# Lab 8: Simulating a Digital Camera

## 1 Introduction

A lot of signal processing goes into making a digital camera effective and affordable. In this lab, you will simulate and implement some important aspects of a digital camera's operation.

Start by copying all the files from the lab directory on `bSpace` to your local workspace.

### 1.1 Lab Goals

- Become comfortable working with color images.
- Simulate capturing an image with a Bayer filter.
- Observe the effects of compression on image quality.

## 2 Image Capture

As you learned in lecture, a digital camera captures light with a two-dimensional array of charge coupled devices (CCDs) or complementary metal oxide semiconductor (CMOS) image sensors. Each sensor tracks the number of photons that hit it during an exposure, and then output a corresponding voltage. However, these technologies cannot capture information about the frequency of incident photons. Therefore, to add color to images, we need to make each sensor capture a specific color. This is done with colored filters that are placed over each image sensor (this is not a signal processing filter, but a physical filter, such as a piece of colored, translucent plastic that lets through only the desired color). Some cameras include three filters for each light sensor (one for red, green, and blue), but this is fairly expensive, so usually, each sensor is assigned one of the three colors. This is illustrated below.

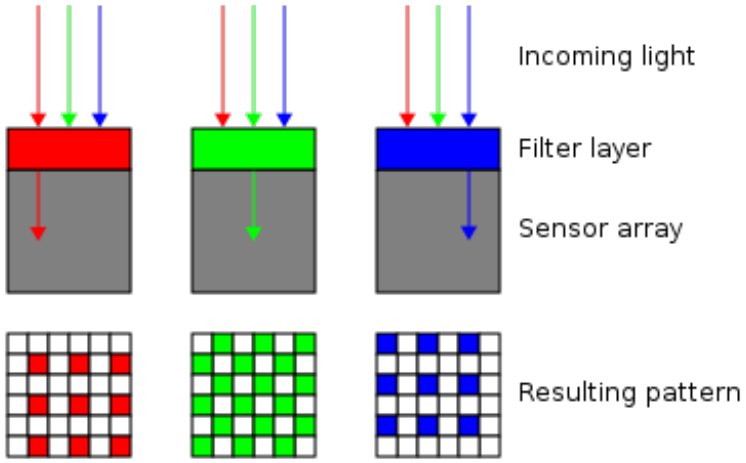


Figure 1: Filtering of incident light in a digital camera. Some image sensors sense red light, some sense blue light, and some sense green light. Image taken from http://en.wikipedia.org/wiki/Bayer_filter .

Notice that there is a very regular pattern for the various pixels. This pattern is called a Bayer filter. When you compose the blue, green, and red pixels, the Bayer filter pattern looks like this:

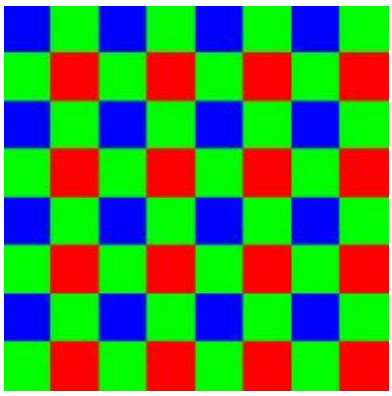

Figure 2: Bayer pattern for color filtering in digital camera capture. Image taken from http://www.laesieworks.com .

There are twice as many green pixels in this pattern as blue or red pixels. This is because green light affects our perception of brightness about twice as much as red or blue light. After capturing an image with a Bayer filter, the image color will look distorted. Here is an example on an image of some flowers.

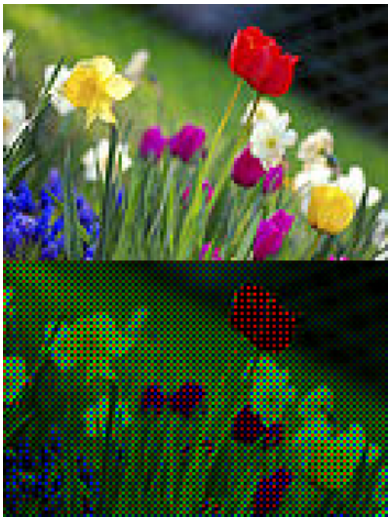

Figure 3: Original (top) and Bayer-filtered image (bottom). Image taken from http://en.wikipedia.org/wiki/Bayer_filter .

