One objective in this lab to learn showing how interpolation is used to create color images for a digital camera. For the interpolation, you will learn how to implement FIR filters in MATLAB, and then use these FIR filters to perform the interpolation of images.

Digital Camera Color Imaging

The digital camera in your smartphone must perform a significant amount of signal processing in order to provide the user with a viewable image. [1] With a little knowledge of DSP, we can investigate some of that processing.

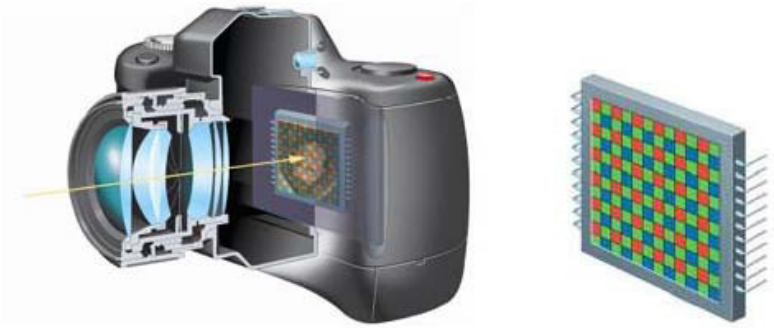

Figure 1: CFA in the digital camera (left), CFA module (right).

A color image requires at least three color values at each pixel location, usually red, green and blue for a computer image. Ideally, a camera would use three separate sensors in order to measure the red, green and blue intensities at every single pixel location. To reduce size and cost, many cameras use a single sensor

[1] Ref: B. K. Gunturk, J. Glotzbach, Y. Altunbasak, R. W. Schafer, R. M. Mersereau, "Demosaicking: Color Filter Array Interpolation in Single-Chip Digital Cameras," Center of Signal and Image Processing, Georgia Institute of Technology, Available at `http://users.ece.gatech.edu/ rmm/fall2003/ece6258/Demosaicking_SPM.pdf`

array in conjunction with a color filter array. The color filter array (CFA) will pass the red, the green or the blue component of light to a given pixel location on the sensor array. This means that the initially captured image matrix does not contain full color information at any pixel location; rather, any one pixel contains only red, or green, or blue intensity information for that pixel. Therefore, the camera must estimate the two missing color values at each pixel, and this estimation process is known as *demosaicking*.

Bayer CFA

Although several possible patterns exist for the CFA, the Bayer pattern is the one that is most commonly used.[2] As shown in Fig. 2, the Bayer CFA measures the green components of the image on a quincunx

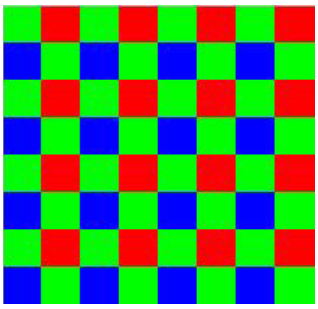

Figure 2: Bayer CFA pattern for an 8×8 image.

(checkerboard pattern) grid, while the red and blue components are measured on rectangular grids. The density of green pixels is twice that of either the red or blue pixels.

Image Sensors

Two technologies exist to manufacture imaging sensors: CCD (Charge Coupled Device) and CMOS (Complementary Metal Oxide Semiconductor). Most digital cameras use CMOS sensors. The CMOS sensor is a collection of tiny light-sensitive diodes that convert photons (light) into electrical current. The brighter the light that is incident on the photodiode, the greater the electrical current. This current is read using an ADC (analog-to-digital converter) that turns each pixel's light value into a digital value that is recorded. In the block diagram of the imaging system shown in Fig. 3, the 2D array of these digitized intensities is the output of the imaging array.

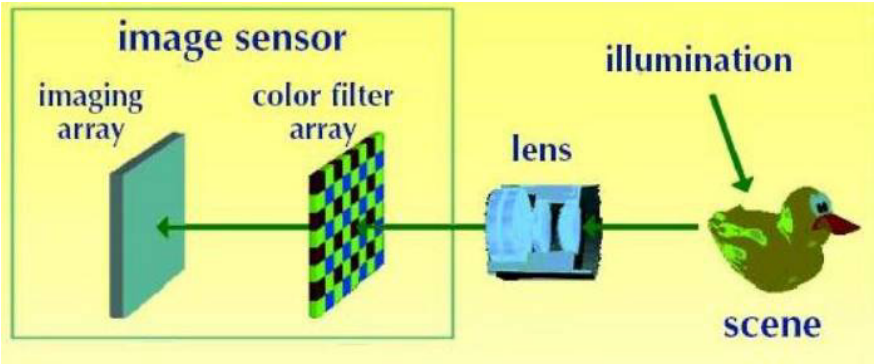

Figure 3: Digital still camera: image capture principle.

