Processing RAW Images in MATLAB

Rob Sumner

Department of Electrical Engineering, UC Santa Cruz

May 19, 2014

Abstract

This is an instructional document concerning the steps required to read and display the unprocessed sensor data stored in RAW photo formats. RAW photo files contain the raw sensor data from a digital camera; while it can be quite scientifically useful, raw data must generally be processed before it can be displayed. In this tutorial, the nature of the raw sensor data is explored. The scaling and color adjustments applied in the standard processing/display chain are discussed. Solutions for reading RAW image data into MATLAB are presented, as well as a standard workflow for achieving a viewable output.

1 Introduction

In the fields of image processing and computer vision, researchers are often indifferent to the origins of the images they work with. They simply design their algorithms to consider the image (typically an 8-bit intensity image or three-channel RGB image with 8 bits per channel) as a multivariate function, a stochastic field, or perhaps a graph of connected pixels.

Sometimes, however, it is important to truly relate an image to the light of the scene from which it was captured. For example, this is the case for any processing that models the behavior of the imaging sensor, such as some High Dynamic Range (HDR) approaches and scientific imaging (*e.g.*, astronomy). In such a case, it is vital to know the entire processing chain that was applied to an image after being captured. If possible, the best image to deal with is the sensor data straight from the camera, the *raw* data.

Access to the raw data is also of interest to anyone who wishes to work on the steps of the image processing flow that takes data from the sensor of the camera to a nice output image. For example, a researcher who works on algorithms for demosaicing Bayer pattern images will need to be able to access such data for real tests.

Though many devices only output a processed digital image, some cameras allow direct access to the information recorded by the imaging sensor itself. Typically, these are mid- to high-range DSLRs (Digital Single Lens Reflex cameras), which have the option to output RAW data files. In the future it may be possible to access the sensor data from other camera types (such as those on smart phones), in which case the following general principles will still apply (though the specific programming may be different).

'RAW' is a class of computer files which typically contain an uncompressed image containing the sensor pixel values as well as a large amount of meta-information about the image generated by the camera (the Exif data). RAW files themselves come in many proprietary file formats (Nikon's .NEF, Canon's .CR2, etc) and at least one common open format, .DNG, which stands for Digital Negative. The latter indicates how these files are supposed to be thought of by digital photographers: the master originals, repositories of all the captured information of the scene. RAW files are intentionally inscrutable data structures, but have been reverse engineered with some success[1] to gain access to the raw data inside.

For the rest of this paper, we use the capitalized term *RAW* when referring to an image file of one file format or another (*e.g.*, .CR2), whereas by a (lowercase) *raw image* we mean unprocessed pixel values directly output by the imaging sensor of a camera.

The 'Invisible' Processing of Images

When using commercial software to read and display a RAW file there are a number of processes that happen behind the scenes which are invisible to those who primarily work with the final product (the 8-bit-per-channel RGB JPG/TIF/etc.). In order to display the image on a screen, the raw image is being linearized, demosaiced, color corrected, and its pixel values are being non-linearly compressed to yield the image information most operating systems and display drivers expect. When working with raw sensor data images it is important to account for each of these steps in order to display your results correctly.

Document Layout

The goal of this document is to describe the steps needed to convert raw data into a viewable image and to attempt to give some explanation as to why they are required. This is done by first describing the nature of the raw sensor data itself, and the workflow that needs to be done to it. Then we refer to some free software which will be necessary for reading the RAW images. Finally, we present a direct tutorial for reading your RAW files into MATLAB. Code examples are included, and are written for clarity rather than efficiency.

2 The Nature of the Raw Sensor Data

Raw data from an image sensor obviously contains information about a scene, but it is not intrinsically recognizable to the human eye. It is a single channel intensity image, possibly with a non-zero minimum value to represent 'black', with integer values that contain 10-14 bits of data (for typical digital cameras), such as that shown in Figure 1. Rather than speaking about an intrinsic 'white' value, no values in the image will be above some maximum which represents the saturation point of the physical pixel CCD. The output may also be larger than the expected pixel dimensions of the camera, including a border of unexposed pixels to the left and above the meaningfully exposed ones.

