Project 3: Imageshop

Most of you have probably had the occasion to use some sort of image ediding software such as Adobe Photoshop™ or Adobe Illustrator™. In this assignment, you will have the chance to build a simple version of an image editor which we will call *ImageShop*, which implements several simple operations on images along with a few more interesting ones.

Performing image manipulation in Python requires the use of the Pillow library, which is not always fully included in the Anaconda distribution. This should have already been resolved on everyone's systems from using Karel and later the monkey spectrum lab, but if you are having issues, please contact me and I can get you sorted out.

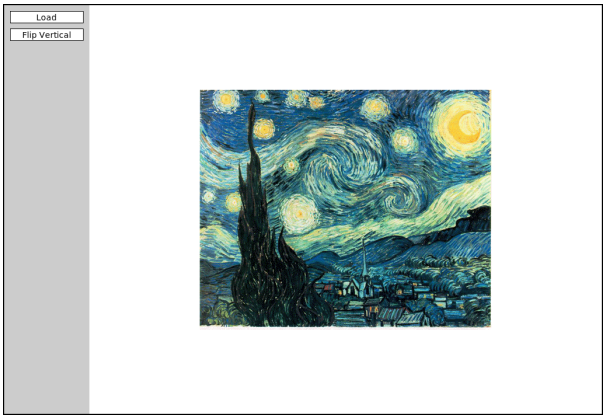
Milestone 0: Understanding the Initial Repository

You can get the starting repository from Github here. This repository contains a skeleton version of `ImageShop.py`, which is the only module that you will need to modify (although you will have to create two others). There are a few other useful library files you will need for this project as well included, so take the time to browse over what resources are available to you. Running the `ImageShop.py` code in the repository will initially create the screen image shown in Figure 1, which include a button area to the left of the screen with **Load** and **Flip Vertical** buttons and then a blank area to the right where the image will be displayed. Clicking **Load** brings up a file chooser that lets you select an image. A directory of sample

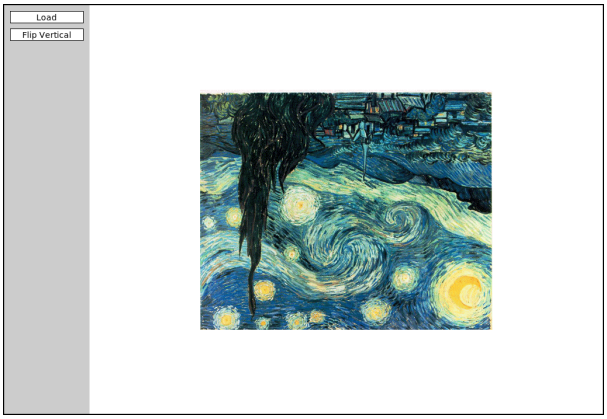
Figure 1: You should get an initially blank screen when you run `ImageShop.py`. An area to the left will hold all the buttons and the images will appear to the right.

images has been included in the repository, so if you navigate to the `images` folder you will see a list of the image files included in this project. If you select `VanGogh-StarryNight.png`,

ImageShop will load that file and center it in the image area, as shown in Figure 2a. If you then click the **Flip Vertical** button, you will get the picture shown in Figure 2b, with the inverted image. Clicking the **Flip Vertical** button again restores the original image.



(a) Initially loading the image and having it appear centered on the screen



(b) The VanGogh image after pressing the **Flip Vertically** button.

Figure 2: Loading and manipulating the `VanGogh-StarryNight.png` image in the supplied `images` folder in the repository.

The code for `ImageShop.py` in the repository uses a new class called `GButton`, defined in the `button.py` library, which is responsible for displaying the buttons on the screen. The function call for `GButton` take the text to display in the button, along with a function to call when the user clicks the button. In the `ImageShop` code, the call to create a new `GButton` object appears within the function `add_button`, which also takes care of placing each new button just underneath the previous one in the button area.

