CS1112 Fall 2015 Project 4 Part B   due Monday, 10/26, at 11pm

(Part A appears in a separate document. Both parts have the sa me submission deadline.)

You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. *Adhere to the Code of Academic Integrity.* For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

Objectives

Completing this project will solidify your understanding of 2-dimensional and 3-dimensional arrays. You will also work with the jpeg image format and the type `uint8`. Pay attention to the difference between `uint8` and MATLAB's default type `double`.

Image Sharpening

Do you have shaky hands? I do, and the pictures that I take tend to be blurry. Fortunately for me there are algorithms that I can use to reduce the blurriness, i.e., sharpen the image. In this project, you will write several functions to sharpen a user specified region of a jpeg image.

We use some of the ideas from the filtering and edge detection examples in §12.4 of *Insight*. We discussed in lecture that median filtering was superior to mean filtering for noise reduction—the mean filter just makes the image blurry without removing much noise at all. In order to perform image sharpening, however, we actually need to start by making a blurry, e.g., mean-filtered, image. Then by "subtracting" the blurry image from the original, we get what can be called the "detail" of the image:

Notice that the "detail" is similar to what we call "edges" in §12.4 and lecture. Now if we "add" some weighted amount of the detail back to the original image, then the image is "sharpened."

Image Smoothing

Implement this function:

```
function Q = smoothPic(P,r)
% Q is the uint8 array representing the smoothed data from uint8 array P.
% Q has the same size as P.
% Smoothing is done using a mean filter with a neighborhood radius of r.
% r is a positive integer. Assume r<n/2 where n is the smaller value between
% the number or rows and the number of columns of P.
```

See lecture notes or *Insight* §12.4 for a discussion on filtering and the definition of a neighborhood. Your function should work with both grayscale (2-d) and color (3-d) jpeg image data. For data from a color image, the smoothing is done on each of the three layers—red, green, and blue—independently.

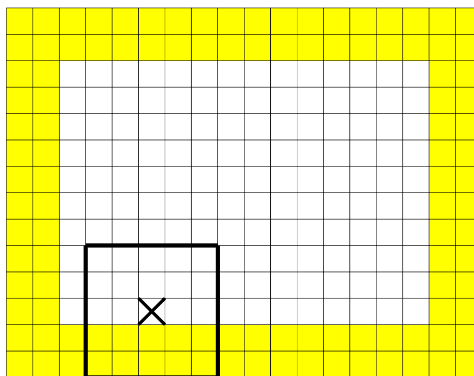
To speed up program execution, you can use vectorized code and the built-in function `sum` as appropriate. Note that `sum` works in type `double` and returns values of type `double`; consider the following statements for example:

```
a= [100 200];  % array of type double
b= uint8(a);   % array of type uint8
c= sum(b);     % c stores the value 300, type double
d= uint8(c);   % d stores the value 255, type uint8
```

To sum the values in a 1-d array, call the function `sum` as shown above. To sum the values in a 2-d array `A`, call the function like this: `sum(sum(A))`. `sum(A)` returns a *row vector* such that the kth component is the sum of the kth column in matrix `A`. Summing this row vector then gives the sum of the matrix `A`.

Image Smoothing, Take 2

Implement another function, `smoothPic2`, to smooth image data. The only difference between `smoothPic` of the previous section and `smoothPic2` is in how the "edge cases" are handled for a neighborhood of radius `r`. In the previous function, as in *Insight* §12.4, the neighborhood is reduced to an appropriate size and shape for each of the edge and corner cases. For example, the radius 1 neighborhood of pixel (1,1) is 2 rows by 2 columns in size, unlike the radius 1 neighborhood of an interior pixel, which is 3 rows by 3 columns. In function `smoothPic2`, you will *pad the image array with extra rows and columns around the edges* as necessary such that the neighborhood of every *original* pixel has the same size. Here's an example for a 2-d case:



The original set of data contains 10×14 pixels, shown in white. If a radius 2 neighborhood is to be used, than a "padding" 2 pixels thick needs to be used, shown in yellow. An example radius 2 neighborhood is drawn above for pixel (10,4)—the *original* edge pixel (10,4)—and it has the same size, 5×5, as the neighborhood of any interior pixel.

What values to use in the padding? The same values in the edge pixels of the original array. First concatenate side by side the leftmost column of the original array, the original array, and the rightmost column of the original array. This gives you a matrix 2 columns wider than the original (one extra column on each side). Next, concatenate to get a matrix 2 rows taller. Now you have a padding 1 pixel thick around the original matrix. Repeat as necessary to get the padding thickness that is needed. For a 3-d array, pad every layer in the same way.

Be careful that you compute the smoothing only for the *original* pixels. The smoothed array that is returned has the same size as the original array.

Image Sharpening

Implement this function:

```
function S = sharpenPic(P,Q,w)
% S is the uint8 array representing a sharpened image given the data for
% the original image (uint8 array P) and the smoothed image (uint8 array Q).
% S, P, and Q have the same size.
% Sharpening is done by weighting (scaling) the difference between P and Q
% by the factor w and adding this weighted difference to P.
```

This function is essentially the code that corresponds to the sequence of diagrams shown on the previous page. The calculations need to be done in type `double` in order to keep the negative values from the subtraction. Recall that you can use the type name as a function name to change the type of a variable, e.g., `double(P)` returns the values in `P` as an array of type `double`. The difference (original data minus the smoothed data) is then multiplied by the weight `w` and added to the original data. These operations may result in values that are outside the range [0,255]. When you set the type of the final array to `uint8`, any value that is less than 0 is truncated to 0 and any value that is greater than 255 is truncated to 255.

