Homework assignment 1

20254251 Computational Imaging, Fall 2025/2026
Weizmann Institute of Science

Due Thursday, Dec. 4, at 11:59pm

This assignment has two parts. The first part aims to introduce you to Python as a tool for manipulating images. For this, you will build your own version of a very basic image processing pipeline. 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`, `PIL`, and `matplotlib`. The purpose of the second part is to introduce you to very basic principles of image formation and exposure control. For this, you will build your own simple *camera obscura*, which is basically a fancy term for a pinhole camera. Towards the end of this document, you will find a "Deliverables" section describing what you need to submit. Throughout the writeup, we also mark in red questions you should answer in your submitted report. Lastly, there is a "Hints and Information" section at the end of this document that is likely to help. We strongly recommend that you read that section in full before you start to work on the assignment.

Figure 1: One possible developed version of the RAW image provided with the assignment. Photo credit: Ohad Herches.

1 Developing RAW images (60 points)

For this problem, you will use the file `campus.NEF` included in the `./data` directory of the assignment ZIP archive. This is a RAW image that was captured with a Nikon camera. As we discussed in class, RAW images do not look like standard images before first undergoing a "development" process[1]. 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]The term "develop" comes from the analogy with the process of developing film into a photograph. As discussed in class, we often refer to this process as "rendering" the RAW images

1.1 Implement a basic image processing pipeline (50 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. 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 -w -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>`, `<white>`, `<r_scale>`, `<g_scale>`, and `<b_scale>`, and include them in your report.

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 sensor 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. As we discussed in class, it is possible that it has an offset due to dark noise, and saturated pixels due to over-exposure. 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 (10 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.

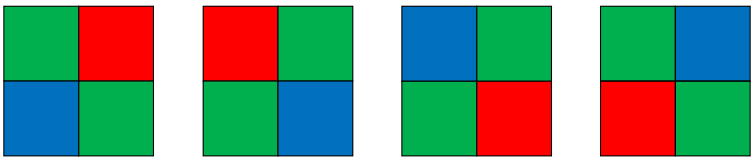

Figure 2: From left to right: '`grbg`', '`rggb`', '`bggr`', '`gbrg`'.

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 (5 points). After identifying the correct Bayer pattern, we want to perform white balancing. Implement both the white world and gray world 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 from the reconnaissance run of `dcraw`. These values are the white balancing presets the camera uses. *After completing the entire developing process*, check what the image looks like when using each of the three white balancing algorithms, decide which one you like best, and report your choice. (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.

Brightness adjustment and gamma encoding (5 points). You now have a full-resolution, linear RGB image, congrats!. This image, however, is not ready for display. To finish the job, we still need to apply gamma-encoding. But before gamma-encoding, let's brighten the image by stretching its histogram slightly.

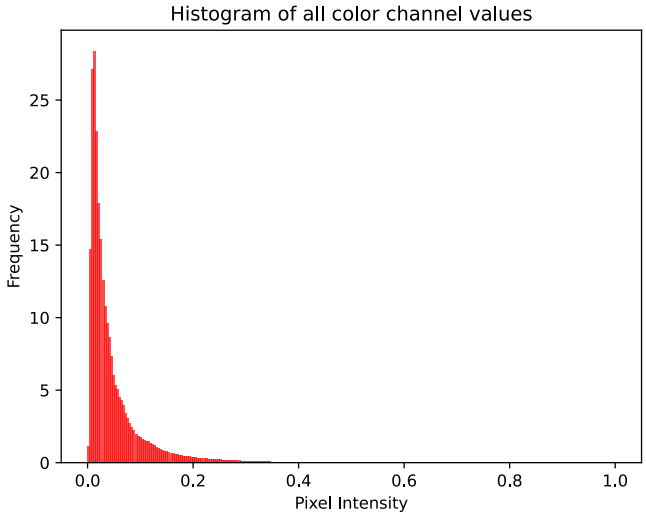

Figure 3: Histogram of pixel intensity values.

At this points, your image's histogram might look similar to the one in Figure. 3. The histogram shows that the majority of pixel values fall within the value range of $[0, 0.4]$ instead of 'using' the full range of $[0, 1]$. This would make the image appear dark, or underexposed, even after gamma-encoding. To fix that, we need to stretch the image's histogram so that the 'mass' of values fills the range $[0, 1]$ nicely. Brighten the image by linearly scaling it, again. This time, select the scaling boundaries directly from the pixel values. Use the `np.percentile` function to help you automatically find where 'most pixel values lay'. Remember to scale all channels using the same scalar, otherwise you risk messing up your white balancing step. You should experiment with several percentages and report and use the one that looks best to you. What could we have done with the camera controls, when capturing this image, to achieve *exactly* the same scaling effect?

After the histogram stretching, apply the non-linear gamma-encoding step. Use the 'default value' we mentioned in class [2]. Plot the image before and after this step, back to back, and add it to your report.

Compression (5 points). Finally, it is time to store the image, either with or without compression. Use `skimage.io` 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. The `skimage.io` function requires the image to be an 8-bit unsigned integer (`uint8`), so be sure to properly convert the

[2] You forgot it, didn't you? It's OKAY, you can look for it in the lecture pdf :).

image before saving. 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?

