Problem set 1

TTIC 31040 – Spring 2021

March 30 2021; due April 6 2021, 11:59pm

Instructions: Turn in a write-up of your solution (as markdown cells) along with the code and the results as an iPython (Jupyter) notebook, including any figures or tables you generate. Please name your notebook `firstname-lastname-ps1.ipynb` and upload it on Canvas. Make sure the notebook you are submitting has been fully run, so that all the results are displayed in it.

Collaboration policy: it is OK (and encouraged) to discuss the problem set and any ideas for solving it with other students. However, in the end you must write your solution on your own. This includes writing any code and running any experiments.

1 Color Alignment

The Russian photographer Sergei Mikhailovich Prokudin-Gorskii (1863-1944) dreamed of producing color photographs, though the technology for color photography did not yet exist. He conducted photographic surveys of the Russian Empire using red, blue and green color filters to take photographs, with the camera moving slightly between shots. This leads to three images (a "triptych") of the same scene with small offsets between them (Figure 1)– there was no way to combine these images automatically in Prokudin-Gorskii's day (instead, he painstakingly aligned them by hand), but we will develop a simple technique for aligning the images and producing the color photos he had tried to make.

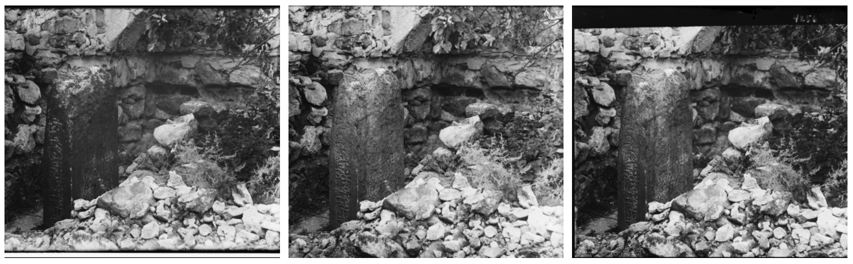

Figure 1: Red, green and blue parts of a tryptich. Note that the camera visibly moved between each shot

Specifically, we will implement an automatic procedure for taking the red, green and blue filter images and combining them into an RGB color image.

1.1 Synthetic (”manufactured”) offsets

The red, green, and blue components of an image are called “channels”. Due to camera motion between shots, the color channels are misaligned. We will start with a simple synthetic example to simulate this effect. We will take a regular RGB image, shift its color channels around so they become misaligned (using a random shift for both the horizontal and the vertical directions for R and B relative to G which stays in place). Then, we need to shift the channels so that they align again, finding the optimal displacement. We need to pick one of the channels as the “canonical” image, then align the other two channels to it according to a metric (measure of alignment quality). More precisely, assuming we use the green channel G as the ”anchor” to which we align the red and blue channels R and B, let D be the measure of discrepancy between two single-channel images; we are seeking

$$(t_R^*, t_B^*) \; = \; \underset{t_R, t_B \in \mathcal{T}}{\arg\min} \left[D(t_R(R), G) + D(t_B(B), G) \right], \tag{1}$$

where is the set of transformations under consideration. In our case is the set of shifts of up to k pixels (horizontally and/or vertically).

We can solve (1) by an exhaustive search over a window of possible displacements to align the red channel and the blue channel to the green channel. For our synthetic example we know that the displacements (in your experiments, limit the random shifts to ± 15 pixels). We will implement two metrics (SSD and NCC), but you are welcome to try other metrics as well.

1.1.1 Naive stacking

Load the provided image `test-image.png.` This is the original image without misaligned color. Extract the color channels, and shift them. Display the original image and the ”naively stacked” image where you simply superimpose the shifted color channels. Take care to pad the color channels when you shift. `numpy.roll` may be useful here.

1.1.2 Alignment with SSD

We need a metric to compare two images as we shift color channels to align. First, we will implement the SSD (sum of squared differences) metric, which is a common distance between vectors. We use this metric to compare image pixel intensity differences. Implement a color channel alignment procedure with this metric.