Performing Image Sharpening on a Selected Region

Implement this function:

```
function F = selectSharpen(jname,r,w)
% Sharpen (a part of) the jpeg image named by file jname and return the uint8 array
% representing the (partially) sharpened image as F.
% Image sharpening is based on a filter with neighborhood radius r and a weight w on
% the image "detail."
% Display the original image for the user to select an area for sharpening and display
% the final image in a separate figure.
% F has the size of the image named by file jname.
```

This function reads the image data from the file named by `jname`, displays the image, allows the user to select a rectangular area, calls functions `smoothPic` and `sharpenPic` to sharpen the selected area of the image, and displays the entire image which is (partially) sharpened.

Use function `imshow` to display the image. (If you are using a version of MATLAB that doesn't have function `imshow`, use instead function `image`. `image` displays an image using the default figure window size instead of the actual size, but it doesn't affect the rest of the code that you will write.)

Using the `title` function, prompt the user to click two points to select an area for sharpening. The two clicks are the opposing corners of a rectangle with sides parallel to the x- and y-axes. (The first click may be *any* corner, not necessarily top-left.) Use function `ginput` to accept the mouse clicks, which are x-y values. You need to round these values in order to use the x-value as a column number and the y-value as a row number. Furthermore, if a user clicks on the gray area outside the image, you should use the nearest valid column and row numbers for the image. Sharpen only the part of the image selected by the user but display the entire final image, partially sharpened as specified by the user, in a separate figure window. Label both figures using the title area. The command `figure` opens a new figure window.

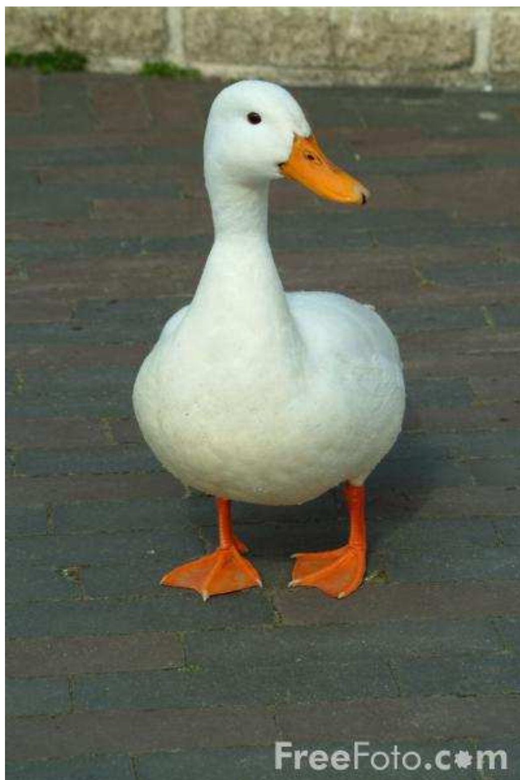
Image filtering is a compute-intensive operation, so throughout this project you should try to be efficient generally.

As you develop your code, use a small image (or select a small area of an image) and a small filter radius (e.g., 1 or 2) to speed up testing. Be sure to try a bigger image and a bigger radius value (e.g., 5) after successful initial testing. Given our use of a mean-filter for smoothing, the appropriate weight to use is 2, although it's ok to try other values. Below are the resulting figures based on this function call:

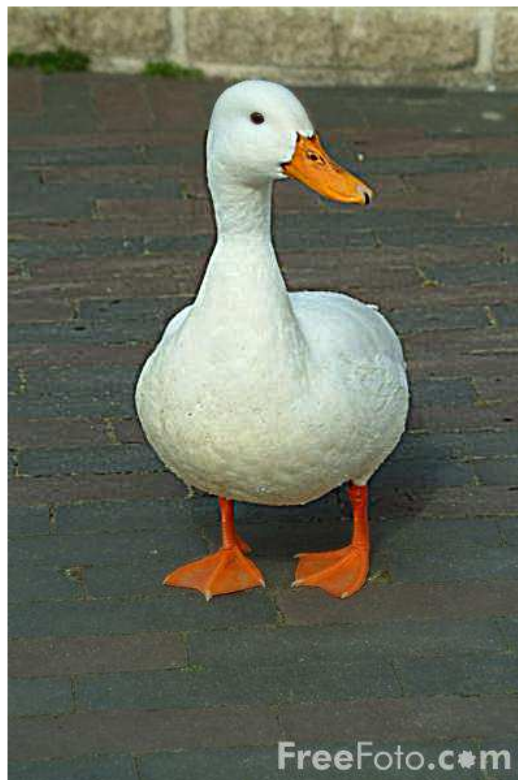
```
S = selectSharpen('duck.jpg',3,2);
```

The area selected for sharpening (as you can see) starts from below the eye and ends at mid-leg. In addition to the sharp "edges" separating the duck from the background, you can also see the texture of the feather more clearly in the sharpened area. The bricks are similarly sharpened in the selected area.

Original Image



Sharpened Image



Submit your files `smoothPic.m`, `smoothPic2.m`, `sharpenPic.m`, and `selectSharpen.m` on CMS.

Final note: You probably used vectorized code and the built-in function `sum` in implementing `smoothPic` in order to get low run-times. That's good—take advantage of the useful features in MATLAB. But some time before the next prelim, be sure to try rewriting `smoothPic.m` *without* vectorized code and without `sum`. That would mean using multiple nested loops to deal with each pixel in the image and each pixel in the filter (neighborhood). It's excellent practice for working with arrays and loops!