Milestone 1: Adding a Flip Horizontal button

To start you off with a relatively straightforward task, add a new **Flip Horizontal** button that flips the image from left to right. To display the button on the screen, you simple need to copy the code that displays the **Flip Vertical** button, adding new functions `flip_horizontal_action`, which will be your button callback, and `flip_horizontal`, which actually implements the image transformation. The only function that really requires any real change from the `flip_vertical` model is `flip_horizontal`, where you have to reverse each individual row in the pixel array instead of reversing the array as a whole. I suggest first adding the button (and making sure it appears on the screen), then the callback function and finally the transformation function itself to make sure you are understanding what each is doing along the way.

Milestone 2: Add the Rotate Left and Right buttons

Your next step is to add two buttons that rotate the image on the screen by 90°. Once again, the most challenging part of this milestone is adding the functions that actually transforms the `GImage`. Implementing these rotations is a bit tricky, since you have to figure out where each of the old pixels ends up in the new image array. Since the dimensions of the new image are inverted from the original (the new width is the old height and vice versa), you need to create a new array with the correct number of elements in each dimension. Once you have done so, you need to copy each old pixel into the correct position in the new array. This process is illustrated in Figure 3, which shows where each pixel in the first row of the old array (before) moves to the new array (after) during a 90° rotation left.

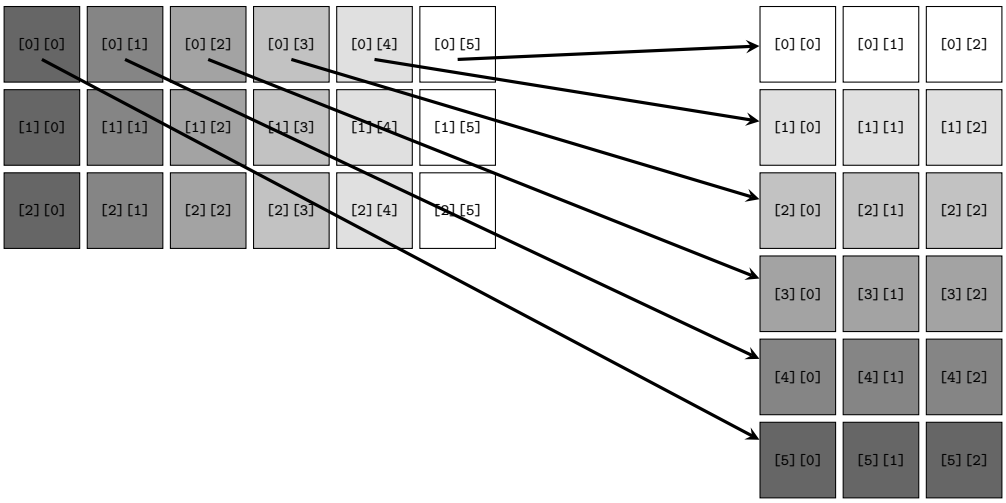


Figure 3: Graphical depiction of rotating an image array 90° counter-clockwise (left). Note that what was previously the last column is now the first row, and what was the first row is now the first column.