[2]Note: The Bayer pattern is the exact CFA that should be referred to for the remainder of this lab.

To summarize:

- A color image requires at least three color samples at each location. This would require three separate sensors at each location.
- For economic and size reasons, cameras are built with a single sensor in each location and a color filter array (CFA).
- At each pixel location, only one of the three color components is recorded.
- The 2D array of digitized intensities is the starting point for the demosaicking process.
- The DSP algorithm in the camera must estimate the two missing color values at each pixel; this is what will be implemented for the lab project.

Demosaiking Process

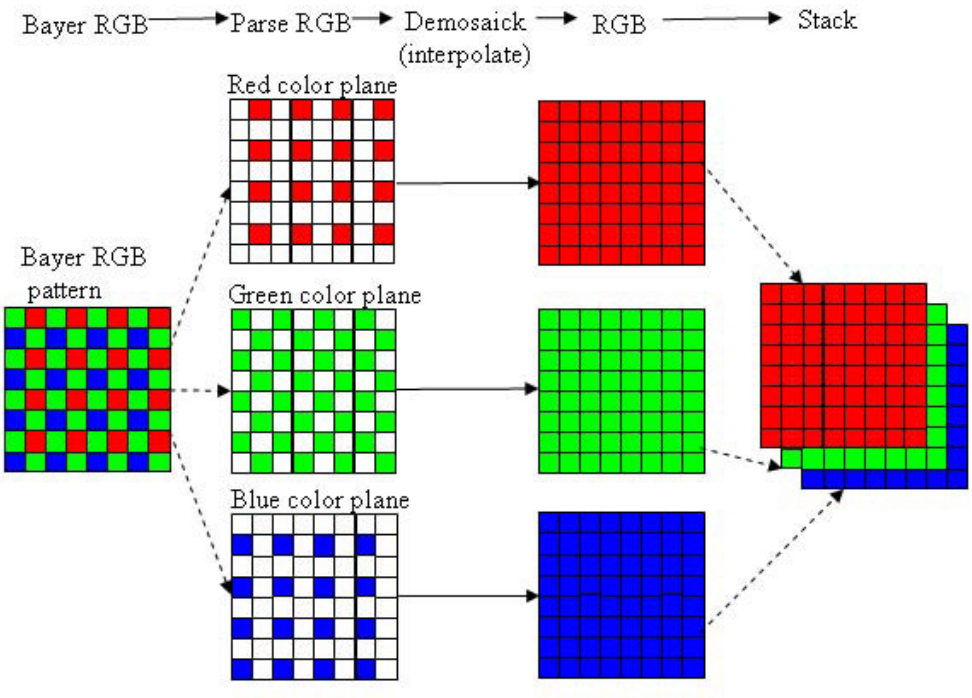

Figure 4: A demosaicking procedure that requires interpolation.

The demosaicking process is illustrated in Fig. 4; the steps in the procedure are as follows:

1. Read in the recorded Bayer CFA array.
2. Then parse the CFA array into three color planes, RGB.
3. Perform different interpolations (with FIR filters) on each of the color planes.
4. Stack the RGB color planes to form a 3D array that can be displayed via `show_img` or `imshow`.

Figure 5 shows the first step of taking the recorded Bayer CFA array (left image) and identifying the RGB color information which is shown in the right image.

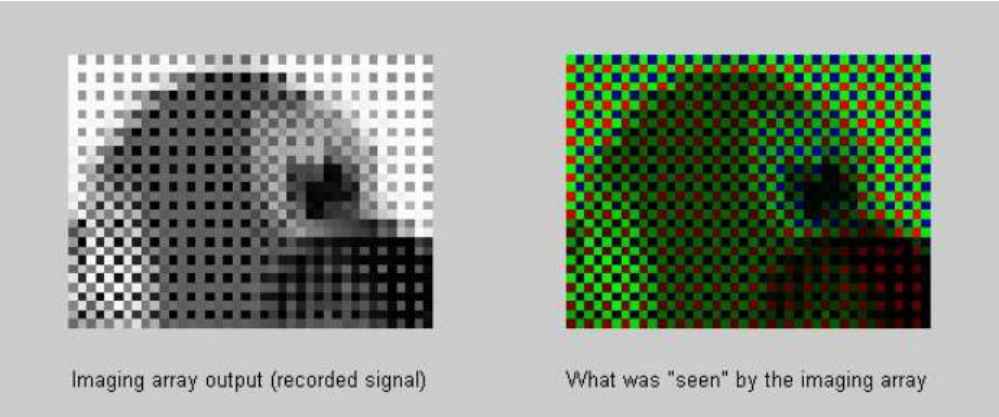

Figure 5: Images at two different stages in the image processing chain: (left) recorded CFA intensities, (right) color image after the RGB pixels have been separated.

