Instructions

The purpose of this assignment is to help you gain experience in writing functions and vectorized code, using Matlab's plotting routines and manipulating images. Remember: You are allowed to discuss the homework with other people in the class, but the work you turn in must be your own. *At no point should you be in possession of someone else's code.* There are three deliverables for this homework — function files named *posterize.m*, *setCurve.m* and *applyCurve.m*.

Code Standards

As before, please include the following header at the top of your submission file, with the various fields filled in.

```
% Name: <Your name goes here>
% NetID: <Your NetID goes here>
% <Names of people you discussed this assignment with>
% <Total time spent on problem>
```

Your program submissions will be required to adhere to the code standards as described in class. A PDF version of the standards is available on the course website. *Violations of the code standards will result in style penalties, so be sure to read the requirements carefully!* In the case of egregious violations, you will be asked to fix your code and resubmit a new version altogether, with a 25% penalty.

Tips

Here are some tips for successfully completing this assignment:

1. *Start early!* Do not wait until the last minute. Budget sufficient time for fixing errors. Seek out help from the course staff if you are having difficulty with any of the course material.

2. *Read this document carefully!* People have been losing points due to careless mistakes and/or a failure to follow directions. This is completely avoidable.

3. *Re-read the code standards document* — in particular, read the parts that contain the guidelines for functions and plots.

4. Review the material that was covered in Labs 9 and 10, as well as the lecture on image processing. This will be useful preparation for this homework assignment.

5. Use the top-down approach as discussed in class when building your programs. Begin with a logical decomposition of the overall task in pseudo-code; successively refine this to arrive at your final program.

6. *Test, test, test!* Once you have a working program, test it to make sure it obeys its specifications. You can compare the output produced by your functions to those included in this document to verify the correctness of your submissions.

Creating Posters

In this problem, you will write a function that *posterizes* a given image. Recall that when an image is loaded into Matlab, it takes the form of an array of `uint8` values. A `uint8` value is an integer in the range $[0, 255]$ — in other words, it can only take one of 256 unique values. Posterization works by *reducing* the number of unique RGB values (or *color quanta*) that are present in an image. The resulting image will display abrupt changes in tone, where the original image showed a smooth, continuous gradation.

Write a function named *posterize* with the following specification:

- **Inputs:** The function should take two input arguments. The first input argument is an array containing pixel data. The second argument is an integer in the range $[1, 256]$, which sets the number of color quanta in the output image. *Your function should validate the user input for the second argument, i.e., it should verify that the supplied value is indeed an integer in the range* $[1, 256]$*. If not, your function should display an appropriate error message and exit.*

- **Description:** Given an input image and a desired number of color quanta, your function should compute a posterized version of the input image. The color quanta in the output image must be equally spaced over the interval $[0, 255]$, starting from 0. For example, if the user desires 4 color quanta in the output image, their values would be 0, 64, 128 and 192. The posterization process works by rounding down each

pixel's RGB values to the nearest color quantum. Thus, if a pixel had initial RGB values of $(70, 125, 255)$, the posterization process would remap its RGB values to $(64, 64, 192)$ (for the case of 4 color quanta in the output). Use vectorized code in your solution for efficiency. Apply a median filter of radius 25 to your posterized image before you return it — this will exaggerate the posterization effect. A function named `medianFilter` is available on the course website for this purpose.

- **Outputs:** Your function should return an array containing the pixel data for the posterized image (with the median filter applied as well). Your function should not print anything to the screen, nor display any figures.

Below is an example of how we expect your function to behave. The input image *obama.jpg* may be obtained from the course website.

```
>> im = imread('obama.jpg');
>> p = posterize(im, 2);
>> imshow(p);
```

Figure 1: Applying the posterize filter to the image on the left with the output color quanta set to 2 produces the image on the right.

Notes:

- To test whether the value of a variable is an integer, you can make use of the fact that if x is an integer, then $\lfloor x \rfloor = \lceil x \rceil$.
- Recall that the `error` function can be used to throw error messages and abort program execution.
- In order to avoid `uint8` rounding errors, you should first convert the input image to type `double`. Once the filtering process is complete, you can recast the result to type `uint8` .
- Note that it is quite straightforward to compute what the color quanta should be: if the user desires q quanta in the output image, then these should be spaced $i = 256/q$ apart. The quanta would be $0, i, 2i, \ldots, (q-1) \cdot i$. For example, if $q = 6$, then $i = 42.67$, and the quanta are $0, 42.67, 85.33, 128, 170.67$ and 213.33. Your posterization routine should thus remap every RGB value in the pixel array to one of these 6 values. Only in the last step should these be converted back to `uint8` values; these numbers would then be rounded to the nearest integers to yield 0, 43, 85, 128, 171 and 213.
- How do you test whether your output image has the correct number of color quanta? If the output image is stored in an array named `out`, then the expression `unique(out)` yields all the unique values that are present in the array `out`. The expression `length(unique(out))` will therefore tell us the number of unique values in the output array, i.e., the number of color quanta. If your function works correctly, then this number should match the desired number of color quanta specified by the user at the time the function was called. So in the example above, typing `length(unique(p))` would yield 2.

