BBM 444 – Programming Assignment 1: Camera Pipeline

Due date: Tuesday, 8-3-2022, 11:59 PM.

Overview

The goal of this assignment is to get you familiarize with the standard camera pipeline of digital photography[1].You will build your own version of a very basic image processing pipeline (without denoising). You will use this to turn the RAW image into an image that can be displayed on a computer monitor or printed on paper. The Python packages required for this assignment are `numpy`, `scipy`, `skimage`, and `matplotlib`.

There is a "Hints and Information" section at the end of this document that is likely to help. It is highly recommended that you read that section in full before starting to work on the assignment.

Developing RAW images

For this problem, you will use the file `campus.nef` within the assignment data ZIP archive provided in the course webpage. This is a RAW image that was captured with a Nikon D3400 DSLR camera. As discussed in class, RAW images do not look like standard images before first undergoing a "development" process[2]. The developed image should look something like the image in Figure 1. The final result can vary greatly, depending on the choices you make in your implementation of the image processing pipeline.

Figure 1: One possible developed version of the RAW image provided with the assignment.

[1] Adapted from the programming assignment developed by Ioannis Gkioulekas for his computational photography class.

[2] The term "develop" comes from the analogy with the process of developing film into a photograph. As discussed in class, the same process is often called "renderin" the RAW images.

1. Implement a basic image processing pipeline (80 points)

RAW image conversion (5 points). The RAW image file cannot be read directly by `skimage`. You will first need to convert it into a `.tiff` file. You can do this conversion using a command-line tool called `dcraw` (`https://www.dechifro.org/dcraw/`). After you have downloaded and installed `dcraw`, you will first do a "reconnaissance run" to extract some information about the RAW image. For this, call `dcraw` as follows:

```
dcraw -4 -d -v -T <RAW_filename>
```

In the output, you will see (among other information) the following:

```
Scaling with darkness <black>, saturation <white>, and
multipliers <r_scale> <g_scale> <b_scale> <g_scale>
```

Make sure to record the integer numbers for `<black>` and `<white>`.

Calling `dcraw` as above will produce a `.tiff` file. Do not use this! Instead, delete the file, and call `dcraw` once more as follows (note the different flags):

```
dcraw -4 -D -T <RAW_filename>
```

This will produce a new `.tiff` file that you can use for the rest of this problem.

Python initials (5 points). We will be using `skimage` function `imread` for reading images. Originally, it will be in the form of a `numpy` 2D-array of unsigned integers. Check and report how many bits per pixel the image has, its width, and its height. Then, convert the image into a double-precision array. (See `numpy` functions `shape`, `dtype` and `astype`.)

Linearization (5 points). The 2D-array is not yet a linear image. It is possible that it has an offset due to dark noise, and saturated pixels due to over-exposure. Additionally, even though the original data-type of the image was 16 bits, only 14 of those have meaningful information, meaning that the maximum possible value for pixels is 4095 (that's $2^{12} - 1$). For the provided image file, you can assume the following: All pixels with a value lower than `<black>` correspond to pixels that would be black, were it not for noise. All pixels with a value above `<white>` are saturated. The values `<black>` for the black level and `<white>` for saturation are those you recorded earlier from the reconnaissance run of `dcraw`.

Convert the image into a linear array within the range [0, 1]. Do this by applying a linear transformation (shift and scale) to the image, so that the value `<black>` is mapped to 0, and the value `<white>` is mapped to 1. Then, clip negative values to 0, and values greater than 1 to 1. (See `numpy` function `clip`.)

Identifying the correct Bayer pattern (20 points). As we discussed in class, most cameras use the Bayer pattern in order to capture color. The same is true for the camera used to capture our RAW image.

We do not know, however, the exact shift of the Bayer pattern. If you look at the top-left 2×2 square of the image file, it can correspond to any of four possible red-green-blue patterns, as shown in Figure 2.

Think of a way for identifying which Bayer pattern applies to your image file, and report the one you identified. It will likely be easier to identify the correct Bayer pattern after performing white balancing.

White balancing (10 points). After identifying the correct Bayer pattern, we want to perform white balancing. Implement both the white world and gray world automatic white balancing algorithms, as discussed in class. Additionally, implement a third white balancing algorithm, where

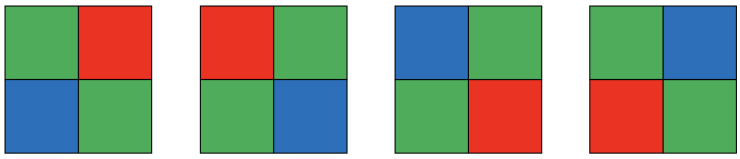

Figure 2: From left to right: `'grbg'`, `'rggb'`, `'bggr'`, `'gbrg'`.

you multiply the red, green, and blue channels with the `<r_scale>`, `<g_scale>`, and `<b_scale>` values you recorded earlier from the reconnaissance run of `dcraw`. These values correspond to the white balancing presets used by the camera. After completing the entire developing process, check what the image looks like when using each of the three white balancing algorithms, and decide which one you like best. (See `numpy` functions `max` and `mean`.)

