Carleton University
SYSC 1005 – Introduction to Software Development – Fall 2013
Lab 11 - Image Filters that Use Lists

Attendance/Demo

To receive credit for this lab, you must make an effort to complete the exercises, and demonstrate your work.

When you have finished all the exercises, call a TA, who will review the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. Finish any exercises that you don't complete by the end of the lab on your own time.

Getting Started

Step 1: Create a new folder named Lab 11.

Step 2: Download Cimpl.py and the three image (JPEG) files that you used for Labs 4 through 7 to your Lab 11 folder.

Step 3: Download more_filters.py and Lab_11_modified_images.zip from cuLearn to your Lab 11 folder.

Lab_11_modified_images.zip is a compressed (zipped) folder that contains the images produced by my solutions to Exercises 2 and 4. You can use these images to help you determine if the modified images produced by your filters are correct.

Exercise 1

For this exercise, use the Online Python Tutor, not Wing IDE 101.

Point your web browser to the OPT Web site:

http://www.pythontutor.com/visualize.html

Delete the example code in the OPT editor.

The *distance* between two integers *a* and *b* is the absolute value of their difference; that is |*a* - *b*|.

Define a function named `find_nearest_integer` that has two parameters: a list of integers, `lst`, and an integer value, `val`. The function returns the integer in `lst` that is nearest to `val`; that is, it determines which integer in the list has the shortest distance to `val`. The function header is:

```
def find_nearest_integer(lst, val):
```

Try these test cases (type these statements below your function definition, then click the Visualize Execution button and step through the execution of your function):

```
nearest = find_nearest_integer([5, 3, 9, 0, 6], 6)  # returns 6

nearest = find_nearest_integer([5, 3, 9, 0, 6], -3)  # returns 0

nearest = find_nearest_integer([5, 3, 9, 0, 6], 8)  # returns 9

nearest = find_nearest_integer([5, 3, 9, 0, 6], 7)  # returns 6
```

Don't delete your function. You may want to refer to it while you work on Exercise 2.

Exercise 2

Launch Wing IDE 101 and open more_filters.py.

The section of the module titled "Exercise 2" contains statements that create a bunch of `Color` objects, followed statements that create six *colour palettes* (lists of colours). For example, `palette_1` is bound to a list containing `Color` objects for the RGB colours black (0, 0, 0), white (255, 255, 255) and gray (128, 128, 128). Because these statements are defined outside of any function, they are executed every time the `more_filters` module is loaded into the Python engine.

An RGB colour can be considered to be the Cartesian coordinates of a point in three-dimensional space; for example, gray can be thought of as the point (128, 128, 128). The distance between two colours can be calculated as the Euclidian distance between the two colour "points".

The `Cimpl` module defines a function that returns the Euclidian distance between two `Color` objects. Here it is:

```
def distance(color1, color2):
    r1, g1, b1 = color1
    r2, g2, b2 = color2

    return math.sqrt((r1 - r2) ** 2 + (g1 - g2) ** 2 +
                     (b1 - b2) ** 2)
```

Step 1: In more_filters.py, define a function named `find_nearest_color` that has two parameters: a `Color` object and a palette. The function header is:

```
    def find_nearest_color(color, palette):
```

The function determines which `Color` object from the list of colours in the palette is nearest to `color`, and returns that colour.

Your function should call `Cimpl`'s `distance` function to determine how near two colours are to each other.

Step 2: In more_filters.py, define a function named `nearest_color_filter` that has two parameters: an `Image` object and a palette. The function header is:

```
      def nearest_color_filter(img, pallette):
```

The function modifies the image so that it contains only those colours in the specified palette. Each pixel's colour is changed to the colour from the palette that is nearest to the pixel's current colour.

Your function should call your `find_nearest_color` function to determine the replacement colour for each pixel.

Use the shell to interactively test your function. For example, to modify an image so that the only colours are black, white, red, green, blue, cyan, magenta, and yellow, we call the function this way:

```
>>> img = load_image(choose_file())
>>> nearest_color_filter(img, pallette_2)
>>> show(img)
```

Try all six palettes. (Remember to reload the original image before you call `nearest_color_filter` with another palette; otherwise, the filter will modify an image that has already been modified.)

