CS 322 Project 1: Color Calibration

out:	Thursday 1 February 2007
Code due:	**Wednesday 14 February 2007**
Writeup due:	**Friday 16 February 2007**

1 Background

All color imaging devices—including cameras, displays (i.e. monitors), scanners, and printers—represent colors as short vectors, usually with either 3 or 4 elements. A camera or scanner has sensors for red, green, and blue; a color display has red, green, and blue primary colors; a color printer uses at least four different pigments (sometimes as many as eight).

In practice these devices are used together: we want to photograph a scene with a camera, view the photograph on a display, and print it out, and have the colors all bear some resemblance to one another. For this reason every device has a numerical method that goes with it to convert between its colors and the colors of other devices. Two common kinds of computations are linear transformations (used for very linear devices like cameras) and interpolation tables (used for "messy" devices like color printers).

In this project you'll calibrate a camera, a scanner, and a printer by finding transformations from these devices' colors to your display's colors and vice versa. The basic method in each case is simple: feed some known colors through the device, then fit a model to what comes out to get a description of what the device does to colors. You then invert this model to correct colors for the device.

2 Camera and scanner calibration

Cameras and scanners are both tools for measuring the color of real objects. Both devices typically represent colors as 3-vectors in the RGB space, where the first element is for red, the second element is for green, and the third is for blue. RGB colors are additive, so black is [0, 0, 0] and white is [1, 1, 1].

The goal (at least the way we'll state it) is to measure an object and end up with a color that, when displayed on your monitor, matches the color of the object. We are fortunate to have a reference object available: a chart known as the Macbeth ColorChecker that contains 24 carefully controlled colors.

Cameras, scanners, and displays are both fundamentally linear devices, which means that a linear transformation can be expected to be a good model for how the device handles colors. You will calibrate these devices by measuring the ColorChecker and finding the 3×3 linear transformation that best matches the device output to the known, correct color values.

3 Printer calibration

A color printer is a device for creating a real object that has a given color. Printers typically work in the CMYK color space, where a color is a 4-vector and colors are subtractive (so white is [0, 0, 0, 0] and black is [1, 1, 1, 1]). Theoretically, only Cyan, Magenta, and Yellow (the CMY) are needed, but because the three of them in practice produce muddy blacks, a fourth pigment of blacK is added.

The goal (at least the way we'll state it) is to print a color and end up with a piece of paper that matches a patch of that color on the screen. Since matching colors between prints (which only reflect the light that falls on them) and displays (which emit their own light) is dicey at best, we will take advantage of our calibrated scanner: the goal will be to make a print that, when we scan it back in, results in the same color we started with.

Because RGB is a 3-vector and CMYK is a 4-vector, this transformation is not necessarily a function – that is, a single RGB color can map to multiple CMYK colors. Because of this, we will work in a simplified CMY space (with no black). However, see the starred problem for more details.

Printers, unfortunately, are by no means linear devices. This means that we can't just fit a matrix transformation. Instead we will use linear interpolation among a set of tabulated values.

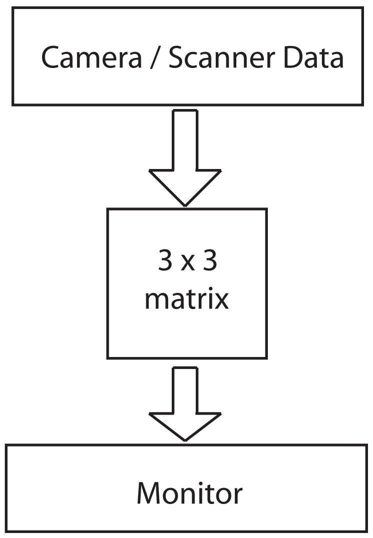
4 Gamma

In practice, monitors are not linear devices – for a given component a in an input color, the monitor produces brightness proportional to a^γ. Experimentally, it has been determined that $\gamma = 2.2$ is a good approximation for the gamma of a standard CRT monitor. Because of this, displaying a raw image in MATLAB using **imshow()** may look unreasonably dark. To see a representative image on the screen, use **imshow(img .ˆ 0.45)**, where *img* is your image data. Note the pointwise exponentiation: this raises every element in the matrix *img* to the specified power. If you forget the period, this will attempt to raise the **matrix** to the appropriate power, which is definitely not the desired behavior (and typically not defined for the matrices we'll be working with).