Color Filter Array

Raw sensor data typically comes in the form of a Color Filter Array (CFA). This is an m-by-n array of pixels (where m and n are the dimensions of the sensor) where each pixel carries information about a single color channel: red, green, *or* blue. Since light falling on any given photosensor in the CCD is recorded as some number of electrons in a capacitor, it can only be saved as a scalar value; a single pixel cannot retain the 3-dimensional nature of observable light. CFAs offer a compromise where information about each of the three color channels are captured at different locations by means of spectrum-selective filters placed above each pixel. While you may only truly know one color value at any pixel location, you can cleverly interpolate the other two color values from nearby neighbors where those colors are known. This process is called *demosaicing*, and produces the m-by-n-by-3 array of RGB values at each pixel location we typically expect from a color digital image.
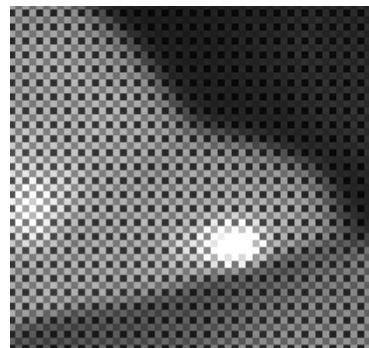

Figure 1: Detail of raw sensor data image. Integer valued, Min = 2047, Max = 15303.

The most common CFA pattern is the Bayer array, shown in Figure 2. There are twice as many pixels that represent green light in a Bayer array image because the human eye is more sensitive to variation in shades of green and it is more closely correlated with the perception of light intensity of a scene.

Note that though this pattern is fairly standard, sensors from different camera manufacturers may have a different "phase." That is, the "starting" color on the top left corner pixel may be different. The four options, typically referred to as 'RGGB', 'BGGR', 'GBRG', and 'GRBG', indicate the raster-wise orientation of the first "four cluster" of the image. It is necessary for a demosaicing algorithm to know the correct phase of the array.
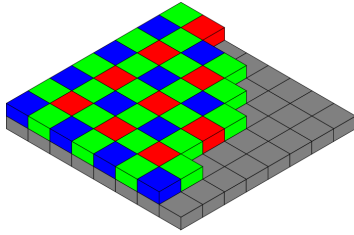
Figure 2: Bayer CFA layout. Each pixel represents *either* the red, blue, or green value of the light incident at the sensor, depending on arrangement in the array. To get all three elements at every location, *demosaicing* must be applied.

Color Channel Scaling

An unfortunate reality of color imaging is that there is no truth in color. Generally, one cannot look at a color image and know that it faithfully represents the color of the subject in question at the time the image was taken. As a severe simplification, one can think of an illuminating light as having an intrinsic color and the object it falls upon also having its own color. These interact, and the light which the camera or eye receives, the light reflected from the object, is an inextricable combination of the two.

What this leads to is that any object can look like any color, depending on the light illuminating it. What we need is a *reference point*, something we know should be a certain color (or more accurately, a certain chromaticity), so that we can adjust the R, G, B values of the pixel until it is that color. This compensates for the color of the illuminating light and reveals the "true" color of the object. Under the assumption that the same illuminant is lighting the entire scene, we can do the same balancing to every pixel in the image. This is the process of *white balancing*. Essentially, we find a pixel we know should be white (or gray), which we know should have RGB values all equal, and find the scaling factors necessary to make them all equal.

Thus the problem reduces to simply finding two scalars which represent the relative scaling of two of the color channels to the third. It is typical that the green channel is assumed to be the channel to which the others are compared. These scalars also take into account the relative sensitivities of the different color channels, *e.g.*, the fact that the green filter of the CFA is much more transparent than the red or blue filters. Thus, the green channel scalar is usually 1, while the others are often > 1.

Camera Color Spaces

The number of dimensions of perceivable colors is three. This may seem obvious to those who are familiar with this result (such as anyone who has ever heard that "any color can be made from a combination of red, green, and blue light"), but the reduction from an infinite-dimensional space to a three-dimensional one is far from trivial. An almost-vector space may be constructed for perceivable colors, leading to the use of many familiar linear algebra operations[1] on what we refer to as "the color space". The 'almost-' qualifier is necessary because one cannot *subtract* color since there is no such thing as negative light (yet). Fortunately, this still leaves us with a convex cone of color which is quite amenable to linear algebra.

In the language of linear algebra, we represent the color of a pixel with three coordinates (which we recognize as the R, G, and B values associated with the pixel). Importantly, these are the pixel's color coordinates *with respect to a particular basis*. What is often not evident is that the basis generated by the physical sensors of the digital camera is *not* the same as those of most displays. A full discussion of this topic is beyond the scope of this document, but we will state simply that we presume the displayable output space to be that defined as a common standard, sRGB[10].

