MIT EECS 6.815/6.865: Assignment 3:

Denoising and Demosaicing

Due Wednesday October 7 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 a sequence of images

2.1 Basic sequence denoising

In our image formation model the image captured by the camera $I(x, y, z)$ is the sum of a latent image $\hat{I}(x, y, z)$ and some random noise $n(x, y, z)$:

$$I(x, y, z) = \hat{I}(x, y, z) + n(x, y, z) \tag{1}$$

Note that both n and I are random variables in this model. If we further assume that n is zero-mean, then the expected value $\mathbb{E}(I) = \hat{I}$ is the true image we wish to capture.

This gives us a simple denoising method: we can take N shots of the same subject (i.e. the subject and the camera are static) and average these measurements (i.e. the captured images). The empirical mean is an estimate of the the true image $\hat{I}$:

$$\mathbb{E}(I) = \frac{1}{N}\sum_{k=1}^{N} I_k \tag{2}$$

Where each I_k is a realization of the random variable I.

1 Write a simple denoising method `Image denoiseSeq(const vector<Image> &imgs)` in `align.cpp` 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 and have the same size.

Try your function on the sequence in the directory `aligned-ISO3200` using following the example in `a3_main.cpp`. We suggest testing with at least 16 images, and experimenting with more images to see how well the method converges.

2.2 Variance

Given the same set of N measurements (each measurement being an entire image), we compute the variance as:

$$\sigma^2(I) = \mathbb{E}((\mathbb{E}(I) - I)^2) \tag{3}$$

We can then use the variance to get an estimate of the noise level in the image. We'll compute the log signal-to-noise ratio as:

$$\text{SNR} = 10 \cdot \log_{10}\left(\frac{\mathbb{E}(I^2)}{\sigma^2(I)}\right) \tag{4}$$

2.a Write a function `Image logSNR(const vector<Image> &imSeq, float scale=1.0/20.0)` in `align.cpp` that returns an image visualizing the per-pixel and per-channel log signal-to-noise ratio (using the formula above) scaled by `scale`. Note: Also, use the original definition of variance (division by $N - 1$)

2.b Compare the signal-to-noise ratio of the `ISO 3200` and `ISO 400` sequences. Which ISO has better SNR? Answer the question in the submission system.

Same as above, use at least 16 images, but more will give you better estimates. Visualize the variance of the images in `aligned-ISO3200` in `a3_main.cpp`.

2.3 Alignment

The image sequences you have looked at so far have been perfectly aligned. Sometimes, the camera might move, so we need to align the images before denoising.

3.a Write a function `vector<int> align(const Image &im1, const Image &im2, int maxOffset=20)` in `align.cpp` that returns the `[x, y]` offset that best aligns im2 to match im1. Ignore the difference for all the pixels less than or equal to `MaxOffset` away from the edges.

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).

The `Image roll(const Image &im, int xRoll, int yRoll)` function in `align.cpp` might come in handy. It circularly shifts an image, causing borders to wrap around. However, since you will be ignoring boundary pixels, wrapping the pixel values should not be a problem. Make sure to test your procedure before moving on.

3.b Use `align` to create a function `Image alignAndDenoise(const vector<Image> &imSeq, int maxOffset=20)` in `align.cpp` that registers all images to the first image in the image sequence then outputs a denoised image. This allows you to produce a denoised image even when the input sequence is not perfectly registered to begin with.

(a) Averaging (b) Aligned Averaging

Figure 1: Result of denoising by naively averaging 9 images (a) and then by averaging after first aligning the images (b). Zoom in on the image edges and note that first aligning the images helps to perserve the crispness of the edges

3 Demosaicing

Most digital sensors record color images through a Bayer mosaic, where each pixels captures only one of the three color channels, and a 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.

> 4 Why does it make sense to oversample the green channel compared to red and blue? Answer in the submission form.

Figure 2: The Bayer mosaic

We represent raw images as grayscale images. You can open these images 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 to demosaic them. We encourage you to debug your code using `signs-small.png` because it is not too large and exhibits many of the interesting challenges of demosaicing.

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. **For the border pixels that you do not calculate, copy the pixels values from the same location in the original raw image to your output image.**

See `http://www.luminous-landscape.com/contents/DNG-Recover-Edges.shtml`

3.1 Basic green channel

> 5 Write a function `Image basicGreen(const Image &raw, int offset=0)` in `demosaic.cpp` that takes as input a raw image and returns a single-channel 2D image corresponding to the interpolated green channel.
>
> The offset encodes whether the top-left pixel or its neighbor immediately to the right is the first green pixel. In the case of Fig. 2, the second pixel in the first row is green so `offset=1`. For the image `signs-very-small.png` `offset=0`. Make your code general for either offset since different cameras use different conventions.
>
> For pixels where green is recorded, simply copy the value. For other pixels, the interpolated green value is simply the average of its 4

recorded green 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 a 4-neighborhood.

Try your image on the included raw files and verify that you get a nice smooth interpolation. You can try on your own raw images by converting them using the program `dcraw`

3.2 Basic red and blue

6.a Write a function `Image basicRorB(const Image &raw, int offsetX, int offsetY)` in `demosaic.cpp` to deal with the sparser red and blue channels. Similarly, it takes a raw image and returns a 2D single-channel image as output. `offsetX, offsetY` 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. The function will be called twice:

```
Image red = basicRorB(raw, 1, 1);
Image blue = basicRorB(raw, 0, 0);
```

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), simply 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.