Demosaicing (10 points). Once white balancing is done, it is time to demosaic the image. Use bilinear interpolation for demosaicing, as discussed in class. Do not implement bilinear interpolation on your own! Instead, use `scipy`'s built-in `interp2d` function.

Color space correction (10 points). You now have an RGB image that is viewable with the standard `matplotlib` display functions. However, the image color does not look quite right. This is because the image pixels have RGB coordinates in the color space determined by the camera's spectral sensitivity functions, and not in the "standard" color space that image viewing software expects. We use scary quotes for "standard" because what color space the image viewing software actually assumes, and what color space it should assume, are neither the same nor trivial to determine. A good default choice is to use the linear sRGB color space [1].

To transform your image to the linear sRGB color space, implement a function that applies the following linear transformation to the RGB vector $[R_{\text{cam}}, G_{\text{cam}}, B_{\text{cam}}]^\top$ of each pixel of your demosaiced image:

$$\begin{bmatrix} R_{\text{sRGB}} \\ G_{\text{sRGB}} \\ B_{\text{sRGB}} \end{bmatrix} = (\mathbf{M}_{sRGB \to cam})^{-1} \cdot \begin{bmatrix} R_{\text{cam}} \\ G_{\text{cam}} \\ B_{\text{cam}} \end{bmatrix}. \tag{1}$$

The 3×3 matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ transforms a 3×1 color vector from the sRGB color space to the camera-specific RGB color space. We use its inverse to apply the transformation in the reverse way.

Computing the matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ involves a sequence of steps. Here, we describe these steps at a high level, with just enough detail for you to be able to implement them. We write the matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ as:

$$\mathbf{M}_{\text{sRGB}\to\text{cam}} = \mathbf{M}_{\text{XYZ}\to\text{cam}} \cdot \mathbf{M}_{\text{sRGB}\to\text{XYZ}}. \tag{2}$$

As the notation suggests, the 3×3 matrix $\mathbf{M}_{\text{sRGB}\to\text{XYZ}}$ transforms a 3×1 color vector from the sRGB to the XYZ color space; and analogously, the 3×3 matrix $\mathbf{M}_{\text{XYZ}\to\text{cam}}$ transforms a 3×1 color vector from the XYZ to the camera-specific RGB color space. Their product, then, implements the transformation from the sRGB to the camera-specific RGB color space, as we want. The XYZ color space is an intermediate reference color space that color management systems use to accurately perform color space transformations.

The *sRGB standard* [1] provides the following values that you can use in your implementation:

$$\mathbf{M}_{\text{sRGB}\to\text{XYZ}} = \begin{bmatrix} 0.4124564 & 0.3575761 & 0.1804375 \\ 0.2126729 & 0.7151522 & 0.0721750 \\ 0.0193339 & 0.1191920 & 0.9503041 \end{bmatrix}. \tag{3}$$

You can find the values of the camera-specific matrix $\mathbf{M}_{\text{XYZ}\to\text{cam}}$ in the raw code of `dcraw`, under the function `adobe_coeff`. Look for the entry corresponding to the camera used to capture the RAW image you are developing. Note that the `dcraw` code provides the matrix values multiplied by 10,000, so make sure to adjust the matrix you use in your implementation accordingly. Additionally, the `dcraw` code shows the matrix as a 1×9 vector, which you should rearrange to a 3×3 matrix assuming row-major ordering.

After computing the matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ as in Equation (2), normalize it so that its rows sum to 1. You need to do this so that you do not have to perform white balancing a second time. Finally, plug the normalized matrix into Equation (1) to correct the color space of your image.

Brightness adjustment and gamma encoding (10 points). You now have a 16-bit, full-resolution, linear RGB image. Because of the scaling you did at the start of the assignment, the image pixel values may not be in a range appropriate for display. Moreover, the image is not yet gamma-encoded. As we discussed in class, this means that when you display the image, it will appear very dark.

Brighten the image by linearly scaling it. Select the scale to set the post-brightening mean grayscale intensity to some value in the range [0, 1]. After the scaling, clip all intensity values greater than 1 to 1. (See `skimage` function `rgb2gray` for converting the image to grayscale, and `numpy` function clip for clipping.) What the best value is is a highly subjective judgment, so you should experiment with many percentages and report and use the one that looks best to you. For the photographically inclined, selecting a post-brightening mean value of 0.25 is equivalent to scaling the image so that there are two stops of bright area detail.