Milestone 3: Add a Grayscale button

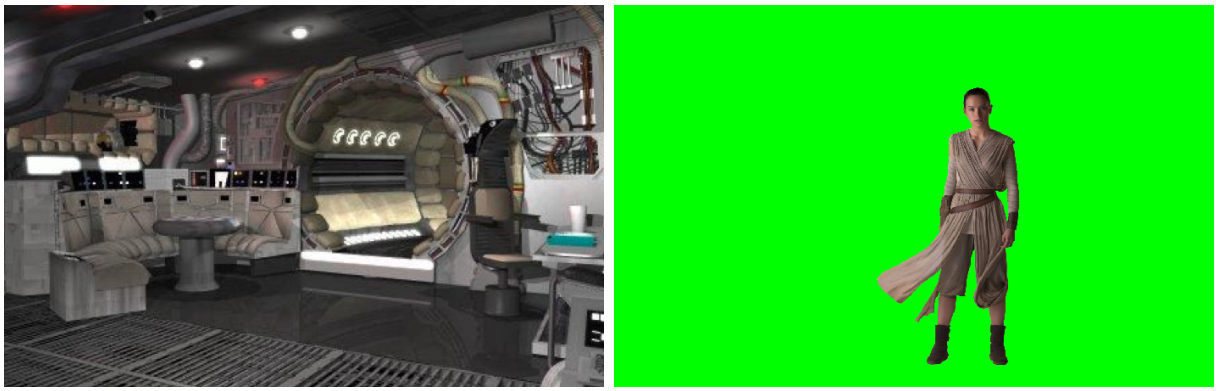
In this simplest of all the milestones, your job is to add a **Grayscale** button that replaces the image with a new one in which the image has been converted to grayscale. The code necessary for this transformation appears not only in the book but also in the file `GrayscaleImage.py` in the repository. To implement this milestone, you should not copy and paste the code from `GrayscaleImage.py`, but instead import the function or functions you need. The advantage of doing so is that there is then only one copy of the code, which is shared between the two applications. Copying code inevitably creates problems for software maintenance, since changes made to one copy may not be incorporated into the other, leading to incompatible versions.

Milestone 4: Implement a Green Screen button

The **Green Screen** button implements an operation that is used all the time in movies to merge actors into a background scene. The basic technique is called *chroma keying*, in which a particular range of colors is used to represent a background that can later be made transparent using a computational process. The most common colors used in chroma keying are green and blue (which give rise to the more specific names green screen and blue screen) because those colors are most easily differentiated from flesh tones.

When the studios use the green screen technique, for example, the actors are filmed in front of a green background. The digital images are then processed so that green pixels are made transparent, so that the background shows through when the partially transparent image is overlaid on top of the background image.

To illustrate this process, suppose that you are making *Star Wars: The Force Awakens* and that you want to superimpose an image of Daisy Ridley's character Rey on top of a shot of the interior of the Millennium Falcon shown in Figure 4a. You then shoot an image of Rey in front of a green screen like that shown in Figure 4b. If you skip the green pixels in



(a) Background shot of the interior of the Millennium Falcon.

(b) Shot of Rey in front of a green screen.

Figure 4

Figure 4b and copy all the others on top of Figure 4a, you get the composite image shown in Figure 5.

The user should always first load the desired background image as per usual using **Load**. Then, when the user clicks the **Green Screen** button, the first thing your program has to do is read in a new image using the `choose_input_file` function in the `filechooser` module in much the same way as the `load_action` function does in the supplied starter code. Once you have the new `GImage`, you need to go through each pixel in both the old image and the new image and replace the old pixel value with the new one, unless the new pixel is green (at least by some definition). It is unlikely though that the pixels that appear in the portion of the green screen image will have a color value exactly equal to the color `"Green"`. Instead, they will have pixel values that lie in a range of colors that appear to be "mostly green". For

Figure 5: Composite image of Rey inside the Millennium Falcon.

this part of the assignment, you should treat a pixel as green if its green component is at least twice as large as the maximum of its red and blue components.

It is not necessary for the old image and the new image to have the same size. Your program should assume that the upper left corners of the two images are at the same place, and update only those pixels whose coordinates exist in both images. The final image, however, should be the same size as the original, and not the overlay. If the images are the same size, as they are for the `Millenniumfalcon.png` and `ReyGreenScreen.png` in the images folder, then the overlay operation will include every pixel in the image. Make sure you test overlaying Rey atop *other* images to ensure that your function behaves properly for differently sized images!

Milestone 5: Implement the Equalize button

Digital processing can do an amazing job of enhancing a photograph. Consider, for example, the countryside image on the left in Figure 6. Particularly when you compare it to the enhanced version on the right, the picture on the left seems hazy. The enhanced version is the result of applying an algorithm called *histogram equalization*, which spreads out the intensities shown in the picture to increase its effective contrast and make it easier to identify individual features.