Thus, though we obtain a familiar RGB image after white balancing and demosaicing a CFA, its colors are not those which the computer monitor expects. To correct for this, we must apply a linear transformation (*i.e.*, an appropriate change of basis matrix) to the RGB-vector of each pixel in the image. It is described later how this matrix can be found.

---

[1]In fact, the man who essentially invented linear algebra, Hermann Grassman, was also noted for exploring the laws pertaining to the perceptual equivalence of the color of various light sources. Treating colors as vectors is historically tied into the origins of vector spaces themselves!

Exif Metadata

In addition to the raw data from the imaging sensor of the digital camera, RAW files carry a large amount of metadata *about* the pixel values and the exposure itself. This information comes in the form of an abundance of Exif tags, which follow a standard `TagName:   Value` format. Like the sensor data, this information is obtusely tucked away in the RAW file and requires special software to retrieve.

The amount of meta-information retained in most RAW files is substantial. It typically contains information about the digital camera itself as well as info about the exposure that was captured, both of which are vital to computational photography. Relevant to this tutorial, this includes information that was mentioned in the previous sections, such as the white balance multiplier values and the black level offset. A small but indicative subset of retrievable information is shown below. Note that the metatags which are present in a file will vary between camera manufactureres.

- **Camera\file properties**: Camera model, preset white balance multipliers, image width and height, metering mode, creation time, flash usage, geotagging, etc.

- **Photographic properties**: ISO, focal length, shutter speed, aperture f-stop, hyperfocal distance, white balance multipliers measured when shot, etc.

The RAW Image Editing Workflow

In order to work with and display in MATLAB images originating from sensor data, we must take into account the aforementioned nature of the raw data. The workflow depicted in Figure 3 is a first order approximation of how to get a 'correct' displayable output image from the raw sensor data. Section 4 will cover how to implement this in MATLAB, but this can also be considered the general approach using any programming language.
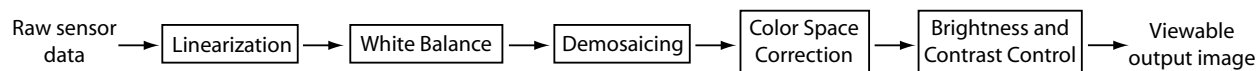
Figure 3: Proposed workflow for raw image processing.

3   RAW Utility Software

In order to work with raw images in MATLAB we must first use other pieces of software to crack the proprietary formats and get at the sweet image data found inside. The following are cost-free and very useful programs– only one *or* the other is necessary, so do not worry if you don't know how to compile C source code for the first option. Also recommended, though not necessary for the workflow described, is Phil Harvey's `ExifTool`[5], a powerful and scriptable open-source metadata reader.

Dave Coffin's `dcraw`

There is a fantastic piece of cross-platform, open-source software, written by Dave Coffin, called `dcraw` [3] (pronounced *dee-see-raw*). This is *the* open-source solution for reading dozens of different types of RAW files and outputting an easily read PPM or TIFF file. Many open-source image editing suites incorporate this program as their own RAW-reading routine. It can read files from hundreds of camera models and perform many standard processing steps to take a RAW file and generate an attractive output.

`dcraw` is a command-line program only, and is officially released only as a C source file (though pre-compiled executables can be found online, and even some MATLAB MEX-implemented versions). This makes it truly cross-platform, as it can be compiled for any system. It is powerful, comprehensive, cleverly and compactly coded, and *notoriously* poorly documented (read: 10k lines, $\sim$ 50 comments). Nonetheless, the workflow found in this document is largely informed by that in `dcraw`.

`dcraw` Functionality

dcraw is run from the command line and accepts a number of optional arguments as well the RAW files to be operated upon. For a full description, see its Unix-like manpage [2]. In part, it provides options over the processing of:

- White balance multipliers used

- Output file colorspace

- Demosaicing algorithm, if any

- Gamma correction applied

- Brightness control / repair options

- 8-bit or 16-bit output

- Image rotation

In this tutorial, we make use of the options that make dcraw access the image information from the RAW file but not process it in any meaningful way so that we may do so ourselves. A useful and informative tutorial of dcraw's full capabilities can be found courtesy of Guillermo Luijk[9].