Exercise 3

Module `more_filters` contains the `build_solarize_lookup_table` and `solarize` functions that were presented in class. Read the two functions, and make sure you understand how the lookup table is used.

These three statements are defined outside of the functions:

```
      solarize_64_table = build_solarize_lookup_table(64)
      solarize_128_table = build_solarize_lookup_table(128)
      solarize_196_table = build_solarize_lookup_table(196)
```

Every time the module is loaded in the Python engine, these statements are executed to create three lookup tables and bind them to variables.

From the shell, load and image and call `solarize` three times and observe how using a different lookup table each time changes the amount of solarization:

```
>>> img = load_image(choose_file())
>>> solarize(img, solarize_64_table)
>>> show(img)

>>> img = load_image(choose_file())
>>> solarize(img, solarize_128_table)
>>> show(img)
>>> img = load_image(choose_file())
>>> solarize(img, solarize_196_table)
>>> show(img)
```

Exercise 4

The `solarize` function uses a pixel's red, green and blue component values as indices in the lookup table. The values obtained from the table are the new (solarized) values for the red, green and blue components:

```
    red = solarize_table[red]
    green = solarize_table[green]
    blue = solarize_table[blue]
```

We can also have lookup tables that store RGB colours instead of component values. You'll use one such table in this exercise.

Hot metal is an image processing technique in which each pixel's red and green components are modified so that shades of red and yellow are dominant. (The RGB colour yellow is a mixture of red and green, and corresponds to the triplet (255, 255, 0)).

Here's a description of the technique. For each pixel in the image, we first calculate a *weighted brightness* for the pixel, using the formula:

$$weighted\_brightness = 0.3 * red + 0.59 * green + 0.11 * blue$$

(This differs from the commonly used formula for calculating a pixel's brightness,

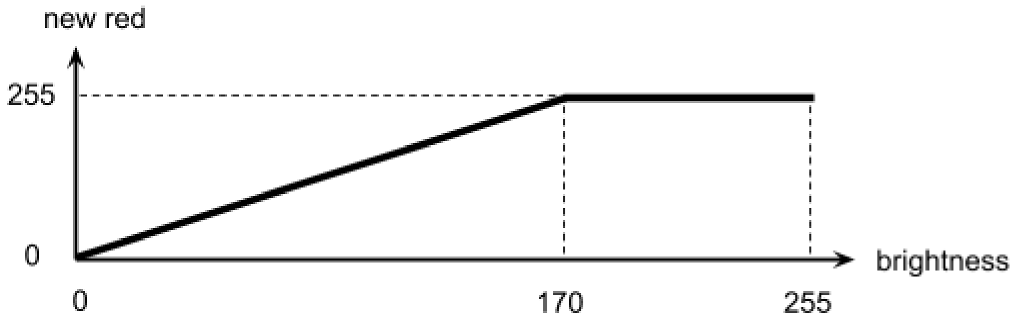
$$(red + green + blue) / 3$$

in that it emphasizes the green component and deemphasizes the blue component.)

We then convert this weighted brightness into an integer, and use that value as an index into a lookup table containing `Color` objects. The colour obtained from `table[weighted_brightness]` is the pixel's new colour.

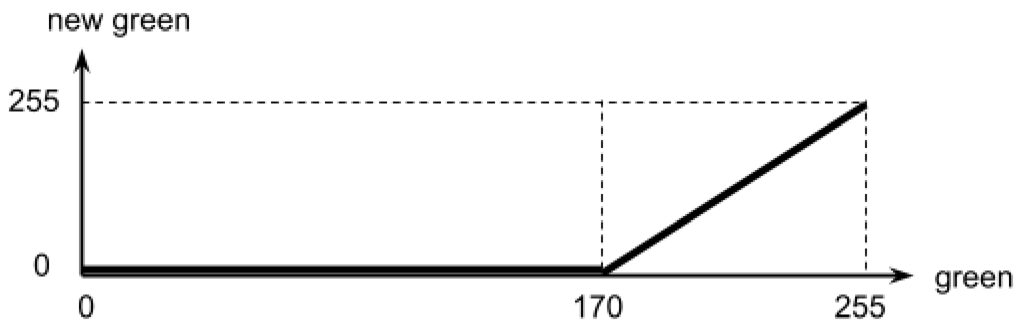
Building the hot metal lookup table is the interesting part of this exercise. Again, this table contains `Color` objects created by `Cimpl`'s `create_color` function, and not integer component values.