As described in Section 7.7 of the book, the individual pixels in an image are represented using four single-byte values, one for the transparency of the image and three representing the intensity of the red, green, and blue components of the color. The human eye perceives some colors as brighter than others, much in the same way that it perceives audible tones

Figure 6: Before and after images illustrating the histogram equalization algorithm results. Left image from Wikipedia.

of certain frequencies as louder than others. The color green, for example, appears brighter than either red or blue, and if our eyes were more sensitive to violet hues, the sky would appear more purple instead of the blue we currently see.

Luminance

This concept of brightness can be quantified using the idea of *luminance*, as described on page 256 in the book. That idea is implemented as a `luminance` function, which is defined in the `GrayscaleImage` module in the repository. The value returned by `luminance` is an integer between 0 and 255, just like the intensity values for red, green, and blue. A luminance of 0 indicates black, a luminance of 255 indicates white, and any other color falls somewhere in between.

The histogram-equalization algorithm you need to write for this assignment uses luminosities rather than colors and therefore produces a grayscale image, much as you did when you implemented the **Grayscale** button. The process requires several steps, each of which is best coded as a helper function, which are described in the following sections.

Calculating the image histogram

Given an image, there may be multiple different pixels that all have the same luminance. In fact, there almost certainly are! An *image histogram* is a representation of the distribution of the luminance throughout the image. Specifically, the histogram is an array of 256 integers–one for each possible luminance–where each entry in the array represents the number of pixels in the image with that luminance. For example, the entry at index 0 of the array represents the number of pixels in the image with a luminance of 0. The entry an index 1 represents the number of pixels in the image with luminance of 1, and so on and so forth.

Looking at an image's histogram tells you a lot about the distribution of brightness throughout the image. The images at the top of Figure 7, for example, shows the original low-contrast picture of the countryside, along with its image histogram. The bottom row

shows an image and histogram for a high-contrast image. Images with low contrast tend to have histograms more tightly clustered around a small number of values, while images with higher contrasts tend to have histograms that are more spread out over the full possible range of values.

