MIT EECS 6.815/6.865: Assignment 4:

# Denoising and Demosaicking

Due Wed October 2 at 9pm

## 1 Summary

- Denoising based on averaging
- Variance and signal-to-noise computation.
- Image alignment using brute force least squares
- Basic green channel demosaicking
- Basic Red and Blue channel demosaicking
- Edge-based green channel demoasaicking
- Red and blue channel demosaicking based on difference to green.
- 6.865 only: reconstructing the color of old Russian photographs

## 2 Denoising from sequence

### 2.1 Basic sequence denoising

Write a simple denoising method `denoiseSeq(imageList)` that takes an image sequence as input and returns a denoised version computed by averaging all the images. At this point, you should assume that the images are perfectly aligned.

Try it on the sequence in the directory `aligned-ISO3200`. We suggest testing with at least 16 images, and experimenting to see how well things converge.

### 2.2 Variance

Write a function `logSNR(imageList, scale=1.0/20.0)` that returns an image visualizing the per-pixel and per-channel log of the signal-to-noise ratio scaled by `scale`.

Notes: the scale is with respect to log10, not with respect to decibels. Also, use the original definition of variance (division by (n-1))

Compare the signal-to-noise ratio of the ISO 3200 and ISO 400 sequences. Which ISO has better SNR?

Same as above, use at least 16 images, but more will give you better estimates.

### 2.3 Alignment

Write a function `align(im1, im2, maxOffset=20)` that returns the `y, x` offset that best aligns im2 to match im1. Use a brute force approach that tries every possible integer translation and evaluates the quality of a match using the squared error norm (the sum of the squared pixel differences). Whereas you can use a loop to try the possible 2D offsets, we highly recommend that you use only numpy functions for the inner loop.

The `numpy.roll` function might come in handy. It wraps around, but you should ignore boundary pixels anyway: Ignore the difference for all the pixels less than `MaxOffset` away from the edge.

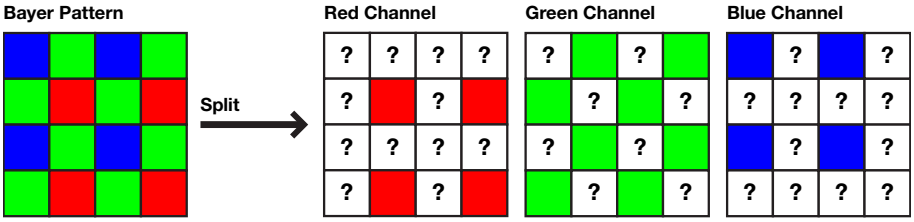
Make sure you test your procedure before moving forward.

Use your `align` function to create a function `alignAndDenoise(imageList, maxOffset=20)` that registers all images to the first one in a sequence and outputs a denoised image even when the input sequence is not perfectly aligned. Warning: your code is probably going to run slow. Test it on small simple inputs before trying a full sequence.

We suggest trying it on at least the 9 first images of the sequence `green`.

## 3 Demosaicking

Most digital sensors record color images through a Bayer mosaic, where each pixel captures only one of the three color channels, and software interpolation is then needed to reconstruct all three channels at each pixel. The green channel is recorded twice as densely as red and blue, as shown below.




We represent raw images as single-channel image, i.e. 2D numpy arrays. They can be read using the `imreadGrey` function in `imageIO`. The images are encoded linearly and imreadGrey does the right thing to keep them linear (unlike imread, it doesn't perform gamma decoding). You can open them in your favorite image viewer and zoom in to see the pattern of the Bayer mosaic.

We provide you with a number of raw images and your task is to write functions that demosaick them. We encourage you to debug your code using `signs-small.png` because it is not too big and exhibits the interesting challenges of demosaicking.

For simplicity, we ignore the case of pixels near the boundary of the image. That is, the first and last two rows and columns of pixels don't need to be reconstructed. This will allow you to focus on the general case and not worry about whether neighboring values are unavailable. It's actually not uncommon

for cameras and software to return a slightly-cropped image for similar reasons. See `http://www.luminous-landscape.com/contents/DNG-Recover-Edges.shtml`

## 3.1 Basic green channel

Write a function `basicGreen(raw, offset=0)` that takes as input a single-channel (2D) raw image and returns a 2D array corresponding to the interpolated green channel. Your function will be called to create the green channel of the final image as

```
out[:, :, 1]=basicGreen(raw, 0)
```

The offset encodes whether the top-left pixel or its right neighbor are the first green pixel. You should make your code general since different cameras make different choices.

For pixels where green is recorded, simply copy the value. For other pixels, the interpolated value is simply the average of its 4 recorded neighbors (up, down, left, right).

You can ignore the first and last row and column. This way, all the pixels you need to reconstruct have their 4 neighbors.

Try your image on the included raw files and verify that you get a nice smooth interpolation. At this point,

You can try on your own raw images by converting them using the program `dcraw`

## 3.2 Basic red and blue

Next, write a function `basicRorB(raw, offsety, offsetx)` to deal with the sparser red and blue channels. Similarly, it takes a 2D raw image and returns a single 2D channel as output. The function will be called twice

```
out[:, :, 0]=basicRorB(raw, 1, 1)
out[:, :, 2]=basicRorB(raw, 0, 0)
```

`offsety, offsetx` are the coordinates of the first pixel that is red or blue. In our case, the figure above shows that 0,0 is blue while 1,1 is the red.