You are now one step away from having an image that can be properly displayed. The last thing you need to take care of is tone reproduction (also known as gamma encoding). For this, implement the following non-linear transformation, then apply it to the image:

$$C_{\text{non-linear}} = \begin{cases} 12.92 \cdot C_{\text{linear}}, & C_{\text{linear}} \leq 0.0031308, \\ (1 + 0.055) \cdot C_{\text{linear}}^{\frac{1}{2.4}} - 0.055, & C_{\text{linear}} > 0.0031308, \end{cases} \tag{4}$$

where $C = \{R, G, B\}$ is each of the red, green, and blue channels. This odd-looking function is not arbitrary: it corresponds to the tone reproduction curve specified in the sRGB standard [1], which also specifies the sRGB color space you used earlier. It is a good default choice if the camera's true tone reproduction curve is unknown.

Compression (5 points). Finally, it is time to store the image, either with or without compression. Use `skimage` function `imsave` to store the image in `.png` format (no compression), and also in `.jpeg` format with the `quality` parameter set to 95. This setting determines the amount of compression. Can you tell the difference between the two files? The compression ratio is the ratio between the size of the uncompressed file (in bytes) and the size of the compressed file (in bytes). What is the compression ratio?

By changing the JPEG quality settings, determine the lowest setting for which the compressed image is indistinguishable from the original. What is the compression ratio?

2. Perform manual white balancing (10 points)

As we discussed in class, one way to do manual white balancing is by: 1) selecting some patch in the scene that you expect to be white; and 2) normalizing all three channels using weights that make the red, green, and blue channel values of this patch be equal. Implement this algorithm, and experiment with different patches in the scene. Show the corresponding results, and discuss which patches work best. (See `matplotlib` function `ginput` for interactively recording image coordinates).

3. Learn to use `dcraw` (10 points)

Besides converting RAW files to `.tiff`, `dcraw` provides options to emulate all steps in the image processing pipeline, including steps that are camera-dependent. Read through `dcraw`'s documentation, and figure out what the correct flags are in order for `dcraw` to perform all the image processing pipeline steps you implemented in Python. Make sure to report the exact `dcraw` flags you used.

Note that `dcraw` outputs images in `.ppm` format and does not have an option to use the `.png` format. You will need to use an image editor to convert the image format. The command-line utility `pnmtopng`, available in the Netpbm toolkit (`http://netpbm.sourceforge.net/`), is an easy way to do the conversion. Alternatively, ImageMagick (`https://www.imagemagick.org/`) also does command-line format conversions.

You should now have (at least) two developed versions of the original RAW image: One (or more) you produced using your implementation of the image processing pipeline, and another you produced using `dcraw`. One more developed image is available as the file `campus.jpg` included within the assignment data ZIP archive provided in the course webpage. This is the developed image produced by the camera's own image processing pipeline, alongside the RAW image you have been using. Compare the three developed images, and explain any differences between them. Which of the three images do you like best?

What to Hand In

Your submitted solution should include the following:

- A PDF report explaining what you did, including answers to all questions asked throughout Problems 1-3. The report should include any figures and intermediate results that you think may help. Make sure to include explanations of any issues that you may have run into that prevented you from fully solving the assignment, as this will help us determine partial credit. The report should also explain any `.png` files you include in your solution (see below).
- The codes you will submit should be well commented and in the form of a Jupyter notebook.
- For Problem 1: At least two `.png` files, showing the final images you created with the two different types of automatic white balancing. You can also include `.png` files for various experimental settings if you want (e.g., different brightness values).
- For Problem 2: `.png` files, showing the results of manual white balancing for different patch selections.
- For Problem 3: The `.png` file produced by dcraw.

You should prepare a ZIP file named `name-surname(s)-assgn1.zip` containing your solution / implementation `assgn1.ipynb` and your report `assgn1-report.pdf`, and submit it via email to `erkut@cs.hacettepe.edu.tr`.

Late policy

You may use up to five *extension* days (in total) over the course of the semester for the programming assignments. Late submission will not be allowed.

Academic Integrity

All work on assignments must be done individually unless stated otherwise. You are encouraged to discuss with your other classmates about the given assignments, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in the pseudocode) will not be tolerated. In short, turning in someone else's work, in whole or in part, as your own will be considered as a violation of academic integrity. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

Hints and Information

- Make sure to download and install the latest version (version 9.28) of `dcraw`. In particular, the default version that comes in older Windows versions does not support the cameras used in this class, and will produce results with a strong purple hue.
- Make sure you have at least versions 1.19.1 for `numpy`, 1.5.0 for `scipy`, 0.16.2 for `skimage`, and 3.3.1 for `matplotlib`. Newer versions should be fine.
- The package `matplotlib` is included with the `skimage` installation and is very useful for visualizing intermediate results and creating figures for the report. For example, the following code reads in two images and creates a single figure displaying them side by side:

```
import matplotlib.pyplot as plt
from skimage import io
# read two images from current directory
im1 = io.imread('image1.tiff')
im2 = io.imread('image2.tiff')

# display both images in a 1x2 grid
fig = plt.figure() # create a new figure
fig.add_subplot(1, 2, 1)
plt.imshow(im1) # draw first image
fig.add_subplot(1, 2, 2) # draw second image
plt.imshow(im2)
plt.savefig('output.png') # saves current figure as a PNG file
plt.show() # displays figure
```

  Note that figures can also be saved by clicking on the save icon in the display window.