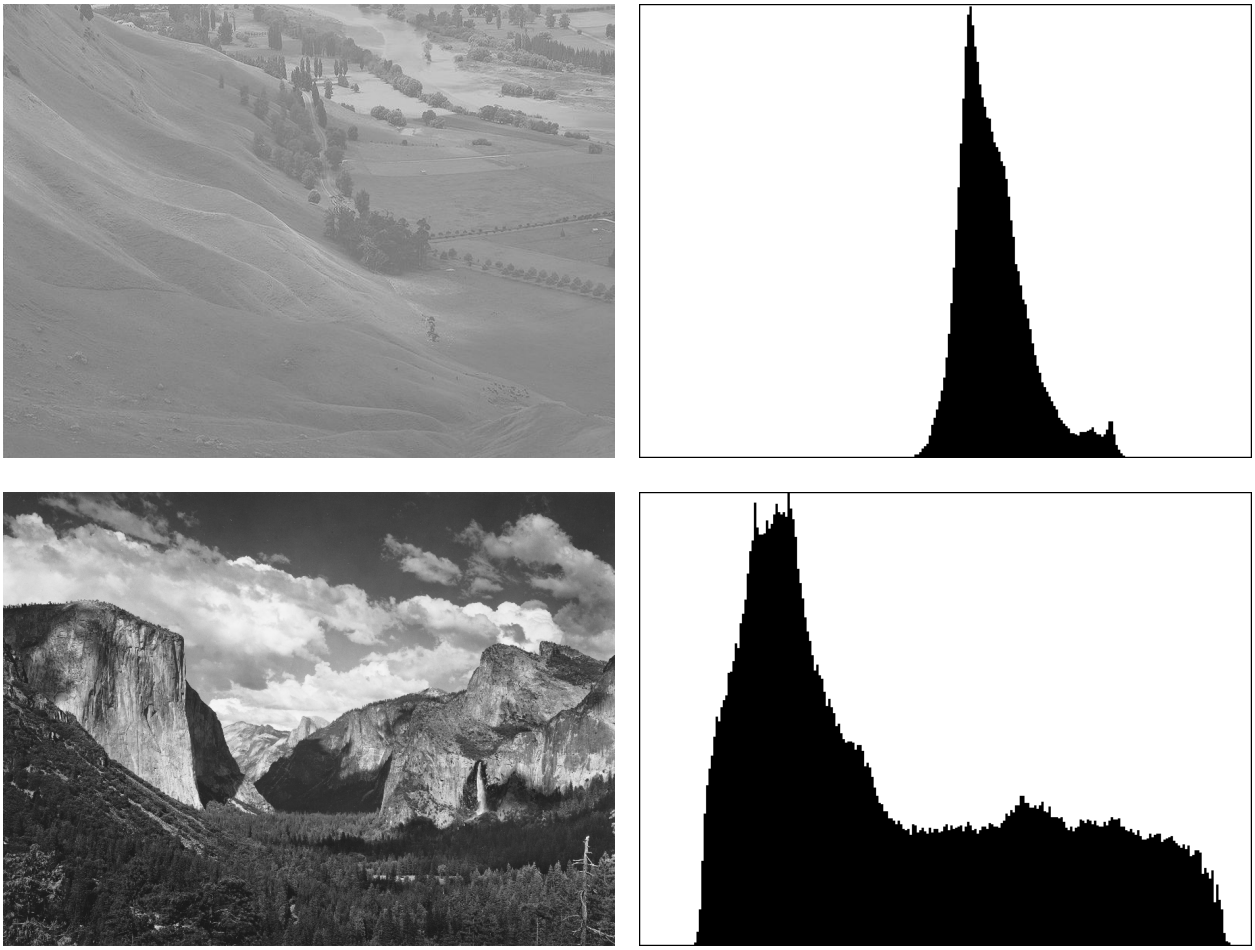


Figure 7: Comparison of histograms for a low contrast image (top) and a high contrast image (bottom). Higher contrast images result in a much broader spread of the image luminosities in the histogram. Bottom left image from Ansel Adams Gallery.

Related to the image histogram is the *cumulative histogram*, which shows not simply how many pixels have a particular luminance, but rather how many pixels have a particular luminance **or lower**. Like the image histogram, the cumulative histogram is an array of 256 values: one for each possible value of the luminance. You can compute the cumulative histogram purely from the image histogram. Each entry in the cumulative histogram is the sum of all the entries in the image histogram up to and including that index position. As an example, if the first six entries in the image histogram are:

$$1, 3, 5, 7, 9, 11$$

then the corresponding entries in the cumulative histogram would be:

$$1, 1+3, 1+3+5, 1+3+5+7, 1+3+5+7+9, 1+3+5+7+9+11$$

or, in other words:

$$1, 4, 9, 16, 25, 36$$

In Figure 8 we can see the cumulative histograms for the two images from Figure 7. Notice how the low-contrast image has a sharp transition in its cumulative histogram, while the higher-contrast image tends to have a smoother increase over time.