For vectors $\mathbf{u}, \mathbf{v}$ define:

$$SSD(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^{n} (\mathbf{u}_i - \mathbf{v}_i)^2 \tag{2}$$

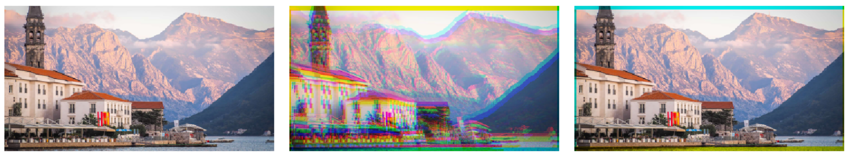

Figure 2: Original image, synthesized misaligned image, reconstructed image. You should see something similar, although not exactly the same due to random shifts.

1.1.3 Alignment with NCC

This metric compares normalized unit vectors using a dot product, for $\mathbf{u}, \mathbf{v}$ define:

$$NCC(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u}}{\|\mathbf{u}\|} \cdot \frac{\mathbf{v}}{\|\mathbf{v}\|} \tag{3}$$

Implement this metric and compare the results qualitatively with the SSD metric. *Note: NCC* is a measure of *similarity*, not *discrepancy* – it is high if the two images $\mathbf{u}$ and $\mathbf{v}$ are similar, and low if they are not. Adjust your math accordingly.

Display the resulting re-aligned images, and discuss ways for quantitatively evaluating the results on the synthetic example (where you have the benefit of knowing the ”ground truth” in the original, clean image).

1.2 Real data

Next we will use the code you developed in the previous section on two real “triptychs” of Prokudin (one of which is seen in Figure 1), using the code you wrote for the synthetic example. Pick a metric you think is preferred. For each triptich, display the results with the naive stacking, and the result with your optimized alignment. A code snippet in the provided starter notebook shows you how to load the three channels from separate files and display the stacked result as a color image.

1.2.1 Bonus

Pick any additional image triptych(s) from the **Prokudin-Gorskii Collection** and align the color channels. You wll need to crop the individual color channels by hand, using a tool (we recommend IrfanView).

Note: you may have to downsample the images to get good results at a reasonable speed with the simple technique we have described in this problem, as the raw images are rather large. Play around with image scale and offsets (for larger images the $[-15, 15]$ window will not work) as well as cropping the dark regions around the image for best visual effect.

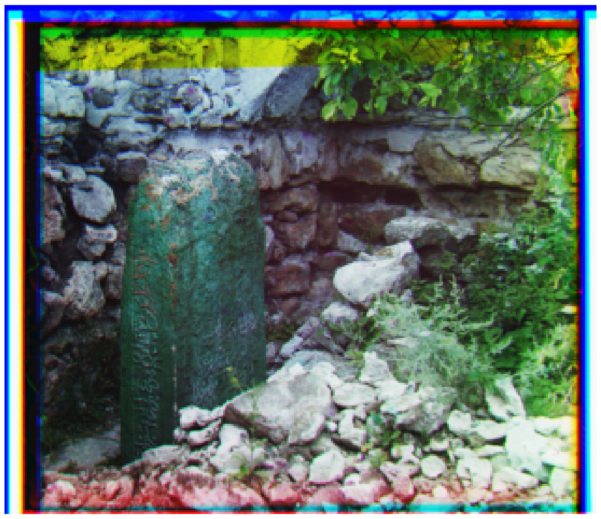

Figure 3: Aligned Prokudin-Gorskii image obtained from the 3 channel images in Figure 1.

2 Demosaicing

Problem 1 introduced the idea of color channel alignment for old photographs, but there is also a modern "color alignment" problem that must be solved whenever a digital picture is taken – demosaicing.

The raw image that is captured by a real digital camera is actually a *single-channel* picture taken through a color filter array (CFA) that disperse the red, green and blue pixels according to some pattern. Figure 4 has a standard "RGGB" Bayer pattern we will use in this problem; you can read up for more information Note here that the green pixels are sampled more frequently than red or blue.

