# BBM 444 – Programming Assignment 1: Camera Pipeline

Due date: Monday, 16/03/2026, 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 `rawpy`, `numpy`, `PIL`, `scipy`, `skimage`, and `matplotlib`.

You are provided with a template Jupyter notebook that you need to fill in with your answers, code, and comments, which will be evaluated as both your source code and report. The provided template is to guide you through steps as modularly as possible. However, you are not obliged to follow each cell strictly (though it might be beneficial for you if you did -at least to some extent). That is, you can add as many functions, code blocks, and markdown cells as you want as long as the report is structured and traceable. Additionally, the code blocks, functions, variables, etc. need to be well-commented. Further, the outputs need to be reproducible, for which you also need to submit the .tiff image file you will have generated (The details are in the "What to Hand in Section"). For implementation details, please refer to this homework text, as the comments in the Jupyter notebook are merely for guidance and hence really brief and abrupt, which may be incomplete.

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 on the course web page. This is a RAW image that was captured with a Fujifilm X100S 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.

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

**Python initials (5 points).** We will be using `rawpy.imread` function 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`.)

Extract the metadata using rawpy functions:

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

[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.

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

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

**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. And 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.

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 Sensor Pattern (10 points).

Many digital cameras use the Bayer filter to capture colour information at the photosite level, with colour filters repeating in a $2 \times 2$ pattern. A drawback of this regular repetition is that fine details in the scene can produce moiré patterns, which are typically suppressed by an optical low-pass filter placed in front of the sensor. While effective, this filter introduces a controlled blur that reduces overall image sharpness.

Fujifilm's X-Trans sensor addresses this differently. Its colour filters are arranged in a $6 \times 6$ pseudo-random tile, as shown in Figure 2, which breaks up the regularity that causes moiré. This allows X-Trans cameras to omit the optical low-pass filter entirely, recovering the sharpness that Bayer-based sensors sacrifice. Unlike Bayer, the X-Trans pattern has no orientation variants — the `raw_pattern` attribute read from the file uniquely defines the tile, so no pattern identification step is required.

Think of a way for identifying how to identify the colors looking at the metadata and the raw

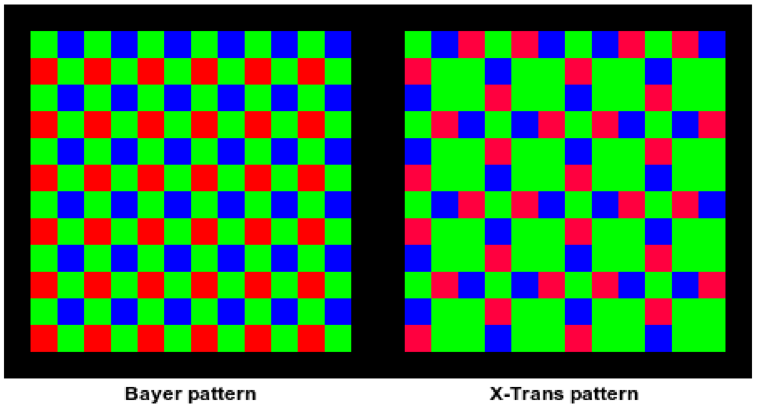

Figure 2: Bayer Pattern vs X-Trans Pattern

image file, and report.

**White balancing (10 points).** Implement both the white world and gray world automatic white balancing algorithms, as discussed in class. Additionally, implement a third white balancing algorithm, where you multiply the red, green, and blue channels with the `<r_scale>`, `<g_scale>`, and `<b_scale>` values you recorded earlier using the `rawpy` functions. 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 (20 points).** Once white balancing is done, it is time to demosaic the image. Use edge-aware interpolation with the residual trick for demosaicing, as discussed in class. You may use `scipy`'s built-in `RegularGridInterpolator` or `convolve` function.

In edge-aware interpolation, instead of interpolating blindly in all directions, we (1) compute gradients to detect edge direction (horizontal vs. vertical), (2) interpolate along the edge rather than across it, and (3) weight neighbors by their similarity to the center pixel. We will use *residual-based correction* as the gradient transfer mechanism.

The residual trick converts a hard problem—interpolating across edges—into an easier one: interpolating a smooth signal. Although algebraically equivalent to interpolating $R$ directly, routing through the smooth residual guides the interpolation through a signal with edges removed, so that even a simple kernel behaves well where it would otherwise blur. Rather than interpolating $R$ and $B$ directly (where edges cause artifacts), we exploit the fact that color difference channels are smooth across edges even when luminance is not. Formally, an edge appears in all three channels simultaneously, so $R - G$ and $B - G$ vary slowly even where $R$, $G$, and $B$ individually change sharply.