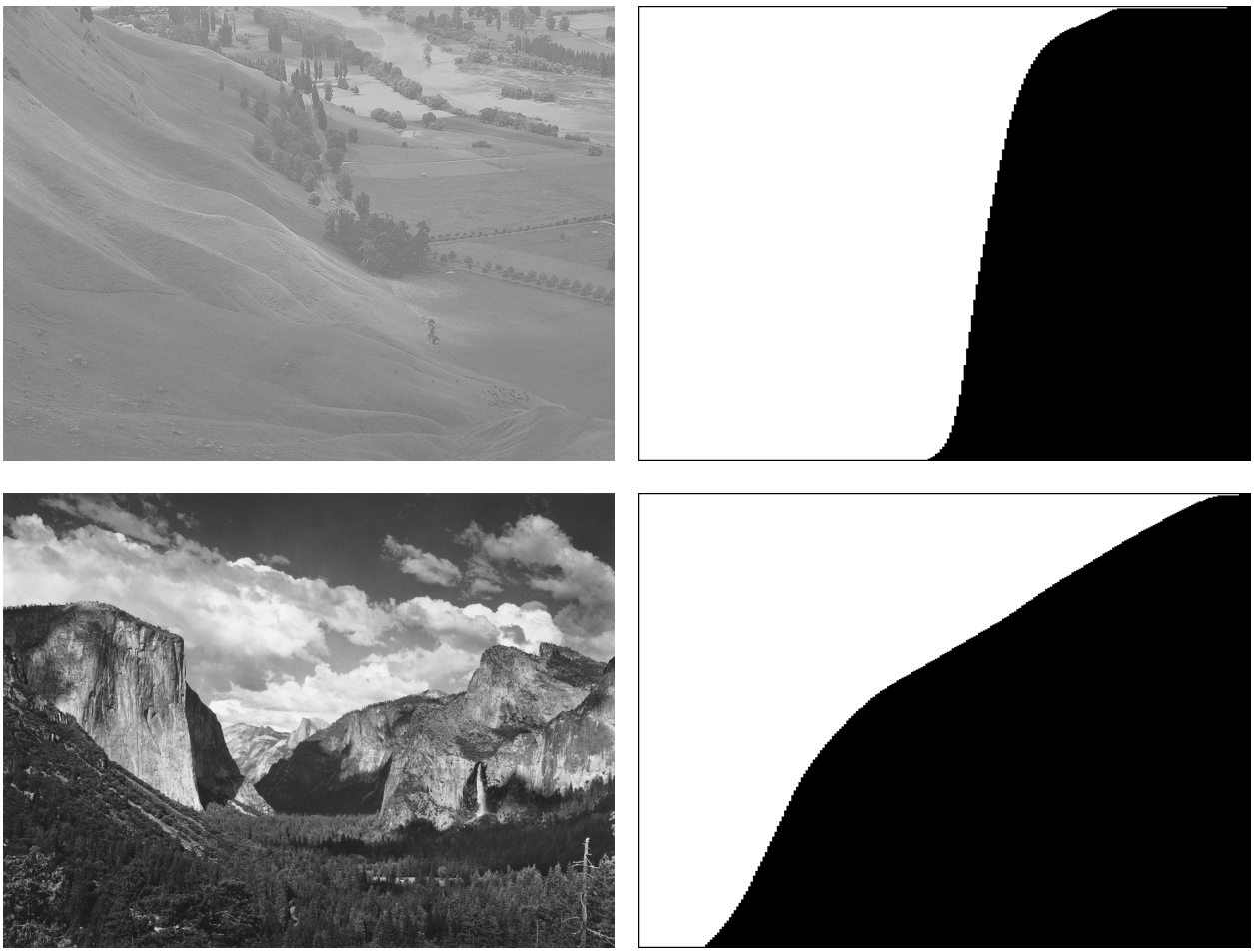


Figure 8: Comparison of cumulative histograms for a low contrast image (top) and a high contrast image (bottom). Lower contrast image tend to create a much sharper "wall" in the cumulative histogram, versus the more gradual increase in higher contrast images.

The histogram-equalization algorithm

The cumulative histogram provides just what you need for the histogram-equalization algorithm. To get a sense to how it works, it helps to start with an example. Suppose that

you have a pixel in the original image whose luminance is 106. Since the maximum possible luminance for a pixel is 255, this means that the "relative" luminance of this pixel is $106/255 \approx 41.5$ percent, which means that the pixel's luminance is roughly 41.5 percent of the maximum possible. If you were to assume that all the intensities were distributed evenly across the image, you would expect this particular pixel to have a brightness greater than 41.5 percent of the other pixels in the image.