Our goal will be take a single-channel "Bayer image" (see Figure 6 left) and "demosaic" it into a proper RGB image.

2.1 Linear interpolation

We will start with a linear interpolation approach to demosaicking. The simplest possible solution to this task would be top loop through the mosaic, placing the correct value into each channel depending on the row and column based on the pattern in Figure 6. However, this approach would be prohibitively slow on large images, and so we will take a faster approach with convolution.

In class we will be discussing *image filters*, a pattern convolved with an image to produce a desired effect (for example, the **box filter** is a commonly-used low-pass filter which blurs an image). Figure 5 gives you a hint on how to

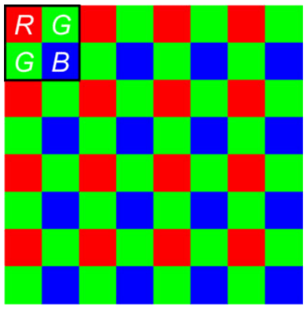

Figure 4: RGGB Bayer pattern

design filters for each color channel.

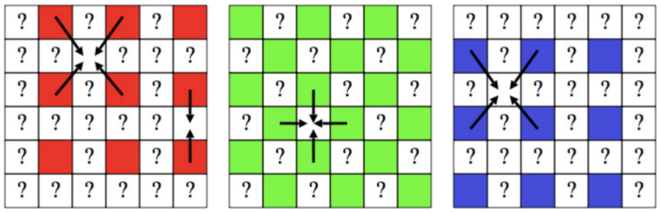

Figure 5: Demosaicking through linear interpolation.

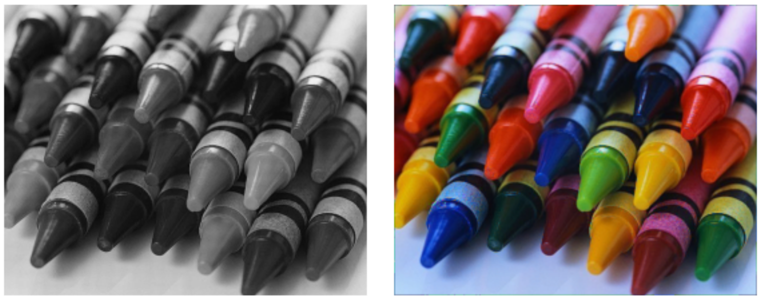

Figure 6: The raw Bayer pattern mosaic (left) which appears as a graylevel image. The RGB image once the left image has been properly separated into color channels (right).

Using the provided single-channel mosaic image `crayons.bmp`, apply naive per-channel interpolation and display the resulting color image next to the mosaic. You should see results similar to Fig 6. If you zoom in, you will notice some artifacts (little color splotches/speckles that clearly are a result of noisy interpolation).

2.2 Freeman method

In 1985 **Bill Freeman** proposed an improvement to the above approach by noting that there are twice as many **G** samples as **R** and **B**. This approach starts by doing a linear interpolation as in 2.1, then keeping the **G** channel fixed but modifying the **R** and **B** channels.

First, this approach computes the difference images $\mathbf{R} - \mathbf{G}$ and $\mathbf{B} - \mathbf{G}$ between the respective (naively) interpolated channels. Mosaicing artifacts tend to show up as small "splotches" in these images. To eliminate the "splotches", apply median filtering (e.g. `cv2.medianBlur,scipy.signal.medfilt2d,`or your own implementation) to the $\mathbf{R} - \mathbf{G}$ and $\mathbf{B} - \mathbf{G}$ images. Finally, create the modified **R** and **B** channels by adding the G channel to the respective difference images. Play around with the size of the median filter and compare the output of this method to the original image – is the Freeman method successful on this image? What kinds of artifacts does it erase or introduce?

Display the color image with Freeman's demosaicing, and explain how you chose the filter size, and what the effects of making the filter size larger seems to be.

Hint: to make the corrective interpolation work as intended, it may be useful to bring the color channels to the same mean value prior to subtractions (by removing the mean for each color channel); you need to remember to add it back before displaying the final color image.