The demosaicing pipeline proceeds in four steps:

1. **Interpolate the green channel first.** Green has twice the spatial samples of red and blue in the X-Trans pattern, making it the best-resolved channel and a natural starting point. Gradients should be computed directly on the raw mosaic rather than on an interpolated version — since an edge manifests as an intensity discontinuity across all channels simultaneously, the raw pixel values carry sufficient luminance variation to reliably estimate edge direction. Compute horizontal and vertical gradients using the first-order finite difference

kernels:

$$\mathbf{k}_h = \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \qquad \mathbf{k}_v = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}. \tag{1}$$

At each $R$ or $B$ pixel location, interpolate green by averaging the two nearest neighbors in the direction with the smaller gradient magnitude.

2. **Compute color difference channels.** At every pixel where the true value is known, compute:

$$D_R = R - G, \tag{2}$$
$$D_B = B - G. \tag{3}$$

These difference images are considerably smoother than $R$ or $B$ alone, as the high-frequency edge structure largely cancels out.

3. **Interpolate the difference channels.** Because $D_R$ and $D_B$ are smooth, bilinear interpolation introduces minimal edge artifacts. For improved results, a guided filter using $G$ as the guidance image may be used instead.

4. **Reconstruct red and blue.** The full-resolution red and blue channels are recovered as:

$$\hat{R} = G_{\text{full}} + \widetilde{D}_R, \tag{4}$$
$$\hat{B} = G_{\text{full}} + \widetilde{D}_B, \tag{5}$$

where $\widetilde{D}_R$ and $\widetilde{D}_B$ denote the interpolated difference channels.

**Color space correction (15 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:

$$\begin{bmatrix} R_{\text{sRGB}} \\ G_{\text{sRGB}} \\ B_{\text{sRGB}} \end{bmatrix} = (\mathbf{M}_{sRGB\to cam})^{-1} \cdot \begin{bmatrix} R_{\text{cam}} \\ G_{\text{cam}} \\ B_{\text{cam}} \end{bmatrix}. \tag{6}$$

The $3 \times 3$ matrix $\mathbf{M}_{\text{sRGB}\to\text{cam}}$ transforms a $3 \times 1$ 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:

$$\mathbf{M}_{\text{sRGB}\to\text{cam}} = \mathbf{M}_{\text{XYZ}\to\text{cam}} \cdot \mathbf{M}_{\text{sRGB}\to\text{XYZ}}. \tag{7}$$

As the notation suggests, the $3 \times 3$ matrix $\mathbf{M}_{\text{sRGB}\to\text{XYZ}}$ transforms a $3 \times 1$ color vector from the sRGB to the XYZ color space; and analogously, the $3\times3$ matrix $\mathbf{M}_{\text{XYZ}\to\text{cam}}$ transforms a $3\times1$ 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:

$$\mathbf{M}_{\text{sRGB}\to\text{XYZ}} = \begin{bmatrix} 0.4124564 & 0.3575761 & 0.1804375 \\ 0.2126729 & 0.7151522 & 0.0721750 \\ 0.0193339 & 0.1191920 & 0.9503041 \end{bmatrix}. \tag{8}$$

You can find the values of the camera-specific matrix $\mathbf{M}_{\text{XYZ}\to\text{cam}}$ from the `adobe_coeff` table in the `dcraw` source code, available at `https://www.dechifro.org/dcraw/dcraw.c`, under the entry corresponding to your camera model.
These 9 values from the link above form the matrix $\mathbf{M}_{\text{XYZ}\to\text{cam}}$, provided as a $1 \times 9$ vector in row-major order and scaled by 10,000. Reshape it to a $3 \times 3$ matrix and divide by 10,000 to obtain the values used in your implementation. Note that the `color_matrix` attribute returned by `rawpy` contains all zeros for this camera, so this manual lookup is necessary.
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:

$$C_{\text{non-linear}} = \begin{cases} 12.92 \cdot C_{\text{linear}}, & C_{\text{linear}} \leq 0.0031308, \\ (1 + 0.055) \cdot C_{\text{linear}}^{\frac{1}{2.4}} - 0.055, & C_{\text{linear}} > 0.0031308, \end{cases} \tag{9}$$

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 `PIL` library 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. Explore `rawpy` Postprocessing Options (10 points)

Besides the manual pipeline you implemented, `rawpy` exposes a rich set of postprocessing parameters, which internally invokes `LibRaw`'s development pipeline. Read through the `rawpy` documentation and explore the following aspects:

For each parameter group, report the values you tested and support your observations with visual comparisons. Using the combination you judge to produce the best result, report the exact parameter values used.

You should now have at least three developed versions of the RAW image: one from your manual pipeline, one from `rawpy`'s `postprocess()`, and the reference `sample.jpg` developed by the camera's own pipeline. Compare all three and discuss differences in color accuracy, sharpness, noise, and overall appearance. Which do you prefer, and why?