Similarly, suppose you have another pixel in the original image whose luminance is 222. The relative luminance of this pixel is $222/255 \approx 87.1$ percent, so we would expect that, should the intensities be evenly distributed, this pixel would be brighter than 87.1 percent of the pixels in the image.

The histogram equalization algorithm works by trying to change the intensities of the pixels in the original image as follows: if a pixel is supposed to be brighter than X percent of the pixels in the image, then the algorithm attempts to map it to a luminance that will make it brighter than as close to X percent of the total pixels as possible. In doing so, the algorithm attempts to more evenly distribute the luminosities across the total available options. Implementing this process turns out to be much easier than it might seem, especially if you have the cumulative histogram for an image.

The key idea is this. Suppose that an original pixel in the image has luminance L. If you look up the L^{th} entry in the cumulative histogram for the image, you will get back the total number of pixels in the image that have luminance of L or less (that is literally what the cumulative histogram tells you). You could then convert this value into the fraction of pixels in the image with luminance L or less by dividing by the total number of pixels in the image. Once you have the fraction of pixels with intensities less than or equal to the current luminance L, you can scale this number (which is currently between 0 and 1) so that it is between 0 and 255, which produces a valid luminance. The histogram equalization therefore consists of the following steps:

1. Compute the image histogram from the original image
2. Compute the cumulative histogram from the image histogram
3. Replace each luminance value (L) in the original image using the formula:

$$\text{new luminance} = \frac{255 \times \text{ cumulative histogram[L]}}{\text{total number of pixels}}$$

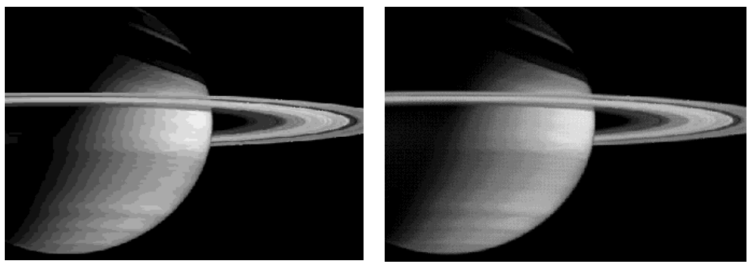
Your task in this part of the assignment will be considerably easier if you decompose the problem into several helper functions and test each function independently as you go. The algorithm for performing histogram equalization is sufficiently complex that it would make sense to code it as a separate module by itself and import the necessary functions into `ImageShop.py`.

Potential Extensions

This project offers essentially unlimited possibilities for extensions. All you need to do is implement features from your favorite image editor! Here are a few ideas though:

- *Implement a "posterize" button.* Shepard Fairey's iconic design of the campaign poster for President Obama's 2008 campaign was widely adapted for other drawings. In this image, all pixels are converted to the closest equivalent color chosen from a restricted list of colors. The image below, for example, contains only red, an off-white ivory tone, and three shades of blue. Your extension could, for example, replace all intermediate colors with the closest match in Java's predefined color palette or use some other strategy you find by searching around the web or thinking up on your own!

- *Implement an averaging filter.* When using low-resolution cameras, images can look rather blotchy. The rightmost image below, for example, shows an image of Saturn taken by the Cassini probe. You can create an image like the one on the right, which looks much smoother, by replacing each pixel with a weighted average of its own luminance and that of its nearest neighbors. You could add a button to your ImageShop program that performs this sort of average.



- *Add a touch-up tool.* If you need to edit an image, it is particularly useful to have a pencil-like tool that allows you to drop a new color on any pixel in the image. The usual strategy is to allow the user to pick a color first and then change individual pixels to that color by clicking on them with the mouse.

- *Implement a crop box.* In the basic assignment, the mouse is used only for buttons, but you could also use it to draw a rectangle on the screen and then limit the functions of the other operators to the region inside the rectangle. It could also make sense to add a **Crop** button that eliminates all pixels outside the crop box.
- *Whatever else you want!* Go wild! Lots of room for some potential ++ scores on this assignment.