The particular set of weights used is specified by *method*. The user may select from those weights given by Floyd-Steinberg and Jarvis-Judice-Ninke by selecting *method* 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.

2) **HW_blur** (ImagePtr I1, int filterW, int filterH, ImagePtr I2)

Function *HW_blur* reads input image from I1 and blurs it with a box filter (unweighted averaging) using a separable implementation. The filter has dimensions $filterW \times filterH$. That is, it has *filterH* rows and *filterW* columns (where *filterW* and *filterH* are odd numbers and not necessarily equal). The output is stored in I2. Try directional blurs in which one of the filter dimensions is much larger than the other. Make sure to use a buffer large enough 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.

3) **HW_sharpen** (ImagePtr I1, int size, double factor, ImagePtr I2)

Function *HW_sharpen* sharpens the image in I1 by subtracting a blurred version of I1 from its original values and adding the scaled difference back to I1. The blurred version is obtained by invoking *HW_blur* with filter dimensions $size \times size$. The difference between I1 and its blurred version is multiplied by *factor* and then added back to I1 to yield the output image stored in I2. Make sure you properly clip values to the range [0, 255].

4) **HW_median** (ImagePtr I1, int sz, ImagePtr I2)

Function *HW_median* applies a median filter to I1 over a neighborhood size of $sz \times sz$. The input values in that neighborhood must be sorted. 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.

5) **HW_convolve** (ImagePtr I1, ImagePtr kernel, ImagePtr I2)

Function *HW_convolve* convolves the input image I1 with *kernel* and store the result in I2. The convolution kernel is stored in image *kernel* as a 2-D array of floating point numbers (float).

Copy the whole input image into a padded buffer (of type unsigned char) 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. Pad the scanlines using the pixel replication method. This problem does not require you to flip the kernel before convolution or use a circular buffer to store the padded rows necessary to compute each output scanline.

Make sure to try this with unweighted averaging and compare the execution time with that of *HW_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