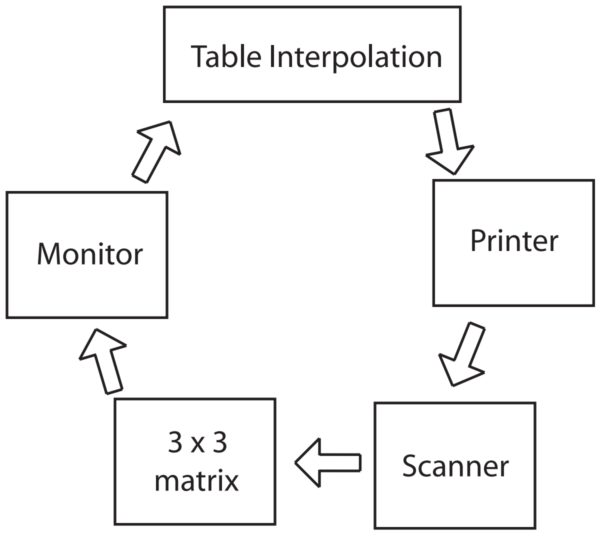
5 Assignment

Your task can be broken up into four parts:

1. Create code to generate the calibration for a scanner and a camera
2. Create a function that, given your calibration from (1) and a similar image, processes the image according to the calibration, returning a color-matched image
3. Using (1), create code to generate the calibration for a color printer
4. Create a function that, given your calibrations from (1) and (3) and a similar image, processes the image according to the calibrations and returns a color-matched image.



Workflow for calibrating cameras / scanners



Workflow for calibrating printers

For both the scanner and camera, your calibration should be a $[3 \times 3]$ matrix M such that for a given RGB triplet $[r; g; b]$ as recorded by the camera or scanner, $[rn; gn; bn] = M[r; g; b]$ will be the RGB values to display on the monitor. Note that there should be different matrices for each scanner or camera that you calibrate, although the process for generating that matrix will be the same.

For the printer, your calibration will be an evenly spaced grid of sample points in RGB where each point contains the CMY value of the printer that corresponds to that RGB 3-vector. Computing the CMY color for a particular (not necessarily grid-aligned) RGB value entails linearly interpolating between the CMY values of the neighboring grid-aligned RGB points.

6 Guidance

The task can be broken up into several phases

6.1 Scanners and Cameras

6.1.1 Acquiring and Converting the Image

First, you should scan the provided Gretag-Macbeth color chart. You should try to get the color chart to be large and relatively aligned within the frame, although you can also align it by hand later. When scanning, you should scan in true-color and save as a TIFF file to avoid any lossy compression. For the scanners in the lab, you should see the CanoScan Toolbox icon on the desktop; if not, start it from the Start Menu. On the screen that pops up, press the ”Save” button, and on the following screen make sure ”Display the Scanner Driver” is checked and that the type of file saved is a TIFF file (you can also specify where the file will be saved to on this screen). When you press ”Scan” a third window should appear. Go to the Advanced tab, click on the preferences button, and in the Color Settings tab make sure that ”None” is selected. Press the preview button, and then adjust the boundary in the preview image to fully enclose the color chart. Once you’re ready, click Scan to save the file to disk.

To make it consistent with the photo data, which is also gamma corrected, we will leave gamma correction (see section 4) enabled when we scan (this is the 2.2 value in the Monitor Gamma settings below the Color Settings tab). Before we perform our linear fit, though, we will need to remove the gamma correction. This is because gamma correction turns linear data into nonlinear data, meaning that our linear fit will be less accurate. We’ll remove the gamma correction in MATLAB after we read in the data, but you should keep it in mind. In particular, if you are using your own scanner you need to make note of whether or not you need to correct the gamma later on – if you don’t know (because you didn’t change any options related to gamma) then you probably do.

As for camera images, sample camera images will be provided on the website. If you have your own camera that can output in RAW format, you are highly encouraged to use that instead; talk to one of the course staff for details on converting your RAW image files into something that can be used in MATLAB. Again, these images are gamma corrected as well, so you’ll need to remove it once you’ve loaded it into MATLAB.

6.1.2 Loading and Aligning the Image

MATLAB provides a command **imread()** which takes a filename of an image and reads it in as a matrix of size $[h, w, 3]$ for an image that is w pixels wide and h pixels high (note which dimensions correspond to the height and width – it may be the reverse of what you expect). To make sure that the data is read in correctly, you can use the **imshow()** command, which takes a matrix of the above format and displays it on the screen as an image. MATLAB will read the data as either an array of unsigned 8-bit integers (uint8). You can see the datatype in the workspace, so you should convert the array into an array of doubles using the **double()** function and divide each entry by $2^8 - 1 = 255$ so that the entries in each 3-vector are between 0 and 1.

If your image had gamma correction applied, we now need to remove it. In order to remove it, we do the inverse of the gamma operation described in section 4: **img = img .ˆ 2.2** where *img* is your image data.

If you take enough care when scanning or taking a picture of the chart, your image should be aligned enough to compute the color values programatically. For those of you who want to try using MATLAB’s fairly extensive image transformation, consult the separate document ”Notes on Image Alignment in MATLAB”.