Similar to the green-channel case, copy the values when they are available. For interpolated pixels that have two direct neighbors that are known (left-right or up-down), simple take the linear interpolation between the two values. For the remaining case, interpolate the four diagonal pixels.

You can ignore the first and last two rows or columns to make sure that you have all the neighbors you need.

Implement a function `basicDemosaic(raw, offsetGreen=0, offsetRedY=1, offsetRedX=1, offsetBlueY=0, offsetBlueX=0)` that takes a single-channel raw image and returns a full RGB images demosaicked with the above function. You might observe some checkerboard artifacts around strong edges. It's expected from such a naïve approach.

You might have to try some different combinations of offsets for some images before you see a reasonable result.

# 4 Edge-based green

One central idea to improve demosaicking is to exploit structures and patterns in natural images. In particular, 1D structures like edges can be exploited to gain more resolution. We will implement the simplest version of this principle to improve the interpolation of the green channel. We focus on green because it has a denser sampling rate and usually a better SNR.

For each pixel, we will decide to adaptively interpolate either in the vertical or horizontal direction. That is, the final value will be the average of only two pixels, either up and down or left and right. We will base our decision on the comparison between the variation up-down and left-right. It is up to you to think or experiment and decide if you should interpolate along the direction of biggest or smallest difference. It's also possible that the slides might help.

Write a function `edgeBasedGreen(raw, offset=0)` that interpolates the green channel adaptively. This function should give perfect results for horizontal and vertical edges.

Also write a function `edgeBasedGreenDemosaic(raw, offsetGreen=0, offsetRedY=1, offsetRedX=1, offsetBlueY=0, offsetBlueX=0)` that does the same thing as `basicDemosaic` but uses your edge based method on the green channel.

# 5 Red and blue based on green

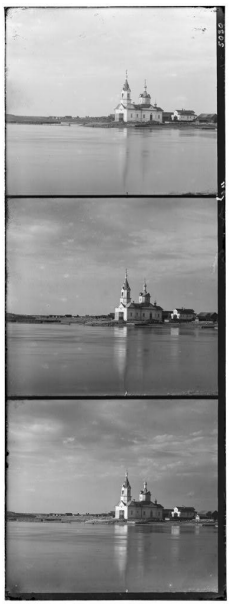
A number of demosaicking techniques work in two steps and focus on first getting a high-resolution interpolation of the green channel using a technique such as `edgeBasedGreen`, and then using this high-quality green channel to guide the interpolation of red and green.

One simple such approach is to interpolate the difference between red and green (resp. blue and green). Adapt your code above to interpolate the red or blue channel based not only on a raw input image, but also on a reconstructed green channel. Your function should be called `greenBasedRorB(raw, green, offsety, offsetx)` and proceeds pretty much as your basic version, except that it is the difference R-G or B-G that gets interpolated. In this case, we are not trying to be smart about 1D structures because we assume that this has been taken care of by the green channel.

The function `improvedDemosaic(raw, offsetGreen=0, offsetRedY=1, offsetRedX=1, offsetBlueY=0, offsetBlueX=0)` should call your new demosaicing functions on a raw image.

# 6 Sergei Prokudin-Gorsky

The Russian photographer Sergey Prokudin-Gorsky took beautiful color photographs in the early 1900s by sequentially exposing three plates with three different filters.
`http://en.wikipedia.org/wiki/Prokudin-Gorskii`
`http://www.loc.gov/exhibits/empire/gorskii.html`



We include a number of these triplets of images in the `data/Sergei/` directory, (courtesy of Alyosha Efros). Your task is to reconstruct RGB images given these inputs.

### 6.1 Cropping and splitting

Write a function split that crops the image and turns it into one 3-channel image. We have cropped the original images so that the image boundaries are approximately 1/3 and 2/3 along the y dimension.

### 6.2 Alignment

The image that you get out of your `split` function will have its 3 channels misaligned. Write the function `sergeiRGB(raw, alignTo=1)` that first calls your `split` function, but then aligns two channels of your rgb image to the remaining third channel (specified by alignTo). Your function should return a beautifully aligned color image.

## 7 Extra credit

Numerically compute the convergence rate of the error for the denoising at each pixel. Use a regression in the log domain (log error vs. log number of images).

Implement a coarse-to-fine alignment.

Take potential rotations into account for alignment. This could be slow!

Implement smarter demosaicking. Make sure you describe what you did. For example, you can use all three channels and a bigger neighborhood to decide

the interpolation direction.

# 8 Submission

Turn in your python files and make sure all your functions can be called from a module `a4.py`. Put everything into a zip file and upload to out submission website (and to Stellar for backup!). Do NOT upload any images with your submission. Just the python files.

Include a `README.txt` containing the answer to the following questions:

- How long did the assignment take?
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented
- Collaboration acknowledgement (but again, you must write your own code)
- What was most unclear/difficult?
- What was most exciting?
- What ISO has better SNR?
- Which direction you decided to interpolate along for the edgeBasedGreenChannel