Pre-Lab

The goal of this lab is to learn how to implement FIR filters in MATLAB, and then study the response of FIR filters to various signals, including images or speech. As a result, you should learn how filters can create interesting effects such as blurring and echoes. In addition, we will use FIR filters to study the convolution operation and properties such as linearity and time-invariance.

In the experiments of this lab, you will use `firfilt()`, or `conv()`, to implement 1-D filters and `conv2()` to implement two-dimensional (2-D) filters. The 2-D filtering operation actually consists of 1-D filters applied to all the rows of the image and then all the columns.

Discrete-time Convolution GUI

This lab involves the use of a MATLAB GUI for convolution of discrete-time signals, `dconvdemo`. This is exactly the same as the MATLAB functions `conv()` and `firfilt()` used to implement FIR filters. This demo is part of the *SP-First* Toolbox.

Overview of Filtering

For this lab, we will define an FIR *filter* as a discrete-time system that converts an input signal $x[n]$ into an output signal $y[n]$ by means of the weighted summation formula:

$$y[n] = \sum_{k=0}^{M} b_k \, x[n-k] \tag{1}$$

Equation (1) gives a rule for computing the n^{th} value of the output sequence from present and past values of the input sequence. The filter coefficients $\{b_k\}$ are constants that define the filter's behavior. As an example, consider the system for which the output values are given by

$$\begin{aligned} y[n] &= \tfrac{1}{3}x[n] + \tfrac{1}{3}x[n-1] + \tfrac{1}{3}x[n-2] \\ &= \tfrac{1}{3}\left\{x[n] + x[n-1] + x[n-2]\right\} \end{aligned} \tag{2}$$

This equation states that the n^{th} value of the output sequence is the average of the n^{th} value of the input sequence $x[n]$ and the two preceding values, $x[n-1]$ and $x[n-2]$. For this example, the b_k's are $b_0 = \frac{1}{3}$, $b_1 = \frac{1}{3}$, and $b_2 = \frac{1}{3}$.

MATLAB has two built-in functions, `conv()` and `filter()`, for implementing the operation in (1), and the *SP-First* toolbox supplies another M-file, called `firfilt()`, for the special case of FIR filtering. The function `filter` implements a wider class of filters than just the FIR case. Technically speaking, both the `conv` and the `firfilt` function implement the operation called *convolution*. The following MATLAB statements implement the three-point averaging system of (2):

CD-ROM
firfilt.m

```
nn = 0:99;               %<--Time indices
xx = cos( 0.08*pi*nn );  %<--Input signal
bb = [1/3 1/3 1/3];      %<--Filter coefficients
yy = firfilt(bb, xx);    %<--Compute the output
```

In this case, the input signal `xx` is contained in a vector defined by the cosine function. In general, the vector `bb` contains the filter coefficients $\{b_k\}$ needed in (1). The `bb` vector is defined in the following way:

```
bb = [b0, b1, b2, ...  , bM].
```

In MATLAB, all sequences have finite length because they are stored in vectors. If the input signal has L nonzero samples, we would normally store only the L nonzero samples in a vector, and would assume that $x[n] = 0$ for n outside the interval of L samples, i.e., don't store any zero samples unless it suits our purposes. If we process a finite-length signal through (1), then the output sequence $y[n]$ will be longer than $x[n]$ by M samples. Whenever `firfilt()` implements (1), we will find that

```
length(yy) = length(xx)+length(bb)-1
```

In the experiments of this lab, you will use `firfilt()` to implement FIR filters and begin to understand how the filter coefficients define a digital filtering algorithm. In addition, this lab will introduce examples to show how a filter reacts to different frequency components in the input.

Discrete-Time Convolution Demo

The first objective of this lab is to demonstrate usage of the `dconvdemo` GUI. If you have installed the *SP-First* Toolbox, you will already have this demo on the `matlabpath`. In this demo, you can select an input signal $x[n]$, as well as the impulse response of the filter $h[n]$. Then the demo shows the ***sliding window*** view of FIR filtering, where one of the signals must be *flipped and shifted* along the axis when convolution is computed. Figure 6 shows the interface for the `dconvdemo` GUI.
In the pre-lab, you should perform the following steps with the `dconvdemo` GUI.