6.1.3 Measuring the Sample Squares

Next, you need to determine the color value for each of the squares in your sample. For each square, you can take a small block of pixels (40×40 should be enough) and use **mean()** to compute the average color value for each sample. Because **mean()** takes the mean per-column, you probably want to use **reshape()** to reorganize your samples into $[(40 * 40) \times 3]$ (or whatever the size of your sample square ends up being) where each row is the RGB value of a pixel in your sample square. If you use **imshow()** to display your image, there is a data cursor in the top toolbar. When you select the cursor and click on the image, it will give you the xy coordinates as well as the color data for that pixel. If your image is aligned properly, using the data cursor to get the position of the center of the top-left square and the bottom-right square should allow you to write a for loop that will compute the average pixel value of all 24 squares programatically (without you having to manually find the center of each square using the data cursor).

The true values for the squares are available on the course website. Note that this file is ordered so that the first row is the RGB value for the brown square, the second is the value for the tan square, and so on. So, for instance, the orange square is row 7, the white square is row 19, and the black square is row 24.

Once you have the average pixel value of both your image and the true values of the color chart, you now have all you need to compute the $[3 \times 3]$ matrix that will transform your camera's or scanner's image into RGB values for the monitor.

6.2 Printer

6.3 Generating a uniform CMY grid

To begin with, you should generate an image consisting of squares uniformly spaced in the CMY color channels. As we have seen, images in MATLAB are just $[h \times w \times k]$ arrays, where k is the number of color channels, so we can just create an array of $[h \times w \times 4]$ and fill it with the correct CMY values for each pixel (set the fourth entry, which corresponds to blacK, to 0).

One strategy would be to create 100 pixel wide squares (so they are large enough to be distinguished) and have the C value vary along the width and M and Y vary separately along the height. If you were to take 6 sample points along each color channel, then you would have 6 squares along the width and 36 along the height, or a 600 pixel width and 3600 pixel height. You can also certainly come up with alternative schemes to make better use of the dimensions of a piece of paper.

MATLAB does not have any capability for displaying a CMYK image directly. Because of this, you should use the command **imwrite()** to write your image to a TIFF file (specify 'TIFF' as the file format). You can take this TIFF file to Upson B7 and open it in Photoshop CS2 to print on the color laser there. When opening in Photoshop, you should set the DPI to 150 (in Image — Image Size — Resolution) and also rescale the image so it will fit on one page (use Document Size in the Image Size panel, which allows you to specify the size in inches). To print, go to File — Print With Preview and in Color management make sure that "No Color Management" in Color Handling is selected – this instructs Photoshop to prevent the printer from doing its own color correction.

6.4 Computing the Printer Transformation

First you should take your printout of the CMY grid, scan it in, convert the colors from unsigned integers to double values between 0 and 1, correct for gamma if needed, and use your scanner calibration matrix to correct the colors. There is the possibility that this final step will produce color values outside of [0, 1], so you might want to clamp the values (set all entries less than 0 to 0 and all entries greater than 1 to 1). Now you can use the same machinery from before to determine the RGB value of each sample square. In general, these RGB values will not be in any particular order – what you have is a sparse sample of the nonlinear function $C_{cmy} = \text{rgbToCmy}(C_{rgb})$ that you are looking for, where C_{cmy} and C_{rgb} are three-vectors of CMY and RGB colors respectively. In order to make interpolation easier, we will now calculate our best guesses for the value of rgbToCmy(r, g, b) on a regularly sampled grid of RGB values. To do this, we need to interpolate over these nonuniformly spaced samples, and a good method is moving least squares. This method will be discussed in class, or you can consult the separate documentation, ”Notes on Local Linear Approximations to Functions”.

On the border, you will probably run into *gamut* issues. This is because the CMYK space and the RGB space do not overlap precisely, and so there are some colors in each space that are not representable as real colors in the other space. This is indicated by negative values for one of the components, or an inability to find enough sample points nearby to compute the moving least squares fit. For RGB points that are outside the gamut of CMYK, it is acceptable to produce white (remember, white in CMY is $[0, 0, 0]$).

Once you have your regular grid of RGB points and the CMY value at each point, then turning an arbitrary RGB value into the CMY color that your approximation claims will best match it is a matter of linearly interpolating among the values of the neighboring grid points. You can do this for a bunch of points (like, say, an image) by using **interp3()**, although you may need some manipulation to get your data in the correct format. You can also write your own function to do the linear interpolation instead.

7 Submission and FAQ

Submission will be through CMS. A FAQ page will be kept on the course website detailing any new questions and their answers brought to the attention of the course staff. Your submission should include all MATLAB code you used.

Your writeup should discuss how you went about computing and applying the calibrations, including any problems you encountered. You should also include the results of testing your system by both correcting your calibration images and another similar (but not identical) image. Include plots of the error residual, and discuss where you believe the error comes from and possible approaches to reduce it.

8 Starred Problems

1. Attempt to do something sensible to convert RGB into CMYK. Explain why you chose the method you did, and the advantages and disadvantages of your approach
2. Perform a quadratic fit of the data for printer calibration when you generate your regular grid of RGB-CMYK samples, instead of the linear fit. What happens to the error residuals?