  When displaying grayscale (single-channel) images, `imshow` maps pixel values in the [0,1] range to colors from a default color map. If you want to display your image in grayscale, you should use the following instead:

```
# read a single-channel image
imgr = io.imread('image_gray.tiff')
plt.imshow(imgr), cmap='gray')
plt.show()
```

  You will need this in several places in this assignment, as many of your images will be grayscale.

- You can access subsets of `numpy` arrays using the standard Python slice syntax `i:j:k`, where `i` is the start index, `j` is the end index, and `k` is the step size. The following example, given an original image im, creates three other images, each with only one-fourth the pixels of the originals. The pixels of each of the corresponding sub-images are shown in Figure 3. You can also use the `numpy` function `dstack` to combine these three images into a single 3-channel RGB image.

```
import numpy as np
# create three sub-images of im as shown in figure below
im1 = im[0::2, 0::2]
im2 = im[0::2, 1::2]
im3 = im[1::2, 0::2]

# combine the above images into an RGB image, such that im1 is the red,
# im2 is the green, and im3 is the blue channel
im_rgb = np.dstack((im1, im2, im3));
```

Im1	Im2	Im1	Im2	Im1	Im2
Im3		Im3		Im3	
Im1	Im2	Im1	Im2	Im1	Im2
Im3		Im3		Im3	
Im1	Im2	Im1	Im2	Im1	Im2
Im3		Im3		Im3	

- You will find it very helpful to display intermediate results while you are implementing the image processing pipeline. However, before you apply brightening and gamma encoding, you will find that displayed images will look completely black. To be able to see something more meaningful, we recom- mend scaling and clipping intermediate image `im_intermediate` before displaying with `matplotlib`.

```
plt.imshow(np.clip(im_intermediate*5, 0, 1), cmap='gray')
plt.show()
```

  Figure 3 shows what you should see with this command after the linearization step.

  Additionally, the colors of intermediate images will be very off (e.g., have a very strong green hue), even if you are doing everything correctly. You will not get reasonable colors until after you have performed at least white balancing. This will come into play when you are trying to determine the Bayer pattern.

References

[1] International Electrotechnical Commission and others. IEC 61966-2-1:1999. *Multimedia systems and equipment–Colour measurements and management–Part 2-1: Colour management–Default RGB colour space–sRGB*, 1999.

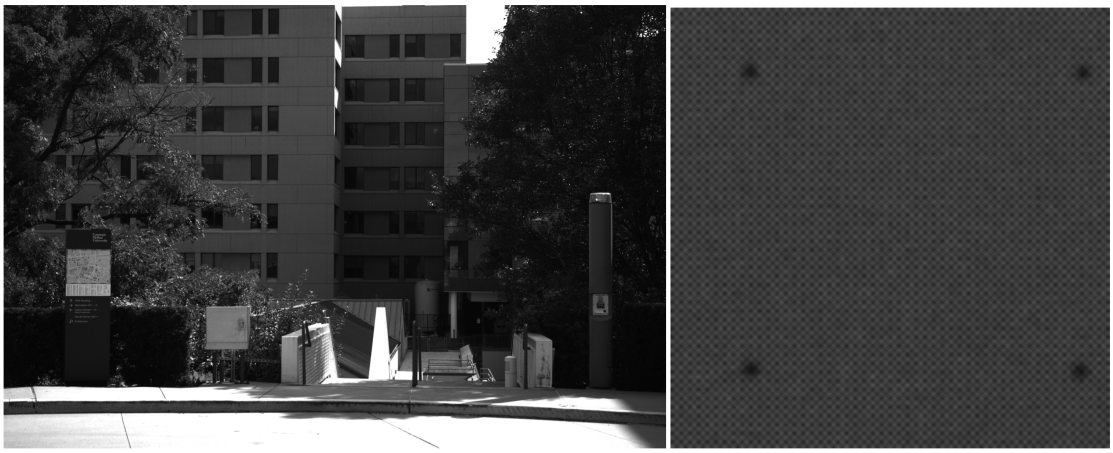

Figure 3: Left: The linear RAW image (brightness increased by 5). Right: Crop showing the Bayer pattern.