Compiling `dcraw`

For the purposes described in this document, the dcraw source file is the only file necessary for compilation of the program; no non-standard supplemental libraries need to be linked, making compilation rather simple. Depending on your operating system and if you need dcraw to be fully functional or not, you may need to disable some options or change some defines. The following additional defines were the only changes necessary to compile dcraw using Microsoft Visual Studio 2008 (as a 32-bit binary on a 64-bit system) with no errors and all functionality required for this tutorial:

```
#define _CRT_SECURE_NO_WARNINGS
#pragma warning(disable:4146)
#define NODEPS
#define getc_unlocked _fgetc_nolock
#define fseeko _fseeki64
#define ftello _ftelli64
```

Adobe DNG Converter

The Adobe DNG Converter[7], unsurprisingly, converts files from almost every proprietary RAW format to the DNG format. This file format, which stands for 'Digital Negative', is open and non-proprietary (though originally proposed by Adobe). It is based on the TIFF file format, and retains a plethora of Exif metadata fields from the original RAW files, though sometimes with different names.

The DNG Converter is free of charge, does not prompt for updating constantly, and is a fairly useful tool to have on hand. Unfortunately, it is only available for Windows and Mac OS X systems.

For our purposes, we must do a slight configuration after downloading the DNG Converter. After opening Adobe DNG Converter, click on Change Preferences and in the window that opens, use the drop-down menu to create a Custom Compatibility. Make sure the 'Uncompressed' box is checked in this custom combatibility mode and the 'Linear (demosaiced)' box is unchecked. 'Backward Version' can be whatever you like.

4   RAW to MATLAB Tutorial

Next, we present a step-by-step guide on how to read a RAW image into MATLAB and process the raw sensor information into a correctly-displayable image. In an effort to be cross-platform, presented here are two alternative approaches to the same ends. The first involves the Adobe DNG Converter, which does a lot of the overhead of calculating and standardizing some parameters/Exif values. It is also unavailable for Linux, and the associated approach will not work for MATLAB versions older than r2011a. Thus the second approach, using `dcraw`, is also presented. **The DNG approach is simpler and recommended for those who can use it (Windows or Mac, MATLAB r2011a or newer).**

Both methods enact the same steps of the workflow, but draw their information from different sources. The specific directions involving the DNG approach are presented in a blue field. Specific directions for the `dcraw` approach are in red. Steps that are agnostic to the original file-read method are presented as normal.

Note: These directions have been written from experience with Canon's CR2 format, with files generated by a Rebel T3/1100D camera. Other proprietary RAW file types *should* work with this pipeline since a lot of the heavy lifting is done by comprehensive commercial software which should be familiar with most formats.

The DNG approach to reading RAW files into MATLAB is fairly simple and utilizes MATLAB's resources to make automation simple. The first and most important step is 'borrowed' from Steve Eddins at The Mathworks[4]. The process involves converting the RAW file to a DNG file first, and then exploits the fact that DNG is just a fancy form of TIFF. Note that this requires the TIFF class in MATLAB, which has only been included since version r2011a. Older versions will give an error.

> ### Option 1: DNG
>
> Note: `imfinfo('file.dng')` returns a structure which contains the Exif information associated with the DNG version of the RAW file. This is a useful structure to take a look at, and a large advantage of this approach is that the DNG Converter standardizes these tags[6]. Thus while this means some of the information will change tag names from those of the original RAW file type, they will no longer vary between different RAW file types, leading to a simpler approach below. Also, some necessary information about the image (such as black level and saturation level) which may not actually be present in the original RAW Exif information should be present in the DNG due to Adobe's tests with these cameras. And so, like many great engineers, we benefit from the hard work of others.

Using `dcraw` is slightly more complicated due to the non-standardization of information in each camera manufacturer's Exif data. Also, some values, such as the black level and saturdation level, are not even stored in many RAW file types, and must be looked up from a table or calculated. By using `ExifTool` and/or `dcraw`'s verbose output and information mode, we should be able to get the information we need.

> ### Option 2: `dcraw`
>
> First, run `dcraw` in verbose mode with 'as shot' white balance to run it through the entire processing chain and output some useful information; a reconnaissance run, as it were. Once `dcraw` has been compiled and is in your path, you can simply get this information (and a preliminary image) from the terminal command line by typing
>
> ```
> $ dcraw -v -w -T <raw_file_name>
> ```
>
> This should output some information along the lines of "`Scaling with darkness <black>, saturation <white>, and multipliers <r_scale> <g_scale> <b_scale> <g_scale>`" where integer numbers fill in the fields above. We will make use of these shortly.