1.2 Boost image color saturation (10 points)

As we discussed in class, your camera ISP is doing all sorts of extra processing steps to make the final image more visually appealing. One easy trick to make an image 'pop' is to boost its color saturation (i.e., making reds more red, greens more green, etc.) This enhancement can be achieved by converting the image from the RGB color space into the HSV color space [2]. Don't worry, we will discuss color spaces in detail in the color lecture. But for now, the HSV color space encodes color information in a different way: the three channels of HSV are (H)ue, (S)aturation and (V)alue. Roughly speaking, the hue channel encodes only the color, the saturation encodes how saturated the color is, and the value encode how dark or bright the pixel is. For example, a saturation value of 255 means that the color is maximally saturated while a saturation of 0 means no color at all (i.e., a gray pixel).

Convert your image into the HSV color space and boost the saturation for all pixels by some appropriate constant value. Then convert the image back to RGB and display. Experiment with several boost levels, and find one that you deem to improve the image best.

2 Camera Obscura (40 points)

A camera obscura is essentially a dark box with a pinhole on one face (hence also "pinhole camera"), and a screen on the opposite face. Light reflecting off an object travels through the pinhole to the screen, and forms an inverted image of the object on the screen. All it takes to make such a camera is having a paper box and something you can use to pinch small holes on it. The caveat is that it will be hard to see the image formed with your naked eye. Instead, you will need to attach to the pinhole camera a digital camera with a long exposure time. Figure 4 shows a schematic of a pinhole camera constructed this way.

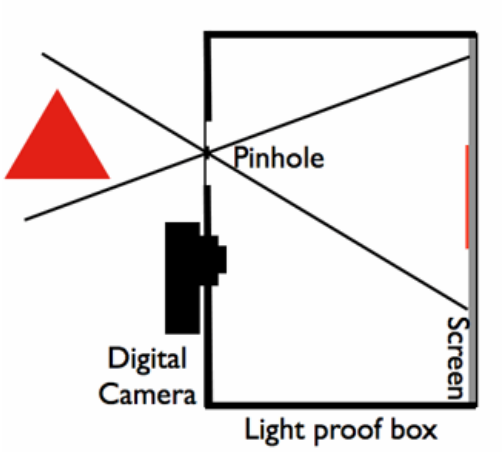

Figure 4: Schematic of a camera obscura.

2.1 Build the pinhole camera (30 points)

To design a basic easy-to-use pinhole camera, we recommend following these steps:

- Get a cardboard box. It does not need to be too big, but it does need to ultimately be "lightproof". A shoebox will do. The cardboard box should be such that the distance between the pinhole and the screen (i.e., the focal length) is longer than the minimum focus of the lens of your digital camera. Otherwise, all the images you will capture will be blurry.
- Download an app to your phone or obtain a digital camera that allows for long exposure times, around 15-30 seconds.
- Determine which inner face of the box should be your screen (i.e., the film or sensor). Cover the screen with white paper (printer paper generally works fine). (Optional) Cover the rest of the faces on the inside with black paper. If you prefer, you can use white and black paint instead of sheets of paper.
- On the face opposite the screen, cut a large hole. This hole should be bigger than the pinhole, e.g., about 3 cm in diameter.
- Take another piece of black paper, and punch the pinhole into that. Then, tape this piece of paper over the bigger hole on the box, so that light only enters through the pinhole. The advantage of this is that you can use multiple pieces of paper to test pinholes of varying diameters. Make your design so that you can easily switch between papers with different pinhole diameters.
- Next to the hole for the pinhole-carrying paper, cut a hole for the digital camera. The size of the hole should be such that you can attach the digital camera's aperture to it in a light-tight way—no light should enter through this hole. Additionally, the digital camera's hole should not be too far from the pinhole, as otherwise the digital camera's field of view may not be wide enough to capture the screen's image. You may need to angle the digital camera a bit towards the pinhole. At the same time, you should make sure that the digital camera is not blocking the light path inside or outside the box.
- Make sure that your digital camera has a memory card and a charged battery. Then attach it to the box. While attaching it, make sure that it is still possible to change its settings (e.g., focus, exposure, ISO) without destroying all of your construction. Turn off autofocus and adjust the focus of the camera manually so that it is focused on the screen.
- Duct tape your box all over! You really want to make it lightproof.

Once you have completed everything, submit a few photos of your constructed pinhole camera. Make sure to also report on all your design decisions, such as screen size, focal length, and field of view.

2.2 Use your pinhole camera (10 points)

It is now time to capture some images with your pinhole camera. We leave it up to you to identify interesting scenes. Once you have found what you want to photograph, point the pinhole towards it. Figure out the appropriate settings for the digital camera, then set it to capture an image for 16-30 seconds. Capture several photos (of different scenes) and add them to the report. Some suggested pinhole diameters are 0.1 mm (really just a pinprick), 1 mm, and 5 mm. These diameters are suggestions: in reality, your pinhole diameter should be about $1.9\sqrt{f\lambda}$, where f is the focal length, and λ is the wavelength of light (550 nm on average, for visible light). If you use this formula, then also go a few millimeters up and down, in order to have three pinhole diameters in total.