(a) Click on the `Get x[n]` button and set the input to a finite-length pulse: $x[n] = (u[n] - u[n-10])$. Note the length of this pulse.

(b) Set the filter to a three-point averager by using the `Get h[n]` button to create the correct impulse response for the three-point averager. Remember that the impulse response is identical to the b_k's for an FIR filter. Also, the GUI allows you to modify the length and values of the pulse.

(c) Observe that the GUI produces the output signal in the bottom panel.

(d) When you move the mouse pointer over the index "n" below the signal plot and do a click-hold, you will get a *hand tool* that allows you to move the "n"-pointer to the left or right; or you can use the left and right arrow keys. By moving the pointer horizontally you can observe the sliding window action of convolution. You can even move the index beyond the limits of the window and the plot will scroll over to align with "n."

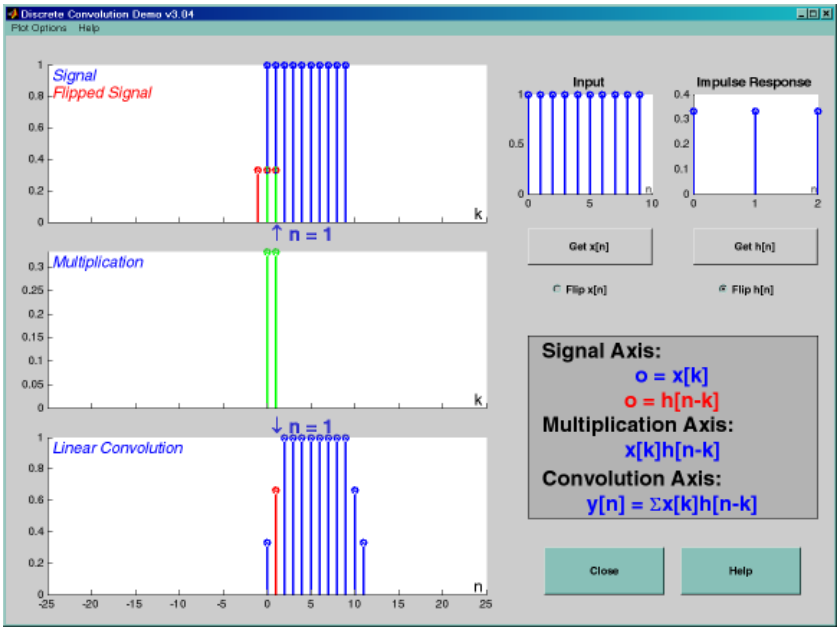

Figure 6: Interface for discrete-time convolution GUI called `dconvdemo`. This is the convolution of a three-point averager with a ten-point rectangular pulse.

Filtering via Convolution

You can perform the same convolution as done by the `dconvdemo` GUI by using the MATLAB function `firfilt`, or `conv`. For ECE-2026, the preferred function is `firfilt`.

(a) For the Pre-Lab, you should do some filtering with a three-point averager. The filter coefficient vector for the three-point averager is defined via:

```
bb = 1/3*ones(1,3);
```

Use `firfilt` to process an input signal that is a length-10 pulse:

$$x[n] = \begin{cases} 1 & \text{for } n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \\ 0 & \text{elsewhere} \end{cases}$$

Note: in MATLAB indexing can be confusing. Our mathematical signal definitions start at $n = 0$, but MATLAB starts its indexing at "1". Nevertheless, we can ignore the difference and pretend that MATLAB is indexing from zero, as long as we don't try to write `x[0]` in MATLAB. For this experiment, generate the length-10 pulse and put it inside of a longer vector with the statement `xx = [ones(1,10),zeros(1,5)]`. This produces a vector of length 15, which has 5 extra zero samples appended.

(b) To illustrate the filtering action of the three-point averager, it is informative to make a plot of the input signal and output signal together. Since $x[n]$ and $y[n]$ are discrete-time signals, a `stem` plot is needed. One way to put the plots together is to use `subplot(2,1,*)` to make a two-panel display:

```
nn = first:last;      %--- use first=1 and last=length(xx)
subplot(2,1,1);
stem(nn-1,xx(nn))
subplot(2,1,2);
stem(nn-1,yy(nn),'filled')   %--Make black dots
xlabel('Time Index (n)')
```

This code assumes that the output from `firfilt` is called `yy`. Try the plot with `first` equal to the beginning index of the input signal, and `last` chosen to be the last index of the input. In other words, the plotting range for both signals will be equal to the length of the input signal, even though the output signal is longer. Notice that using `nn-1` in the two calls to `stem()` causes the x-axis to start at zero in the plot.