4.1 Reading the CFA Image into MATLAB

Option 1: DNG

Assuming you have downloaded and configured the DNG Converter as described in Section 3, first convert your file into DNG form. Then the following code, with 'file.dng' replaced with your filename, will read it into a MATLAB array called `raw`, as well as creating a structure of metadata about the image, `meta_info`. The code also uses information from the metadata to crop the image to only the meaningful area.

```matlab
filename = 'file.dng';                        % Put file name here
warning off MATLAB:tifflib:TIFFReadDirectory:libraryWarning
t = Tiff(filename,'r');
offsets = getTag(t,'SubIFD');
setSubDirectory(t,offsets(1));
raw = read(t);                                % Create variable 'raw', the Bayer CFA data
close(t);
meta_info = imfinfo(filename);
% Crop to only valid pixels
x_origin = meta_info.SubIFDs{1}.ActiveArea(2)+1;        % +1 due to MATLAB indexing
width = meta_info.SubIFDs{1}.DefaultCropSize(1);
y_origin = meta_info.SubIFDs{1}.ActiveArea(1)+1;
height = meta_info.SubIFDs{1}.DefaultCropSize(2);
raw = double(raw(y_origin:y_origin+height-1,x_origin:x_origin+width-1));
```

Option 2: dcraw

To get the raw sensor data into MATLAB, first we use `dcraw` with the following options to output a 16bpp TIFF file. This will also overwrite the previously produced 'recon image.'

- -4 : writes linear 16-bit, unbrightened and un-gamma-corrected image, same as '-6 -W -g 1 1'

- -D : Foregoes demosaicing

- -T : Writes to TIFF file instead of PPM

Thus we output the simple sensor data from the command line with the following:

```
$ dcraw -4 -D -T <raw_file_name>
```

You can now read this file into MATLAB using `raw = double(imread('file.tiff'))`, which will yield the raw CFA information of the camera. The image will be slightly larger than the pixel dimensions quoted by the camera, but that is because `dcraw` does not discard some of the valid border pixels that most programs do.

4.2 Linearizing

The 2-D array `raw` is not yet a linear image. It is possible that the camera applied a non-linear transformation to the sensor data for storage purposes (*e.g.*, Nikon cameras). If so, the DNG metadata will contain a table under `meta_info.SubIFDs{1}.LinearizationTable`. You will need to map the values of the `raw` array through this look-up table to the full 10-14 bit values. If this tag is empty (as for Canon cameras), you do not need to worry about this step. If you are using the `dcraw` approach, the '-4' option will already have applied the linearization table so this step is not necessary.

Even if there is no non-linear compression to invert, the raw image might still have an offset and arbitrary scaling. Find the black level value and saturation level value as below and do an affine transformation to the pixels of the image to make it linear and normalized to the range [0,1]. Also, because of sensor noise, it is possible that there exist values in the array which are above the theoretical maximum value or below the black level. These need to be clipped, as follows.

Note: There may exist a different black level or saturation level for each of the four Bayer color channels. The code below assumes they are the same and uses just one. You may choose to be more precise.

<div align="center">Option 1: DNG</div>

The black level and saturation level values are stored in the DNG metadata and can be accessed as shown. If the values are stored non-linearly, undo that mapping.

```
>> if isfield(meta_info.SubIFDs{1},'LinearizationTable')
ltab=meta_info.SubIFDs{1}.LinearizationTable;
raw = ltab(raw+1);
end
>> black = meta_info.SubIFDs{1}.BlackLevel(1);
>> saturation = meta_info.SubIFDs{1}.WhiteLevel;
>> lin_bayer = (raw-black)/(saturation-black);
>> lin_bayer = max(0,min(lin_bayer,1));
```

<div align="center">Option 2: dcraw</div>

The black level and saturation level were found during the informational first run of dcraw.

```
>> black = 2047;                          % For Canon 1100D, from dcraw
>> saturation = 15000;
>> lin_bayer = (raw-black)/(saturation-black);
>> lin_bayer = max(0,min(lin_bayer,1));
```

4.3   White Balancing

Now we scale each color channel in the CFA by an appropriate amount to white balance the image. Since only the ratio of the three colors matters[2], we can arbitrarily set one channel's multiplier to 1; this is usually done for the green pixels. You may set the other two white balance multipliers to any value you want (*e.g.*, the Exif information for the original RAW file may contain standard multiplier values for different standard illuminants), but here we use the multipliers the camera calculated at the time of shooting.