## What to Hand In

Your submitted solution should include the following:

- The filled-in Jupyter Notebook as both your source code and report. The notebook should include (1) markdown cells reporting your written answers alongside any relevant figures and images and (2) well-commented code cells with reproducible results.
- Any figure you want to appear in any markdown cell in the notebook.
- The `.png` file produced by your manual pipeline, named `manual_result.png`.
- The `.png` file postprocessed by `rawpy`, named `rawpy_postprocessed.png`.

You should prepare a ZIP file named `b<studentnumber>.zip` containing the files stated above, and submit it to `submit.cs.hacettepe.edu.tr`, where you will be assigned a submission. The file hierarchy is up to you as long as your Jupyter notebook works fine.

## 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 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

- Download and install the latest version of `LibRaw` from `https://www.libraw.org/download`, then install the Python binding by running `pip install rawpy`. To verify the installation, check both the `rawpy` and bundled `LibRaw` versions by inspecting `rawpy.__version__` and `rawpy.libraw_version`.
- Make sure you have at least versions 1.19.1 for `numpy`, 1.5.0 for `scipy`, 0.16.2 for `skimage`, 3.3.1 for `matplotlib`, and 0.10.0 for `rawpy`. Newer versions should be fine.
- The package `matplotlib` is very useful for visualizing intermediate results and creating figures for the report. RAW images should be read using `rawpy` as described in Section 1, after which the resulting `numpy` arrays can be displayed and saved directly with `matplotlib`. For example, the following code displays two images side by side and saves the figure:

  ```
  import matplotlib.pyplot as plt
  import numpy as np

  # display two numpy arrays in a 1x2 grid
  fig = plt.figure()
  fig.add_subplot(1, 2, 1)
  plt.imshow(im1, cmap='gray')  # draw first image
  fig.add_subplot(1, 2, 2)
  plt.imshow(im2, cmap='gray')  # draw second image
  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:

  ```
  # display a single-channel (grayscale) numpy array
  plt.imshow(imgr, cmap='gray')
  plt.show()
  ```

  You will need this in several places in this assignment, as many intermediate results — such as the raw mosaic after linearization and the individual $R$, $G$, $B$ channels during demosaicing — will be single-channel grayscale arrays.

- 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. Unlike a standard Bayer sensor, the X-Trans pattern repeats on a $6 \times 6$ tile rather than a $2 \times 2$ tile, so channel extraction requires iterating over the 36 positions of the tile and using a stride of 6. For a position $(i, j)$ within the tile, the corresponding pixels across the full image are accessed as `im[i::6, j::6]`. The color at each tile position is determined by the `raw_pattern` attribute of the `RawPy` object. You can use the `numpy` function `dstack` to combine the reconstructed channels into a single 3-channel RGB image. Figure 3 shows what you should see 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.

- The comments provided in the Jupyter notebook template are merely for guidance. Refer to this homework text for any implementation detail.

- Please be careful about the order of runs of cells. Doing the homework, it is likely that you will be running the cells in different orders, however, they will be evaluated in the order they appear in the notebook. Hence, please try running the cells in this order before submission to make sure they work.

- You need to specify the paths correctly for any I/O operation, be it for rendering a figure in a markdown cell or reading an image in a code cell, as they are to be rerun for evaluation. That is, your outputs should be reproducible.

- There will be a lot of reusing the same functionality, hence implementing the algorithms you are asked with functions (and making appropriate calls when necessary, rather than just code blocks ) is likely to be beneficial.

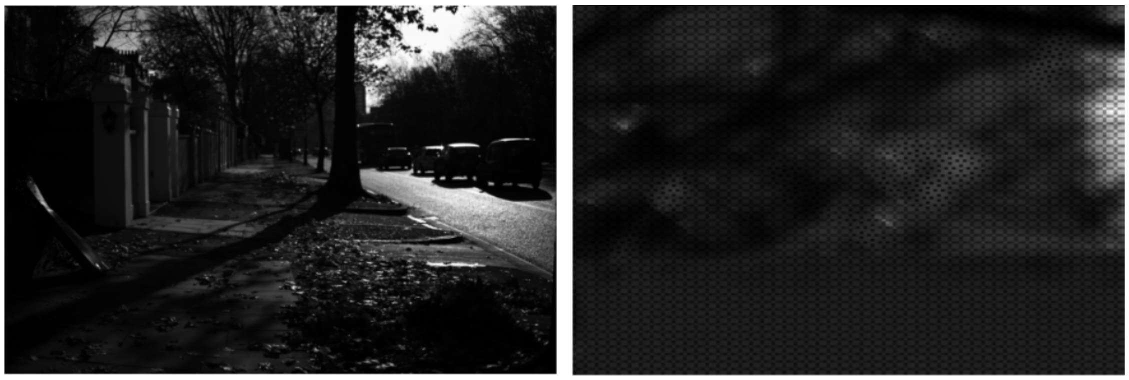

Figure 3: Left: The linear RAW image. Right: Crop showing the Bayer pattern.

## 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.