2.3 Bonus: Camera obscura in your room (10 points)

Instead of using a cardboard box, you can build a camera obscura using a room with a window. Use thick black paper to cover the window, and pinch a small hole at its center. Then, on the wall opposite the window, you will see an inverted projection of the view outside your room. Use a digital camera to capture an image showcasing this projection. See Abelardo Morell's Camera Obscura gallery for inspiration and some further instructions. Antonio Torralba and Bill Freeman also wrote a fun computer vision paper about room-sized pinhole cameras [1], which is definitely worth a read.

3 Deliverables

Your submission should be a single `zip` file which includes the following:

- A PDF report explaining what you did for each problem, including answers to all questions asked in Problems 1 and 2 (marked in red throughout the writeup), as well as any of the bonus problems you choose to do. 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).
- All your Python code, including code for the bonus problems you choose to complete, as well as a `README` file explaining how to use the code.
- For Problem 1.1: 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 1.2: `.PNG` files, showing the before and after the saturation boosting step.
- For Problem 2.1: At least two photographs of your constructed pinhole camera.
- For Problem 2.2: At least two photographs captured with your pinhole camera (ideally different scenes).
- For Bonus Problem 2.3: A photograph of your room's "screen" with the projection of the view from outside, as well as a photograph of your covered window "pinhole".

4 Hints and Information

dcraw version. Make sure to download and install the latest version (version 9.28) of `dcraw`. For Windows, you can download the `.exe` file in this link. To install `dcraw` on a Linux machine, install dcraw with:

```
sudo apt-get install dcraw -y
```

For macOS, use:

```
brew install dcraw
```

Package management. We recommend using Anaconda for Python package management. Once you install `conda`, you can create an environment specifically for the class as follows:

```
conda create --name <ENV-NAME> numpy scipy scikit-image
```

If you feel more comfortable using some other Python package management system, such as `pip`, you are welcome to do so. Either way, 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.

Image visualization. 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)    # draw first image
plt.imshow(im1)
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:

```
# 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.

Indexing. 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 5. 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	

Figure 5: Indexing sub-images.

Displaying results. 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 recommend scaling and clipping intermediate image `im_intermediate` before displaying with `matplotlib`.

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

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.

Working with the `PIL` package. To apply operations with `PIL`, you first need to convert your numpy array into a `PIL` object. Then, once you are finished doing your processing, you can convert back as shown below.

```
from PIL import Image
# convert numpy array to PIL image class
image_pil = Image.fromarray(image, 'RGB')

# convert PIL image class back to numpy array
image_np = np.array(image_pil)
```

Note that you might need to do these conversions several times back and forth to achieve your desired processing.

Exposure for camera obscura. In many point-and-shoot cameras, as well as in most manual-control cameras (DSLR, rangefinder, mirrorless, and so on), exposures up to 30 seconds are readily available. If this exceeds your camera's settings, you can reduce exposure times by using a large aperture size for your lens, as well as a large ISO setting (possibly up to 1600).

Modern smartphone cameras typically have a hardware limit on maximum exposure time at around 4 seconds. However, many phones nowadays support a "long exposure" mode, where longer exposure times are simulated by capturing a sequence of images and summing them all up at the RAW stage. This article provides some information for both iOS and Android phones.

If you decide to use your smartphone, you should also use an app that enables manual control of camera settings (especially focus, exposure time, and ISO). Additionally, make sure to disable the flash.

Data capture tips. Below are some tips that will hopefully make capturing images with your pinhole camera easier:

- We strongly recommend that you image a scene **outside during a sunny day**, while ideally being yourself in the shade. Capturing images indoors will require extremely long exposures, which is impractical.
- Make sure to place your pinhole camera on a surface that is absolutely steady throughout the exposure time. Definitely do not hole it with your hand.
- When focusing your digital camera, do so with the aperture of your lens fully open (even if you later decide to stop down the aperture while capturing images.
- If your photographs are blurry, it could be because either your camera is not focused correctly on the screen, or the pinhole is too large. To make sure that it is not a focusing issue, place a piece of paper with text on the screen and take a photo (or look at the viewfinder of your digital camera) with your box open. Make sure the camera is focused so that the text appears sharp. If the issue is the size of your pinhole, try another one that is smaller.

5 Credits

This assignment was adapted from an assignment originally created by Ioannis Gkioulekas at Carnegie Mellon University. According to Ioannis: "A lot of inspiration for the first part of this assignment came from Robert Sumner's popular guide for reading and processing RAW files. Some components of the first part were adapted from James Tompkin's computational photography course at Brown. Some inspiration for the second part came from James Hays' and Gordon Wetzstein's computational photography courses, at Brown and Stanford respectively."

References

[1] A. Torralba and W. T. Freeman. Accidental pinhole and pinspeck cameras: Revealing the scene outside the picture. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 374–381. IEEE, 2012.

[2] Wikipedia contributors. Hsl and hsv. https://en.wikipedia.org/wiki/HSL_and_HSV, 2024.