Clearly, there are a lot of artifacts and the colors are off. In order to make the image acceptable, it needs to undergo a process called *demosaicing*, which you will implement. Open the file `main.m`. This is going to be your main script. So to test your functions, you will run `main.m`. Start out by downloading the image `captured.mat` from `bSpace` and save it to your working directory. When you run `main.m`, you should see a Bayer-filtered image in Figure 1. Your task will be to demosaic it.

# 3 Demosaicing

Clearly, the image you captured in the previous section is not ready to be displayed. To make the image look reasonable, we need to *demosaic* it. This is a filtering process in which we essentially guess at the missing pixel colors. For instance, for each red pixel in the Bayer pattern, we need to guess the blue and green pixel values. There are many ways to do this. We're going to walk through two methods in this lab, known as pixel doubling interpolation and bilinear interpolation.

## 3.1 Pixel doubling interpolation

In this method, we will consider each 2×2 square of pixels individually. Now, to start off, we are given the following data in each square:

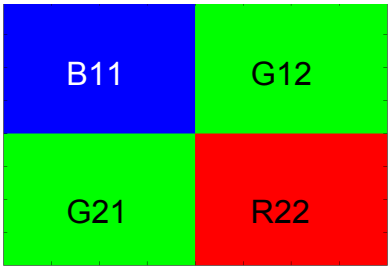


Figure 4: Pixel color pattern in each 2×2 square in the image.

This method fills in the missing color values by using neighboring measurements. So in the square above, we would fill in the missing color pixels as follows:


Figure 5: Interpolated color pixels for each 2×2 square in the image.

Note that in this pattern, we always use the green pixel from the same row.

1. Open the function `demosaicPDI.m`. Implement the interpolation technique described above. Don't worry about weird effects at the boundaries. (Hint: Is this filter LTI or not?)

2. Once your function is complete, run `main.m`. Does your demosaiced image look better than the captured image? Does it look perfect?

### 3.2 Bilinear Interpolation

The pixel doubling technique works fairly well, and it has the advantage of being very simple. However, you should be able to see artifacts that are not present in the original image. These artifacts can be reduced by using more complex interpolation tools. One such tool is called bilinear interpolation. The concept is quite simple: If a pixel is the desired color, just keep the sampled value. If the pixel does not have information for the desired color, consider the surrounding eight pixels. Some of those pixels will be of the desired color. Average the surrounding pixels of the desired color, and use that average. For instance, if we are interpolating the green value of a pixel in the interior of the image (i.e. not on the border), this would work as follows:

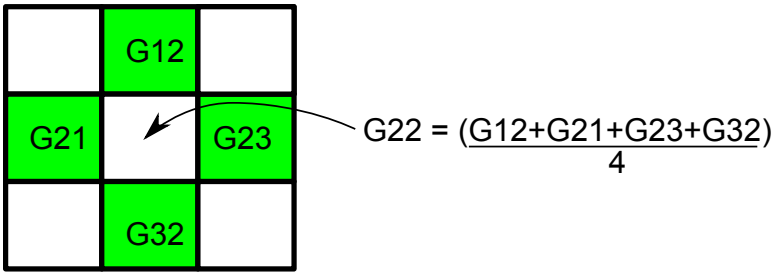


Figure 6: Bilinear interpolation for green pixels.

