Homework 4: Manipulating Images with Python

Original image of Sunset over Florence (the top-left image above) by http://www.flickr.com/people/sherseydc/ (http://flickr.com/photos/sherseydc/2954982676/) [CC-BY-SA-1.0 (http://creativecommons.org/licenses/by-sa/1.0) or CC-BY-2.0 (http://creativecommons.org/licenses/by/2.0)], via Wikimedia Commons

Due Thursday, November 13, at 11:30pm

Turn in your homework via the course web page as an updated version of the `hw4.py` text file that I have provided. Do not rename the file, and do not turn in (or modify) any other files.

Make sure your code can be successfully loaded into the Python interpreter; if not you get an automatic 0 for the homework! Note that we are using Python version 2 --- version 3 has incompatible syntax and libraries, so beware!

Recall the CS131 Academic Honesty Policy! You must say whom you discussed the assignment with at the top of your assignment, and also what other resources you used.

For this assignment you will manipulate images in the PPM format. A `.ppm` image file has the format described here in the items 1-9, except we make the following simplifying assumptions:

- The whitespace after the "magic number" (item 2) is a single newline.
- The whitespace after the width (item 4) is a single space.
- The whitespace after the height (item 6) is a single newline.
- The maximum color value (item 7) is always less than or equal to 255.
- The whitespace after the maximum color value (item 8) is a single newline.
- Each pixel value (item 9) is encoded as a one-byte integer in binary.

Here's an example of a (very small) image in the PPM format:

```
P6
2 3
255
10 23 52 82 3 215 30 181 101 33 45 205 40 68 92 111 76 1
```

Note that the first three lines are encoded as text (ASCII characters) while the last line is encoded as 18 continguous binary bytes.

The 18 numbers represent six RGB triples: the first two triples are the pixels in the top row in the image from left to right, the next two triples are the pixels in the middle row in the image, and the last two triples are the pixels in the bottom row of the image. The file `example.ppm` contains this tiny example image.

Viewing Images: Some image viewing applications can directly display `ppm` files, including the Aquamacs text editor on Mac OS X. However, many image-viewing applications require that the image first be converted to a JPEG. This can be done with the `pnmtojpeg` program at the command line as follows: `pnmtojpeg input-file.ppm > output-file.jpg`. This program comes with many Linux distributions as well as with Cygwin for Windows; it is also installed on the SEAS Unix machines. If you don't already have it, you can download `pnmtojpeg` as part of the NetPBM library. If you use a Mac and have MacPorts installed, you can use the command `sudo port install netpbm`

Now on to the assignment! Try to make use of Python's unique features, such as list comprehensions and `lambda`s: they will come in very handy.

1. To manipulate images, we first need to parse them into a Python data structure. We'll use a dictionary containing four entries: "width" will map to the image's width as integer, "height" will map to the image's height as an integer, "max" will map to the image's maximum color value as an integer, and "pixels" will map to a list of RGB triples (a Python tuple), each of which is an integer. Implement the function `parsePPM` along with its inverse `unparsePPM`. For parsing binary bytes into ints, and vice versa, you may find the `unpack` and `pack` functions in the struct library useful. (The character format `'B'` represents a plain old binary byte.)

   Make sure the dictionaries your `parsePPM` function produces have **exactly** the format described above, because *you will get no credit for this problem or for other problems that have improperly formatted dictionaries*. To help you, I've written a test case that checks whether you produce the right dictionary for the example image above. Specifically, the `testParsePPM` function should return `True`. I encourage you to make your own small test cases for this and later problems.

2. Implement the function `negate`, which takes a dictionary representing an image and returns a new dictionary representing the color negative of that image (see the second Florence image above). To create the color negative of an image, simply replace each pixel value `v` with `max-v`, where `max` is the image's maximum color value.

3. Implement the function `mirrorImage`, which takes a dictionary representing an image and returns a new dictionary representing the corresponding mirror image (see the third Florence image above).

4. Implement the function `greyscale`, which takes a dictionary representing an image and returns a new dictionary representing its greyscale version (see the fourth Florence image above). This new dictionary is in the same format as the input dictionary, with one exception: each pixel is represented by a single greyscale value (an integer) rather than by an RGB triple. This single value is obtained by essentially an average of the three original values for the pixel, except that in practice the following formula tends to produce more pleasing pictures, so you should use it instead: `.299 * R + .587 * G + .114 * B`
   You should round the resulting value to the nearest integer (see Python's `round` function).

   To view the greyscale image, also implement the `unparsePGM` function. This time you'll be creating a PGM image rather than a PPM image. The format is identical except that the "magic number" is now P5 instead of P6, and there is only one value per pixel rather than three.

5. Implement the function `gaussianBlur`, which blurs an image using a Gaussian filter (see the fifth Florence image above). The blurring replaces each pixel's value with a kind of average of the pixel values surrounding it in the image. How this is done is determined by the values in the Gaussian filter, which is produced by the `gaussianFilter` function I've provided.

   A Gaussian filter is a square matrix, represented as a list of lists, of size `2*radius+1` by `2*radius+1` where `radius` is a parameter to the function. The other parameter, `sigma`, is a floating point number that determines the values in the filter. You can experiment with different values for each parameter.

   Given such a filter, you create the new value for a pixel's RGB triple by conceptually placing that pixel at the center of the filter. Then the new value for the pixel's R value, for example, is produced by summing the contribution from each R value covered by the filter, with each such value scaled by the filter. For example, consider this filter (resulting from the call `gaussianFilter(1, 2.0)`):

```
[[0.1018680644198163, 0.11543163961422663, 0.1018680644198163],
 [0.11543163961422663, 0.1308011838638283, 0.11543163961422663],
 [0.1018680644198163, 0.11543163961422663, 0.1018680644198163]]
```

Then the blurred R value at row `i` and column `j`, which we'll denote `R[i][j]` is computed as follows, rounded to the nearest integer:

```
0.1018680644198163 * R[i-1][j-1] + 0.11543163961422663 * R[i-1][j] + 0.1018680644198163 * R[i-1][j+1] +
0.11543163961422663 * R[i][j-1] + 0.1308011838638283 * R[i][j] + 0.11543163961422663 * R[i][j+1] +
0.1018680644198163 * R[i+1][j-1] + 0.11543163961422663 * R[i+1][j] + 0.1018680644198163 * R[i+1][j+1]
```

We will use a *clamping* semantics to handle situations when this calculation would take you out of bounds of the image. Specifically, whenever this process requires accessing a pixel whose row (column) is out of bounds, that row (column) should be substituted with the closest row (column) that is in bounds. For example, if the pixel at row -1 and column 1 is required, you'd instead use the pixel at row 0 and column 1, and if the pixel at row -1 and column -2 is required, you'd instead use the pixel at row 0 and column 0. Similarly, if the pixel at row 100 is required but the image has height 96, then you'd instead use the row value 95.