Once the values are found, multiply every red-location pixel in the image by the red multiplier and every blue-location pixel by the blue multiplier. This can be done by dot-multiplication with a mask of these scalars, which can be easily created by a function similar to the following.

<div align="center">Option 1: DNG</div>

An array of the *inverses* of the multiplier values, for [R G B], is found in meta_info.AsShotNeutral. Thus we invert the values and then rescale them all so that the green multiplier is 1.

```
>> wb_multipliers = (meta_info.AsShotNeutral).^-1;
>> wb_multipliers = wb_multipliers/wb_multipliers(2);
>> mask = wbmask(size(lin_bayer,1),size(lin_bayer,2),wb_multipliers,'rggb');
>> balanced_bayer = lin_bayer .* mask;
```

---

[2]This claim discounts color distortions that happen due to saturation and clipping of values scaled too high. [9] contains a discussion on this effect.

```matlab
function colormask = wbmask(m,n,wbmults,align)
% COLORMASK = wbmask(M,N,WBMULTS,ALIGN)
%
% Makes a white-balance multiplicative mask for an image of size m-by-n
% with RGB while balance multipliers WBMULTS = [R_scale G_scale B_scale].
% ALIGN is string indicating Bayer arrangement: 'rggb','gbrg','grbg','bggr'

colormask = wbmults(2)*ones(m,n); %Initialize to all green values
switch align
    case 'rggb'
        colormask(1:2:end,1:2:end) = wbmults(1);     %r
        colormask(2:2:end,2:2:end) = wbmults(3);     %b
    case 'bggr'
        colormask(2:2:end,2:2:end) = wbmults(1);     %r
        colormask(1:2:end,1:2:end) = wbmults(3);     %b
    case 'grbg'
        colormask(1:2:end,2:2:end) = wbmults(1);     %r
        colormask(1:2:end,2:2:end) = wbmults(3);     %b
    case 'gbrg'
        colormask(2:2:end,1:2:end) = wbmults(1);     %r
        colormask(1:2:end,2:2:end) = wbmults(3);     %b
end
end
```

4.4 Demosaicing

Apply your favorite demosaicing algorithm (or MATLAB's built-in one) to generate the familiar 3-layer RGB image variable. Note that the built-in `demosaic()` function requires a `uint8` or `uint16` input. To get a meaningful integer image, scale the entire image so that the max value is 65535. Then scale back to 0-1 for the rest of the process.

```matlab
>> temp = uint16(balanced_bayer/max(balanced_bayer(:))*2^16);
>> lin_rgb = double(demosaic(temp,'rggb'))/2^16;
```

4.5 Color Space Conversion

The current RGB image is viewable with the standard MATLAB display functions. However, its pixels will not have coordinates in the correct RGB space that is expected by the operating system. As described in Section 2, any given pixel's RGB values, which represent a vector in the color basis defined by the camera's sensors, must be converted to some color basis which the monitor expects. This is done by a linear transformation, so we will need to apply a 3x3 matrix transformation to each of the pixels.

The correct matrix to apply can be difficult to find. `dcraw` itself uses matrices (gleaned from Adobe) which transform from the camera's color space to the XYZ color space, a common standard. Then the transformation from XYZ to the desired output space, *e.g.*, sRGB, can be applied. Better yet, these two transformations can be combined first and then applied once.

As an added complication, however, these matrices typically are defined in the direction of sRGB-to-XYZ and XYZ-to-camera color basis. Thus, the desired matrix must be constructed as follows:

$$A_{sRGB \leftarrow Cam} = (A_{Cam \leftarrow XYZ} \, A_{XYZ \leftarrow sRGB})^{-1}$$

One other necessary trick, as found in `dcraw`, is to first normalize the rows of the sRGB-to-Cam matrix so that each row sums to 1. Though it may seem arbitrary and somewhat ad hoc, we can see that this is necessary if we consider what will happen when a white pixel in the camera color space is transformed to a white pixel in the output space: we can argue that it should still be white because we have already applied white balance multipliers in order to make it so. Since white in both spaces is represented by the RGB coordinates $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T$, we see we must normalize the rows so that

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}_{Cam} = \begin{bmatrix} A_{Cam \leftarrow sRGB} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}_{sRGB}$$

The matrices used for the output-to-XYZ colorspace transformations can be found at Bruce Lindbloom's comprehensive website[8]. For convenience, the most commonly desired one, the matrix from sRGB space to XYZ space, is given here.

$$A_{XYZ \leftarrow sRGB} = \begin{bmatrix} 0.4124564 & 0.3575761 & 0.1804375 \\ 0.2126729 & 0.7151522 & 0.0721750 \\ 0.0193339 & 0.1191920 & 0.9503041 \end{bmatrix}$$

Option 1: DNG

You can find the entries of the XYZ-to-camera matrix in the `meta_info.ColorMatrix2` array. These entries fill the transformation matrix in a C row-wise manner, not MATLAB column-wise.