Curves Adjustment

Oftentimes, a photograph lacks contrast due to reasons such as flat lighting or because it was shot through a dirty window. Usually, such images can be salvaged by applying a *curves adjustment* to the image which improves its contrast. In this problem, you will write functions that increase the contrast of a grayscale image by letting the user specify an adjustment curve graphically.

Image Histograms

The histogram of a grayscale image is a plot of the number of pixels at every brightness or *luminosity* value. For example, figure 2 shows the supplied image *lowcontrast.jpg* and its histogram. A tell-tale sign of an image with low

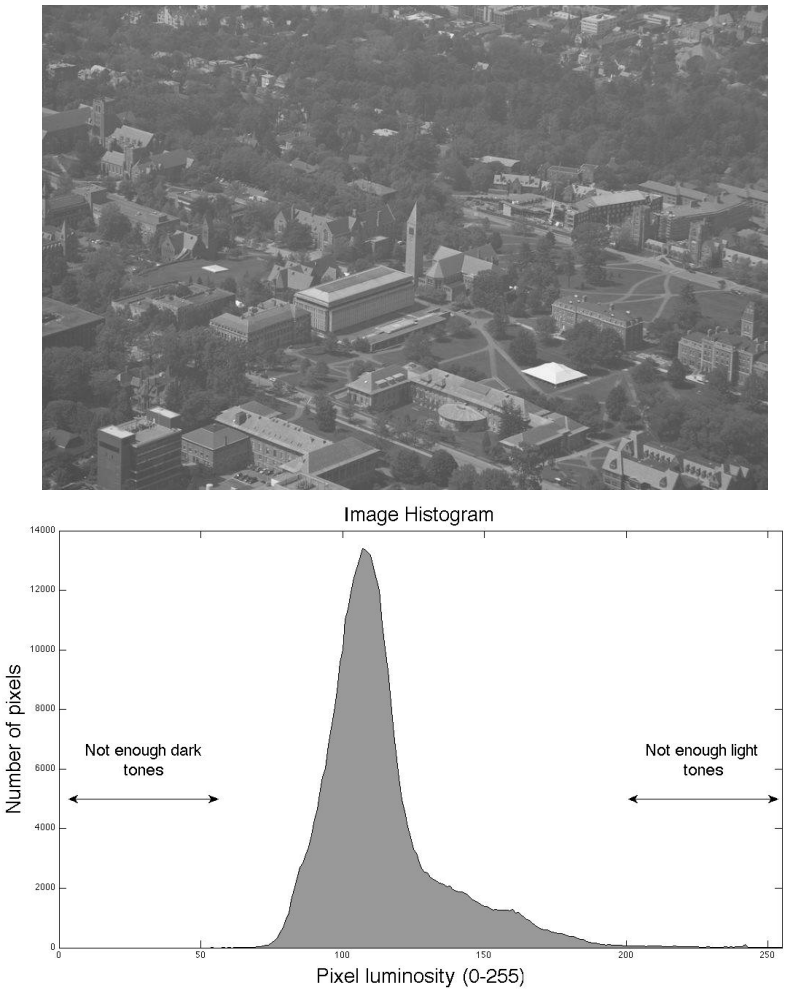

Figure 2: A low contrast image and its luminosity histogram.

contrast is a "bunched-up" histogram, where the vast majority of pixels fall in the middle of the plot, with very few pixels at either extreme of 0 or 255. This can be fixed by "stretching" out the histogram, which increases the overall contrast by making dark tones darker and light tones lighter. Applying a

curves adjustment performs precisely this kind of stretching. Figure 3 shows the outcome of applying a curve to the picture from figure 2, along with the resulting histogram. To get such pleasing results however, we need to stretch