This takes a particularly simple form because there are so many green pixels in the image. If we are instead interpolating a red or blue pixel, it would work as follows:

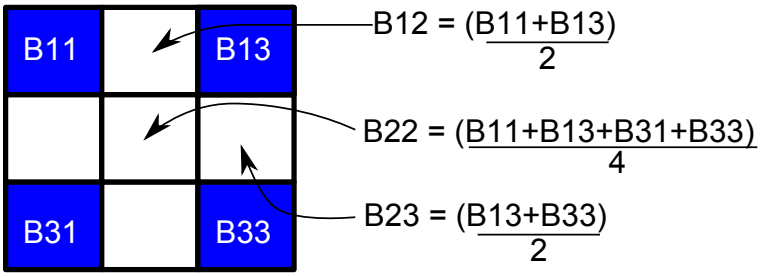


Figure 7: Bilinear interpolation for blue pixels. The interpolation for red pixels is identical.

Note that in this scenario, some pixels interpolate from 4 blue data points, while others interpolate from only 2, because there are not enough blue pixels in the vicinity. Note that this procedure is identical for red pixels.

1. To start out, open `demosaicBilinear.m`. The first thing you need to do is convert the image to the `double` data type. Images are represented as 3-D matrices of type `uint8`, and basic operations like averaging do not work as we would expect with such data types. You may find the function `im2double` useful. This will scale the image integers (which are in the set {`0,1,...,255`}) down to the interval `[0,1]`.

2. Apply the bilinear interpolation to the whole image. Do not implement this with for loops! Think about whether this filter is LTI.

3. Note that in your function `demosaicBilinear`, you will need to scale your values from `[0,1]` back up to `[0,255]` and you will also need to convert your filtered image back to the type `uint8`.

4. Once you have interpolated your image, run `main.m` and show both of the demosaiced images. How do the two techniques compare? Why?

# 4 Compression

The last task we will investigate is image compression. You learned in class about the JPEG standard. For this section, we just want you to play with a MATLAB GUI made by Anna Di Benedetto.

1. In your MATLAB console, type `compressione`. You should see a GUI pop up. In the text box, type either `images/monster.jpg`, or the path of another image in your working directory. Set the compression level to 8, and click the `Visualize in more detail` button. The middle panel shows the discrete cosine transform (DCT) coefficient magnitudes for each 8×8 block of pixels. Recall that the DCT is a transform similar to the DFT, used widely in image processing. The coefficients in the top left are low frequency, while bottom right coefficients are higher frequency, as shown here:

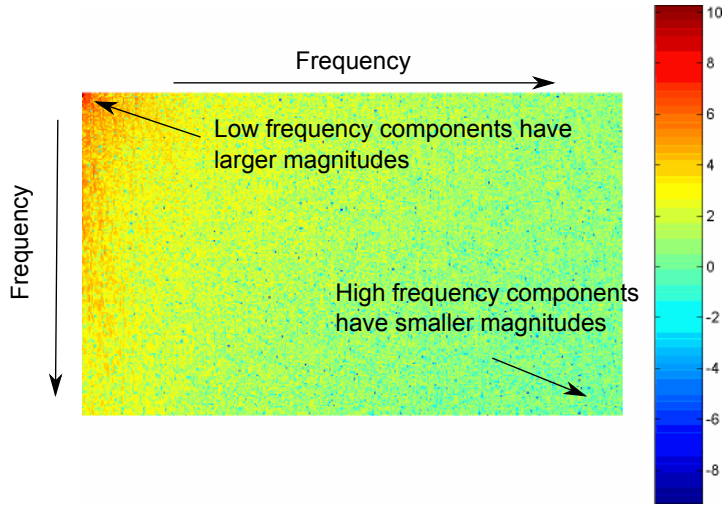


Figure 8: Sample DCT of an image.

   Zoom in on the DCT coefficient image so that you can make out individual pixels. The strips of black throughout most of the DCT coefficients suggest that some of them have been zeroed out. This is not the case! What you are seeing is the fact that high-frequency coefficients in real images are often very low or zero. Therefore, if we get rid of these small coefficients, it probably won't affect image quality too much.

2. To test this, set the compression level to 5, and click `Visualize in more detail` again. This time, the system is only keeping the top-leftmost 5×5 square from each larger 8×8 square of DCT coefficients. Does this degrade the image quality?

3. Now try compression level 1. This is keeping only the zero-frequency or DC component.