Option 2: `dcraw`

The entries for the XYZ-to-camera matrix (times 10,000) for your camera model can be found in the source code under the `adobe_coeff` function. These entries fill the transformation matrix row-wise.

The following function and example of its application show a possible way of applying this color-space transformation to an image.

```
function corrected = apply_cmatrix(im,cmatrix)
% CORRECTED = apply_cmatrix(IM,CMATRIX)
%
% Applies CMATRIX to RGB input IM. Finds the appropriate weighting of the
% old color planes to form the new color planes, equivalent to but much
% more efficient than applying a matrix transformation to each pixel.

if size(im,3)~=3
    error('Apply cmatrix to RGB image only.')
end

r = cmatrix(1,1)*im(:,:,1)+cmatrix(1,2)*im(:,:,2)+cmatrix(1,3)*im(:,:,3);
g = cmatrix(2,1)*im(:,:,1)+cmatrix(2,2)*im(:,:,2)+cmatrix(2,3)*im(:,:,3);
b = cmatrix(3,1)*im(:,:,1)+cmatrix(3,2)*im(:,:,2)+cmatrix(3,3)*im(:,:,3);

corrected = cat(3,r,g,b);
end
```

```
>> rgb2cam = xyz2cam * rgb2xyz;          % Assuming previously defined matrices
>> rgb2cam = rgb2cam ./ repmat(sum(rgb2cam,2),1,3);     % Normalize rows to 1
>> cam2rgb  = rgb2cam^-1;
>> lin_srgb = apply_cmatrix(lin_rgb, cam2rgb);
>> lin_srgb = max(0,min(lin_srgb,1));               % Always keep image clipped b/w 0-1
```

4.6   Brightness and Gamma Correction

We now have a 16-bit, RGB image that has been color corrected and exists in the right color space for display. However, it is still a linear image with values relating to what was sensed, which may not be in a range appropriate for being displayed. We can brighten the image by simply scaling it (adding a constant would just make it look gray), or something more complicated, *e.g.*, applying a non-linear transformation. Here we will do both, but be aware that the steps of this subsection are *highly subjective* and at this point we are just tweaking the image so it looks good. It is already 'correct' in some sense, but not necessarily 'pretty.' If you are unhappy with your output image after this tutorial, this is the first place to look.

As a *extremely* simple brightening measure, we can find the mean luminance of the image and then scale it so that the mean luminance is some more reasonable value. In the following lines, we (fairly arbitrarily) scale the image so that the mean luminance is 1/4 the maximum. For the photographically inclined, this is equivalent to scaling the image so that there are two stops of bright area detail. This is not extremely clever, but the code is simple.

```
>> grayim = rgb2gray(lin_srgb);
>> grayscale = 0.25/mean(grayim(:));
>> bright_srgb = min(1,lin_srgb*grayscale);
```

The image is still linear, which will almost certaintly not be the best for display (dark areas will appear too dark, etc). We will apply a "gamma correction" power function to this image as a simple way to fix this. Though the official sRGB compression actually uses a power function with $\gamma = \frac{1}{2.4}$ and a small linear toe region for the lowest values, this is often approximated by the following simple $\gamma = \frac{1}{2.2}$ compression. Note that in general you only want to apply such a function to an image that has been scaled to be in the range [0,1], which we have made sure our input is.

```
>> nl_srgb = bright_srgb.^(1/2.2);
```

Congratulations, you now have a color-corrected, displayable RGB image. It is real valued, ranges from 0-1, and thus is ready for direct display by `imshow()` Example images for each step of this process are shown in Figure 4.

5   Concluding Remarks

The previous sections have presented a simple approach for accessing in MATLAB the raw sensor data stored in a RAW file and then applying the basic transformations required to turn them into a displayable image. The code and explicit directions found in this document are self-sufficient in producing acceptable results, shown compared in Figure 5 to those produced by dcraw with default settings. This MATLAB code, as well as supplemental files, may also be found at [11].