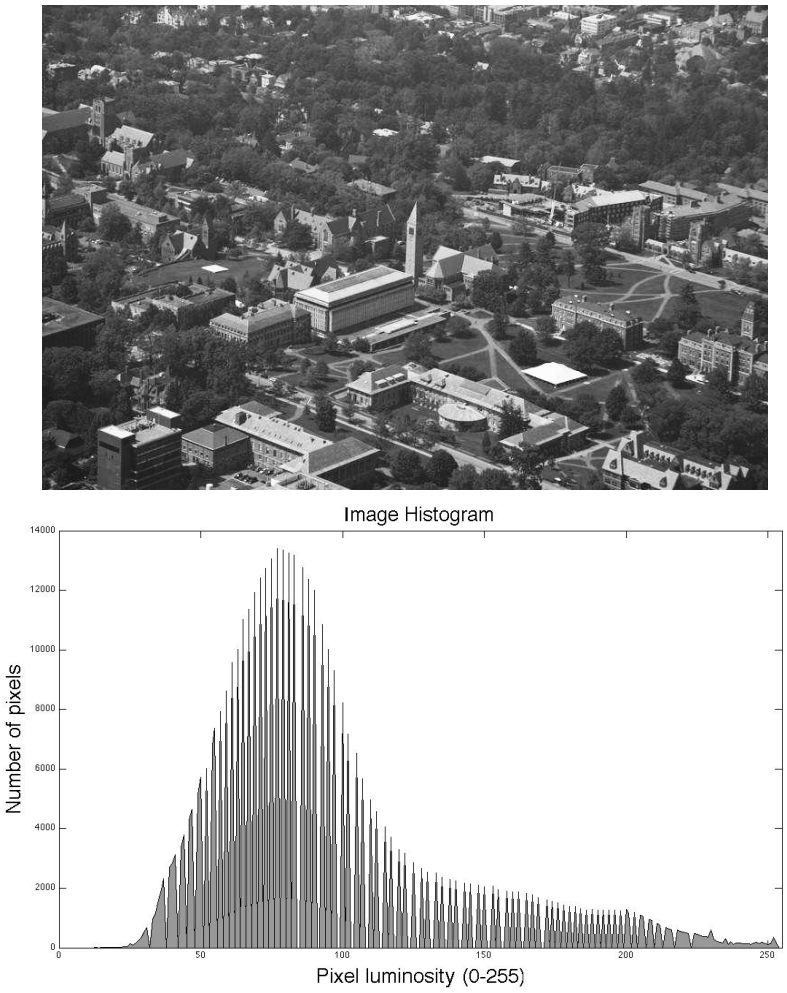

Figure 3: The image from figure 2 with a curves adjustment applied, along with the new histogram

out the histogram in a clever way — in particular, we want to rescale the pixel luminosities in a non-linear fashion. Sigmoid curves are useful for this purpose.

Sigmoid Curves

A sigmoid curve is defined by the function:

$$y(x) = \frac{m}{1 + e^{\frac{\alpha - x}{\beta}}}$$

where m, α and β are constants. For $m = 255$, $\alpha = 128$ and $\beta = 25$, the behavior of the curve is shown in figure 4 over the interval $[0, 255]$.

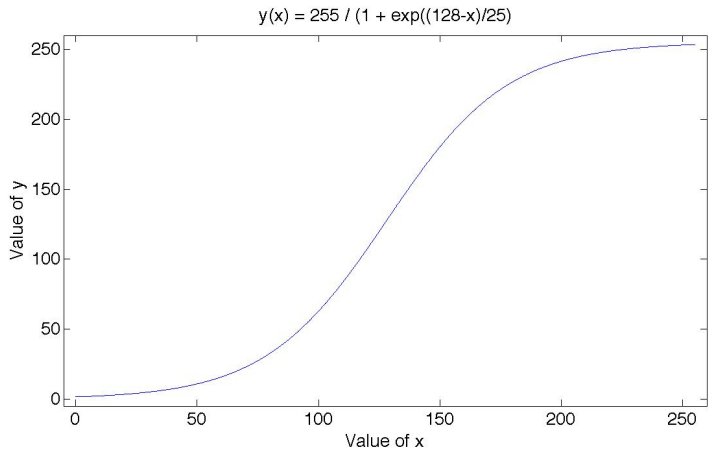

Figure 4: Sigmoid curve plotted over the interval $[0, 255]$

What happens if we were to apply the sigmoid function to every pixel's luminosity value? It has exactly the effect we desire! The dark tones are "dragged" down and made darker, and symmetrically, the light tones are made lighter. Note that by adjusting the parameters α and β, we can change the "steepness" of the curve in the middle section — a steeper curve will make the input image more contrasty, while a smoother curve will make a more subtle contrast enhancement.

The Tasks

To implement the curves filter, we have two sub-problems to solve:

1. We would like to offer the user a graphical way to set the tone curve, so they have control over the amount of contrast enhancement that is applied.

2. Once a curve has been chosen, we need to apply it to the input image.

You will write two separate functions that solve these sub-problems.