The red component of each colour is determined by this graph:



If a pixel's brightness is between 170 and 255, the red component is of its new colour is 255. If a pixel's brightness is between 0 and 170, the red component of its new colour is the value that lies on the straight line between (0, 0) and (170, 255).

The green components of each colour is determined by this graph:



For example, if a pixel's brightness is between 0 and 170, the green component of its new colour is 0. If a pixel's brightness is between 170 and 255, the green component of its new colour is the value that lies on the straight line between (170, 0) and (255, 255).

The blue component of the new colour is always 0.

Here's an example of how one `Color` in the table is created. Suppose a pixel's brightness is 170. From the graphs, we see that the corresponding red component is 255 and the corresponding green component is 0. So, we initialize location 170 in the lookup table with the colour (255, 0, 0). When the hot metal filter is executed, if a pixel's weighted brightness is 170, the replacement colour obtained from the table is (255, 0, 0).

Here's another example. Suppose a pixel's brightness is 255. From the graphs, we see that the corresponding red component is 255 and the corresponding green component is 255. So, we initialize location 255 in the lookup table with the colour (255, 255, 0). When the hot metal filter is executed, if a pixel's weighted brightness is 255, the replacement colour obtained from the table is (255, 255, 0).

Step 1: In more_filters.py, define a function named `build_hot_metal_lookup_table`. The function header is:

```
    def build_hot_metal_lookup_table():
```

This function creates and returns a lookup table containing 256 `Color` objects, with the RGB values calculated using the approach described earlier. Hint: use a loop to "step through" all of the brightness values (0, 1, 2, ..., 255). For each brightness level, determine the values of the new red and green components (you have to determine the formulas from the graphs), create the new colour, and add the colour to the table.

Use the shell to interactively call your function, and inspect the table it returns. Are the colours in the table correct?

Step 2: Add this statement to more_filters.py, immediately after the definition of `build_hot_metal_lookup_table`:

```
hot_metal_table = build_hot_metal_lookup_table()
```

Be careful with your indentation. Make sure this statement isn't inside a function.

Now, every time your `more_filters` module is loaded into the Python engine, a new lookup table will be created and bound to `hot_metal_table`.

Step 3: In more_filters.py, define a function named `hot_metal` that has two parameters: an `Image` object and a lookup table returned by your `build_hot_metal_lookup_table` function. The function header is:

```
    def hot_metal(img, table):
```

For each pixel in the image, the function calculates the pixel's weighted brightness, and replaces the pixel's colour with the colour obtained from the lookup table.

Interactively test your hot metal filter.

```
>>> img = load_image(choose_file())
>>> hot_metal(img, hot_metal_table)
>>> show(img)
```

Wrap-up

1. Remember to have a TA review your more_filters.py file and give you a grade (Satisfactory, Marginal or Unsatisfactory) before you leave the lab.
2. Remember to save a copy of your more_filters.py file (copy it to a flash drive, or email a copy to yourself, or store it on your M: drive - remember, files left on your desktop or in your Documents folder are not guaranteed to be there the next time you log in).
3. Log in to cuLearn, click the Submit Lab 11 link and submit more_filters.py. After you click the Add submission button, drag the file to the File submissions box. After the icon for the file appears in the box, click the Save changes button. At this point, the submission

status for your files is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button. After you've finished uploading the file, remember to click the Submit assignment button to change the submission status to "Submitted for grading".