Though we briefly justified these steps on account of the nature of the raw data, we should note that this is not the *definitive* way to implement this process. The process proposed here can be viewed as the bare-bones approach necessary to get a 'correctly displayed' image from the raw sensor data. For example, there are other ways to implement color balance and color space conversion; some intelligent methods combine the two into one step.

There are *certainly* better ways of adjusting the brightness and contrast than the simplistic method suggested above. There is a whole world of approaches to color, brightness, and contrast adjustment for creating visually appealing images (see *every resource on photography ever*). This guide is not intended to address such problems, but has aimed to explain the origins of the process so that one can start talking about these issues. Hopefully this document has done a little to expose the normally hidden process of what happens to our images so that we can enjoy them.
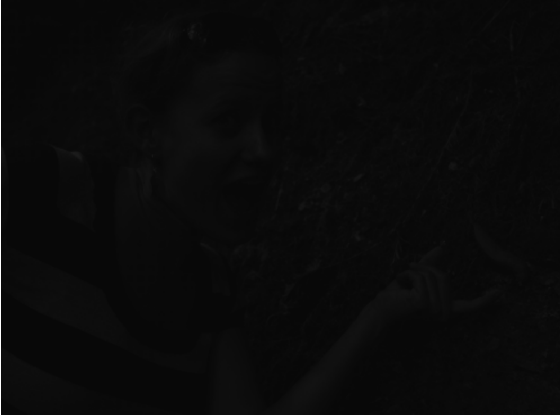
(a) Raw sensor data, `raw`, *m*-by-*n* array with integer values between `BlackLevel` and `WhiteLevel` (disregarding noise).

(b) Linearized data, `lin_bayer`, *m*-by-*n* array with real values between 0 and 1.

(c) Demosaiced image, `lin_rgb`, *m*-by-*n*-by-3.

(d) sRGB color space image, `lin_srgb`.

(e) Brightened linear sRGB image, `bright_srgb`.

(f) Gamma corrected image, `nl_srgb`.

Figure 4: Example images at every step of the proposed workflow. In order to keep the process transparent, these images are displayed as is, without any scaling for viewability. *E.g.*, the raw image (a) has meaningful values between 2047 and 15000, but none of those are very bright compared to the full `uint16` range of 0-65335, so the image is mostly black.

13

(a) `dcraw` output using `-w -T` options for TIFF output and measured white balance.

(b) Tutorial's proposed workflow output.

(c) Tutorial's output, without the Color Space Conversion step.

Figure 5: Comparison between `dcraw`'s output, the proposed workflow output, and the proposed workflow output neglecting the transformation to sRGB color space. The latter appears desaturated due to the RGB coordinates of each pixel relating to the Canon 1100D's sensor, not correct for display.

References

[1] Laurent Clevy. Inside the Canon RAW format version 2. http://lclevy.free.fr/cr2/, 2013.

[2] Dave Coffin. Manpage of dcraw. http://www.cybercom.net/~dcoffin/dcraw/dcraw.1.html, 2009.

[3] Dave Coffin. Decoding raw digital photos in Linux. http://www.cybercom.net/~dcoffin/dcraw/, 2013.

[4] Steve Eddins. Tips for reading a camera raw file into MATLAB. http://blogs.mathworks.com/steve/2011/03/08/tips-for-reading-a-camera-raw-file-into-matlab/, 2011.

[5] Phil Harvey. ExifTool by Phil Harvey. http://www.sno.phy.queensu.ca/~phil/exiftool/, 2013.

[6] Adobe Systems Inc. Digital Negative Specifications. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/products/photoshop/pdfs/dng_spec_1.4.0.0.pdf, 2012.

[7] Adobe Systems Inc. Camera raw, DNG : Downloads. http://www.adobe.com/products/photoshop/extend.displayTab2.html#downloads, 2013.

[8] Bruce Lindbloom. RGB/XYZ Matrices. http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html, 2011.

[9] Guillermo Luijk. Dcraw tutorial. http://www.guillermoluijk.com/tutorial/dcraw/index_en.htm, March 2010.

[10] M. Stokes, M. Anderson, S. Chandrasekar, and R. Motta. A Standard Default Color Space for the Internet - sRGB. http://www.w3.org/Graphics/Color/sRGB, 1996.

[11] Robert Sumner. Raw guide. http://users.soe.ucsc.edu/~rcsumner/rawguide/, 2013.