6.b Implement a function `Image basicDemosaic(const Image &raw, int offsetGreen=0, int offsetRedX=1, int offsetRedY=1, int offsetBlueX=0, int offsetBlueY=0)` in `demosaic.cpp` that takes a raw image and returns a full RGB image demosaiced with the above functions. You might observe some checkerboard artifacts around strong edges. This is expected from such a naïve approach.

4 Edge-based green

One central idea to improve demosaicing 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.

if you should interpolate along the direction of biggest or smallest difference.

7.a Should we interpolate along the direction of biggest or smallest variation? Answer in the submission form.

It is up to you to think or experiment and decide what to do. It's also possible that the slides might help...

7.b Write a function `Image edgeBasedGreen(const Image &raw, int offset=0)` in `demosaic.cpp` that takes a raw image and outputs an adaptively interpolated single-channel image corresponding to the green channel. This function should give perfect results for horizontal and vertical edges.

7.c Write a function `Image edgeBasedGreenDemosaic(const Image &raw, int offsetGreen=0, int offsetRedX=1, int offsetRedY=1, int offsetBlueX=0, int offsetBlueY=0)` in `demosaic.cpp` that takes a raw image and returns a full RGB images with the green channel demosaiced with `edgeBasedGreen` and the red and blue channels demosaiced with `basicRorB`.

7.d Do you see any artifact with this new method? If yes, what could be improved? (Answer in the form)

5 Red and blue based on green

A number of demosaicing 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 blue.

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.

8.a Write a function called `Image greenBasedRorB(const Image &raw, Image &green, int offsetX, int offsetY)` in `demosaic.cpp` that 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 demosaicing pipeline is then as follows:

```
Image green = edgeBasedGreen(raw, 0);
Image red = greenBasedRorB(raw, green, 1, 1)
Image blue = greenBasedRorB(raw, green, 0, 0)
```

8.b Write a function `Image improvedDemosaic(const Image &raw, int offsetGreen=0, int offsetRedX=1, int offsetRedY=1, int offsetBlueX=0, int offsetBlueY=0)` in `demosaic.cpp` that takes a raw image and returns a full RGB images with the green channel demosaiced with `edgeBasedGreen` and the red and blue channels demosaiced with `greenBasedRorB`.

Try this new improved demosaicing pipeline on `signs-small.png` in `a3_main.cpp` and notice that most (but not all) artifacts are gone.

(a) `basicDemosaic` (b) `edgeBasedGreenDemosaic` (c) `improvedDemosaic`

Figure 3: Results of demosaicing using the 3 different methods. Notice how artifacts appear around the edges of the resulting image when using basic interpolation. However, an edge aware demosaicing algorithm significantly decreases the artifacts around these edges.

6 6.865 only (or 5% Extra Credit): Sergey 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 `Input/Sergey` (courtesy of Alyosha Efros). Your task is to reconstruct RGB images given these inputs.

Figure 4: Sample image sequence by Prokudin-Gorsky

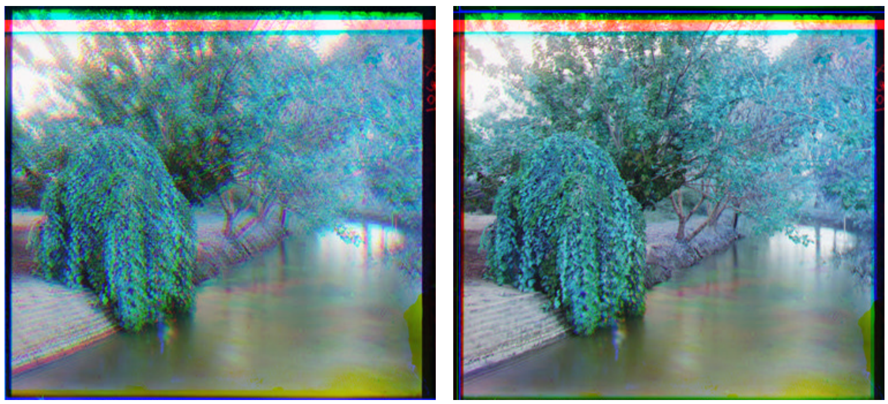
6.1 Cropping and splitting

9 Write a function `Image split(const Image &sergeyImg)` in `align.cpp` that vertically splits an image in 3 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. Use `floor` to compute the height of your final output image from the height of your input image.

6.2 Alignment

The image that you get out of your split function will have its 3 channels misaligned.

10 Write the function `Image sergeyRGB(const Image &sergeyImg, int maxOffset=20)` in `align.cpp` that first calls your `split` function, but then aligns the green and blue channels of your RGB image to the red channel. Your function should return a beautifully aligned color image.



(a) Naive RGB (b) Aligned RGB

Figure 5: Generating an RGB image from a single grayscale Sergey image

7 Extra credit (maximum of 10%)

- 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 demosaicing. 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 files to the online submission system (link is on Stellar) and make sure all your files are in the `asst` directory under the root of the zip file. If your code compiles on the submission system, it is organized correctly. The submission system will run code in your main function, but we will not use this code for grading. The submission system should also show you the image your code writes to the `./Output` directory

In the submission system, there will be a form in which you should answer the following questions:

- How long did the assignment take? (in minutes)
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented and their function signatures if applicable
- Collaboration acknowledgment (you must write your own code)
- What was most unclear/difficult?
- What was most exciting?