First, write a function named *setCurve* with the following specification:

- **Inputs:** None
- **Description:** The function should first plot the sigmoid curve over the interval $[0, 255]$ with $m = 255$, $\alpha = 128$ and $\beta = 25$ using a solid blue line. The x-axis should range from -5 to 260 and have the label 'Input'. The y-axis should range from -5 to 260 and have the label 'Output'. The title of the figure should be 'Tone Curve'. The function should then allow the user to define a new curve by clicking on the plot window as many times as they wish, terminated by a press of the return key. A new sigmoid curve should be fitted to the set of points entered by the user, by calling the supplied *fitSigmoid* function (see the 'Notes' section for details). The new sigmoid function should then be plotted on the same figure as the old curve using a solid red line. The points the user clicked on must also be displayed as black squares.
- **Outputs:** The function should return the new values of the parameters α and β as determined by the *fitSigmoid* function.

Below is an example of how we expect your function to behave. Note that the values of the parameters α and β will vary based on where you click on the plots.

```
>> [alpha beta] = setCurve

alpha =

  128.1427

beta =

   41.9684
```

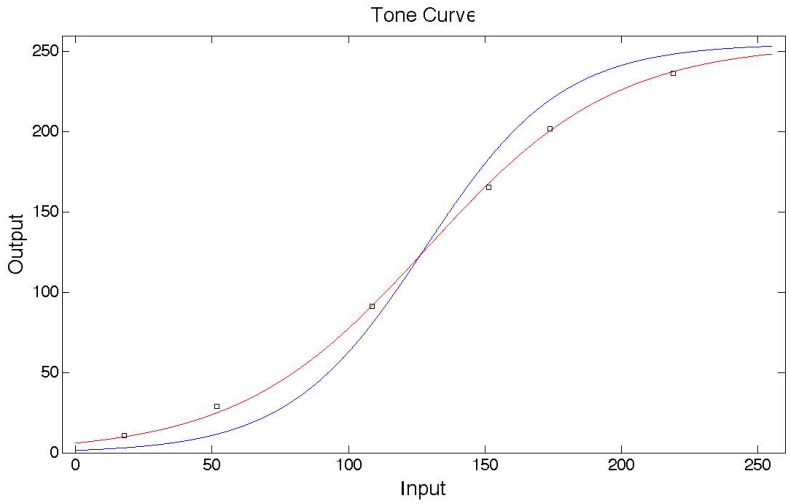

Figure 5: Sigmoid curves displayed by *setCurve*. Initially, only the blue curve is displayed. Once the user clicks on a few points (terminated by the return key), a new curve is fit to these points and displayed (denoted by the red line). The user input points are displayed as black squares.

Notes

- The *fitSigmoid* function accepts two inputs — a vector of x-values and a vector of y-values, marking the points where the user clicked. It then adjusts the parameters α and β to fit a new sigmoid curve through the user input points. The values of α and β for the new curve are returned. Note that the value of m is not modified, and should be fixed to 255. For example, if the points that the user clicked on are stored in vectors `x` and `y`, then the statement:

  ```
  [alpha beta] = fitSigmoid(x, y);
  ```

  will fit a new curve through the points defined by `x` and `y` and return the values of α and β in the variables `alpha` and `beta` respectively.

- Read the documentation for the `hold` command. This is a useful command for situations where you want to add a plot to an existing figure, without erasing what's already there.

Next, it is time to apply the curve to the image. Write a function named *applyCurve* with the following specification:

- **Inputs:** The function should take one input argument, an array containing pixel data. You may assume that the image is grayscale.[1]
- **Description:** The function should first call your *setCurve* function to allow the user to set values for the parameters α and β by clicking on the sigmoid figure. It should then apply the sigmoid function $y = 255/(1+e^{\frac{\alpha-x}{\beta}})$ to each pixel's value. Use vectorized code for an efficient solution.
- **Outputs:** A `uint8` array containing the modified pixel values.

Notes

- As with the *posterize* function, be careful to avoid `uint8` rounding errors. First convert the input image to type `double`. Once the tone curve has been applied, you can recast the result to type `uint8` .

Submission

Please submit your files *posterize.m*, *setCurve.m* and *applyCurve.m* via CMS. If you wish to resubmit a file, simply upload a new version — this will replace any previous submissions. Make sure to go through the attached checklist prior to turning in your files.

[1] Applying curves to color images is a little trickier, as modifying each color channel independently can lead to undesirable shifts in the color balance of the image.

Checklist

- Correctness
  - ☐ Functions have been tested and found to obey specifications
- Documentation
  - ☐ Each function file contains a header with your name, NetID, names of collaborators, and time spent on the problem
  - ☐ Each function contains a header detailing its purpose, as well as a description of its inputs, outputs and assumptions (if any).
  - ☐ The submissions contain comments documenting the various sub-tasks that are solved by the programs
- Presentation
  - ☐ Output from intermediate computations have been suppressed where necessary
- Readability
  - ☐ Variable names are meaningful and follow a consistent naming convention
  - ☐ The code is neatly indented (Ctrl-A, Ctrl-I)
  - ☐ Long lines (more than 80 columns) are wrapped around to the next line
  - ☐ Unnecessary code fragments have been eliminated