(c) Explain the filtering action of the three-point averager by comparing the plots in the previous part. This averaging filter might be called a "smoothing" filter, especially when we see how the transitions in $x[n]$ from zero to one, and from one back to zero, have been "smoothed."

In-Lab Exercises

Discrete-Time Convolution

you will generate filtering results needed in a later section. Use the discrete-time convolution GUI, `dconvdemo`, to do the following:

(a) The convolution of two impulses, $\delta[n-3] * \delta[n-5]$.

(b) Filter the input signal $x[n] = (-3)\{u[n-2] - u[n-8]\}$ with a first-difference filter. Make $x[n]$ by selecting the "Pulse" signal type from the drop-down menu within `Get x[n]`, and also use the text box "Delay."
Next, set the impulse response to match the filter coefficients of the first-difference. Enter the impulse response values by selecting "User Signal" from the drop-down menu within `Get h[n]`. Illustrate the output signal $y[n]$ and write a simple formula for $y[n]$ which should use only two impulses.

(c) Explain why $y[n]$ from the previous part is zero for almost all n.

(d) Convolve two rectangular pulses: one with an amplitude of 2 and a length of 7, the other with an amplitude of 3 and a length of 4. Make a sketch of the output signal, showing its starting and ending points, as well as its maximum amplitude.

(e) State the *length* and *maximum amplitude* of the convolved rectangles.

(f) The first-difference filter can be used to find the *edges* in a signal or in an image. This behavior can be exhibited with the `dconvdemo` GUI. Set the impulse response $h[n] = \delta[n] - \delta[n-1]$. In order to set the input signal $x[n]$, use the *User Input* option to define $x[n]$ via the MATLAB statement `double((sin(0.5*(0:50))+0.2)<0)`, which is a signal that has *runs* of zero and ones.
The output from the convolution $y[n]$ will have only a few nonzero values. Record the locations of the nonzero values, and explain how these are related to the *transitions* in the input signal. Also, explain why some values of $y[n]$ are positive, and others are negative.

Filtering Images via Convolution

One-dimensional FIR filters, such as running averagers and first-difference filters, can be used to process one-dimensional signals such as speech or music. These same filters can be applied to images if we regard each row (or column) of the image as a one-dimensional signal. For example, the 50^{th} row of an image is the N-point sequence `xx[50,n]` for $1 \leq n \leq N$, so we can filter this sequence with a 1-D filter using the `conv` or `firfilt` operator, e.g., to filter the m_0-th row:

$$y_1[m_0, n] = x[m_0, n] - x[m_0, n-1]$$

(a) Load in the image `echart.mat` (from the *SP-First* Toolbox) with the `load` command. This will create the variable `echart` whose size is 257×256. We can filter one row of the image by applying the `conv()` function to one row extract from the image, `echart(m,:)`.

```
bdiffh = [1, -1];
yy1 = conv(echart(m,:), bdiffh);
```

Pick a row where there are several black-white-black transitions, e.g., choose row number 65, 147, or 221. Display the row of the input image `echart` and the filtered row `yy1` on the screen in the same figure window (with `subplot`). Compare the two stem plots and give a qualitative description of what you see. Note that the polarity (positive/negative) of the impulses will denote whether the transition is from white to black, or black to white. Then explain how to calculate the width of the "E" from the impulses in the stem plot of the filtered row.
Note: Use the MATLAB function `find` to get the locations of the impulses in the filtered row `yy1`.

Information About File Formats

Images obtained from JPEG files might come in many different formats. Two precautions are necessary:

1. If MATLAB loads the image and stores it as 8-bit integers, then MATLAB will use an internal data type called `uint8`. The function `show_img( )` cannot handle this format, but there is a conversion function called `double( )` that will convert the 8-bit integers to double-precision floating-point for use with filtering and processing programs.

   ```
   yy = double(xx);
   ```

   You can convert back to 8-bit values with the function `uint8()`.

2. If the image is a color photograph, then it is actually composed of three "image planes" and MATLAB will store it as a 3-D array. For example, the result of `whos` for a 545×668 color image would give:

   ```
   Name       Size          Bytes  Class
   xx      545x668x3    1092180  uint8 array
   ```

   In this case, you should use MATLAB's image display functions such as `imshow( )` to see the color image. Or you can convert the color image to gray-scale with the function `rgb2gray( )`. For more information on the image processing functions in MATLAB, try help:

   ```
   help images
   ```