The Beauty of Numbers

We've jumped through a lot of hoops to represent an image with numbers. Just to recap, we had to:

- chop the image into (many!) discrete units called pixels.
- assign a number (greyscale) or 3 numbers (color) to each pixel to represent the intensity/color of each pixel.
- store all of those numbers in binary so that a computer can understand them.

Okay, it wasn't that many hoops.

But here we are. And now it's time to reap the benefits of having our picture made up of numbers by using math to modify the numbers.

Notation

To perform mathematical operations on our digital images we're going to follow this pattern: start with a source image, then modify it through some mathematically-defined operation to create a new (presumably different) destination image. Let's define the source image as X and the destination image as Y. I use capital letters as a kind of short hand to refer to the *entire* image, which means that the operation we perform happens to every pixel.

As an example, a "no op" operation (i.e., one that doesn't do anything) would look like this:

$Y = X$

In other words, every pixel in the destination image is set to the exact same value as the corresponding pixel in the source image.

Addition/Subtraction

What happens if we add or subtract a number from a pixel? In the case of a greyscale pixel, adding a value will *make the pixel brighter* and darkening will *make the pixel darker*.

$Y = X + v$
$Y = X - v$

Where v is a single number in the case of greyscale images.

Some individual pixel examples:

pixel value + offset value = new pixel value
0.1 + 0.5 = 0.6
.75 + .2 = .95

.5 - .5 = 0.0
…

We have to be careful not to go below 0.0 or above 1.0 while doing the math, otherwise we'll have values that are beyond the valid range of intensities that we originally defined (What's darker than black? What's whiter than white? The latter question actually has an answer, called "super white," that we'll get to sometime later). This process of limiting each pixel to lie in the range between 0 and 1 is called **clamping**.

0.5 + 1.0 = 1.5 (invalid!)
0.5 + 1.0 = 1.0 (clamped, valid)
0.5 - .9 = -0.4 (invalid!)
0.5 - .9 = 0.0 (clamped, valid)

Color doesn't complicate things too much. Instead of adding a single offset value, you add a vector value (a triple) to the color:

[Roriginal Goriginal Boriginal] + [Roffset Goffset Boffset] = [Rfinal Gfinal Bfinal]
[0.1 0.5 0.3] + [.2 .2 .2] = [.3 .7 .5]
[0.1 0.5 0.3] + [0 .5 0] = [.1 1.0 .3]
…

The R, G, B values of the offset you want to add or subtract could all be the same or could all be different. For instance, you can brighten the overall red of a picture while simultaneously darkening the overall green and blue by adding an offset like [0.1 -0.2 -0.2] to each pixel in your RGB image (Note: by allowing for both positive and negative values in the offset, we can accomplish both addition and subtraction just by using addition). Once again, we have to be sure to clamp so that the values remain valid.

Caution

What happens if you take a digital image, add .3 to every pixel, then subtract .3 from every pixel? The math seems quite straightforward:

X + 0.3 - 0.3 = Y
X = Y

HOWEVER, this does not include the effects of clamping! So in general, adding an offset then subtracting the very same offset will NOT give you the image you started with. With the example offset of 0.3, for instance, any pixels that started with values between 0.7 and 1.0 will end up as 0.7. Why? Let's go through it.

Here is the addition step:

X + 0.3 = Y

y is the new pixel value. If x is greater than 0.7, however, y will end up greater than 1.0. So our automatic clamping step (remember, we want to keep the color values valid) will kick in and set y explicitly to 1.0. Now we do the subtraction:

Y - 0.3 = Z

For all pixels x that are between 0 and 0.7, this process will result in z = x. But for those that were clamped to 1.0, z will end up equaling 0.7 while x might be anything between 0.7 and 1.0.

This is called a **loss of information**. The mathematical manipulation you tried to use, as innocent as it seemed, actually caused the pixel values that were originally between 0.7 and 1.0 to all be set to 0.7. So if you had some detail in the bright portions of the original image it would now be lost. So tread carefully. The most successful technique for ensuring that you are back where you started is to use your software packages "undo" feature instead of trying to "undo" the mathematical operation that you just attempted.

(insert graphs and images to demonstrate addition/subtraction and clamping)

Multiplication (and Division)

Multiplication (and, by extension, division) can also be used to alter an image though the effects are subtly different from addition. Consider multiplying each pixel value by a number less than 1.0:

X * 0.2 = Y

The resulting pixels, y, are all 20% or their original values (x). This is different from addition in two important ways: it keeps black pixels black, and it changes brighter pixel values more than it changes darker pixel values. Recall: addition changed every pixel by the same amount.

An 80% reduction like the example above will darken the image but won't result in any clamping (can you prove this to yourself?). If we multiplied by a number greater than 1, however, we would risk clamping.

Caution (again)

Does this mean we can apply a multiplier of 0.5, then change our minds and multiply again by 2.0 and return to the original image? Let's see:

X * 0.5 = Y
Y * 2.0 = Z

The 0.5 multiplier won't result in any clamping, so it seems that x would equal z. HOWEVER, this is not true! To understand why, we need to go back to how computers store numbers.

With a bit depth of 8 bits per pixel, the computer can store a maximum of 256 unique values. Let's assume those values are evenly distributed in the range 0 to 1. In other words, the values are:

0 = 0
1/255 = 0.00392
2/255 = 0.00784
3/355
…
255/255 = 1.0

Now let's take a candidate pixel value of 3/255. If we multiply this pixel by 0.5 (or ½), the new pixel value *should* be 3/510. But the computer, forced to store this number in only one of 256 possible values, has to choose something different. The two closest options are 1/255 and 2/255. They're both equally good… and equally bad. To see why, let's try to reverse the operation by multiplying by 2.

3/255 * ½ = 1.5/255. Rounded down (say) this is 1/255.
1/255 * 2 = 2/255 which does NOT equal 3/255!

Once again there is information loss.