

Creative Computing II

Image Filtering

16th March 2010

This lab sheet covers the use of *Octave* for image reading, writing and filtering.

1. This part covers reading an image file, and converting a colour image to grayscale.
 - (a) using `imread`, read in a colour image; ideally one with many different colours; be sure to save the image data to a variable (not called `i`!) and to suppress printing the data.
 - (b) call the *Octave* function `size` on your image data. Check that the pixel resolution is what you expect.

The `size` function should return a three-element vector, with the first two entries being the width and height of the image and the third being 3 (corresponding to the three colour channels in a digital image).

- (c) use `imshow` to display your image.
 - (d) one way of converting an image to grayscale is to average the red, green and blue channels for each pixel. Use the expression on p. 112 of Volume 1, Chapter 5 of the subject guide to evaluate this; display the result with `imshow`.

The expression in the subject guide is `mean(image, 3)`, and that produces a matrix with each entry being the average of the three channels. Unfortunately, that matrix does not work directly with `imshow` (though it does with `imagesc` as suggested in the subject guide).

The reason is that the original image data as produced by `imread` is in fact a special kind of Octave matrix; most Octave matrices are implicitly matrices of double floats, while `imread` produces a matrix of bytes (`uint8`). To see this, use the `typeinfo` operator on your image data, and on the average data: one will be of type `uint8 matrix`, while the other is just `matrix`.

This wouldn't be a problem, except that `imshow` treats the two matrix types differently. If something is a `uint8 matrix`, then `imshow` automatically sets the range of the colour space to be 0–255; if the image data passed is a generic `matrix`, then the range is 0–1. So to use `imshow`, you would need to scale the data by 255:

`imshow(mean(image,3)/255)`.

- (e) the sRGB primaries used in digital images and displays are not equally luminous, and the sRGB values are not linearly scaled, so computing the mean is not the correct way to compute the equivalent grayscale value. Instead, the weighted average of the linear C_l values (Volume 2, Chapter 1) should be used: weighted by the amount each channel contributes to Y (the luminance in CIE XYZ coordinates), and then gamma corrected:

$$C_{\text{gray}} = 1.055 \left(\begin{pmatrix} 0.2126 & 0.7152 & 0.0722 \end{pmatrix} \begin{pmatrix} \left(\frac{C_r + 0.055}{1.055} \right)^{2.4} \\ \left(\frac{C_g + 0.055}{1.055} \right)^{2.4} \\ \left(\frac{C_b + 0.055}{1.055} \right)^{2.4} \end{pmatrix} \right)^{1/2.4} - 0.055$$

Implement this transformation, and display the resulting grayscale image with `imshow`.

As in the part above, one obstacle to deal with is the numerical format of the Octave images. Before being able to do floating point arithmetic with the integers, the `uint8` matrix data needs to be converted to a regular `matrix`; this can be done using the `double` function. The following is a function definition implementing the above transformation:

```
function y = grayscaleize(im)
    lin = (((double(im)+0.055)/1.055).^2.4);
    wavg = 0.2126*lin(:,:,1)+0.7152*lin(:,:,2)+0.0722*lin(:,:,3);
    y = 1.055*(wavg).^(1/2.4)-0.055;
end
```

The first line of the function body computes the linearized RGB values from the sRGB (gamma-corrected) values; then the second line performs a weighted average, and the third converts that average back into the gamma-corrected colour space.

- (f) visualize the difference between the two methods of producing grayscale images. *Here, it is appropriate to use `imagesc` on the difference between the two grayscale image matrices. You should find (by eye) that green regions have their lightness underestimated by the unweighted average, and blue regions an overestimated lightness.*
 - (g) save your two grayscale images using `imwrite`, and load them into an image processing program. Can you produce a visualization of the difference between the images? What about in *Processing*?
2. Just as with audio, an image filter kernel can be applied using convolution: in two dimensions, we use the `conv2` operator.
- (a) Taking your grayscale image from part 1, experiment with applying echo, blur and edge detection filters using `conv2`.
 - (b) Verify that you get the same results by using the Fourier method (with `fft2` and `ifft2`) of applying filters. *Just as with the audio application of `fft` to implement convolution, some care needs to be taken to get the output size right. Other than that, you should find that the two-dimensional Fourier operators allow you to implement two-dimensional convolution.*