# IMAGE PROCESSING 9

Electrical Engineering 20N
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

HSIN-I LIU, JONATHAN KOTKER, HOWARD LEI, AND BABAK AYAZIFAR

## 1 Introduction

Some of the most popular applications of digital signal processing occur in the growing field of image processing. Unlike audio signals, however, image signals exist in two dimensions, as opposed to one: this fact makes their processing more complicated. In this lab session, we will implement basic image processing in LabVIEW; specifically, we will focus on generating image signals in LabVIEWand extrapolate to generating two-dimensional periodic patterns. We will also employ the two-dimensional variants of several well-known one-dimensional filters on images, and explore and tweak their effects.

### 1.1 Lab Goals

- Extend various filters and concepts in digital signal processing to image processing.
- Explore the analogies and the connections between temporal signals and spatial signals, such as images.
- Use the file processing mechanisms available in LabVIEW.
- Use and optimize image processing algorithms.

### 1.2 Checkoff Points

1. Pre-Lab Section . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
   (a) Hexadecimal Number System: A Primer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
   (b) There's an Image . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
   (c) All the Pretty Colors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
   (d) RGB Ain't Got Nothin' On Me . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . (2.5%)
   (e) Phantom of the Colormap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . (7.5%)
   (f) Submission Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
   (g) Submission Instructions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

2. In-Lab Section
   (a) Do You Have Your 2D Glasses?
   (b) It's a Sine of Things to Come ... (45 minutes, 20%)
   (c) Image Processing 101
      i. Living on the Edge
      ii. 2D Convolution
      iii. Getting an Edge Start
      iv. All a Bit of a Blur ... (45 minutes, 15%)
      v. Playing Detective ... (45 minutes, 15%)
3. Parting Words
4. Acknowledgments
5. References

# 2 Pre-Lab Section

## 2.1 Hexadecimal Number System: A Primer

We live our daily lives in the **decimal number system**: we do all of our mathematical operations using the digits `0` through `9`. However, for several other purposes, other number systems are much more useful. For example, you might have seen the **binary number system** before, where all mathematical operations are done using the digits `0` and `1`[1]. In the case of image processing, the **hexadecimal number system** is the most useful number system, where digits go from `0` to `9`, and then from `A` to `F`: `A` would represent the number `10` in the decimal system, while `F` would represent the number `15` in the decimal system.

DECIMAL

BINARY

HEXADECIMAL

A number written in the decimal number system is evaluated based on the following rule: the $i^{\text{th}}$ digit (from the right) $A_i$ contributes $A_i \times 10^{(i-1)}$ to the final value. For example, the number `12345` is equivalent to

$$(1 \cdot 10^4) + (2 \cdot 10^3) + (3 \cdot 10^2) + (4 \cdot 10^1) + (5 \cdot 10^0).$$

Analogously, a number written in the hexadecimal number system is evaluated based on the following rule: the $i^{\text{th}}$ digit (from the right) $A_i$ contributes $A_i \times 16^{(i-1)}$ to the final value. For example, the number `C0FF3E` is equivalent to

$$(C \cdot 16^5) + (0 \cdot 16^4) + (F \cdot 16^3) + (F \cdot 16^2) + (3 \cdot 16^1) + (E \cdot 16^0).$$

This allows us to convert from the hexadecimal number system to the decimal number system.

Moving from the decimal number system to the hexadecimal number system is slightly trickier: we will need to represent that number using a linear combination of appropriately scaled powers of 16. Thus, for example, the number `23456`, represented in the decimal number system, can be broken up as

$$23456 = (5 \cdot 16^3) + (B \cdot 16^2) + (A \cdot 16^1) + (0 \cdot 16^0).$$

As a result, `23456`, when represented in the hexadecimal number system, turns out to be `5BA0`. (Verify that this is true!)

[1]There is a well-known joke that goes, "There are 10 types of people in this world: people who can count in binary, and people who cannot."

For the rest of this lab guide, we will follow the convention that numbers in the decimal number system will be suffixed with the subscript `10`: for example, $\texttt{12345}_{10}$, while numbers in the hexadecimal number system will be suffixed with the subscript `16`: for example, $\texttt{C0FF3E}_{16}$.

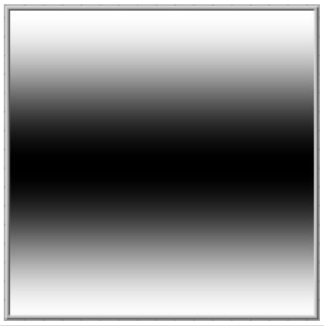
Convert the following numbers between the decimal and hexadecimal number systems, as appropriate:

1. $\texttt{FF}_{16}$.
2. $\texttt{D3AD}_{16}$.
3. $\texttt{15}_{16}$.
4. $\texttt{15}_{10}$.
5. $\texttt{16}_{10}$.
6. $\texttt{512}_{10}$.

It turns out that numbers represented in the binary number system can be succinctly represented in the hexadecimal number system, since a collection of four **bits** (or **bi**nary digi**ts**) in the binary number system can be replaced by one digit in the hexadecimal number system. As a result, the number $\texttt{1001}_2$ can be expressed as $\texttt{9}_{16}$, the number $\texttt{1111}_2$ can be expressed as $\texttt{F}_{16}$, and the number $\texttt{10011111}_2$ can be expressed as $\texttt{9F}_{16}$. BITS

## 2.2 There's an Image

**Figure 1** An image where the intensity varies sinusoidally in the vertical direction.



A two-dimensional image is represented in LabVIEW as a two-dimensional array, where element $(i, j)$ stores information about the element of the picture, or the **pixel** (**pic**s + **el**ement), at location $(i, j)$. Figure 1 shows a grayscale image, with dimensions 200 pixels by 200 pixels (under a zooming factor of 100%), where the intensity of the image varies sinusoidally. The top row of pixels in the image is white. As we move down the image, it gradually changes to black and then back to white, completing one cycle. Hence, we can state that the **vertical frequency** is $\frac{1}{200}$ cycles per pixel[2]. Similarly, the image is constant horizontally, and so it has a **horizontal frequency** of 0 cycles per pixel. PIXEL VERTICAL FREQUENCY HORIZONTAL FREQUENCY

## 2.3 All the Pretty Colors

Colors can be represented by a **hexadecimal triplet**: three hexadecimal pairs (which is equivalent to 24 bits; why?) that are arranged in the order red, green and blue. Thus, a color of $\texttt{FF0000}_{16}$ is entirely red, while $\texttt{0000FF}_{16}$ is blue. The number $\texttt{FF}_{16}$ corresponds to $\texttt{255}_{10}$, which is $2^8 - 1$. An image is then simply a grid of these color values. This hexadecimal triplet representation is known as **TrueColor** and is shown for a TRUECOLOR

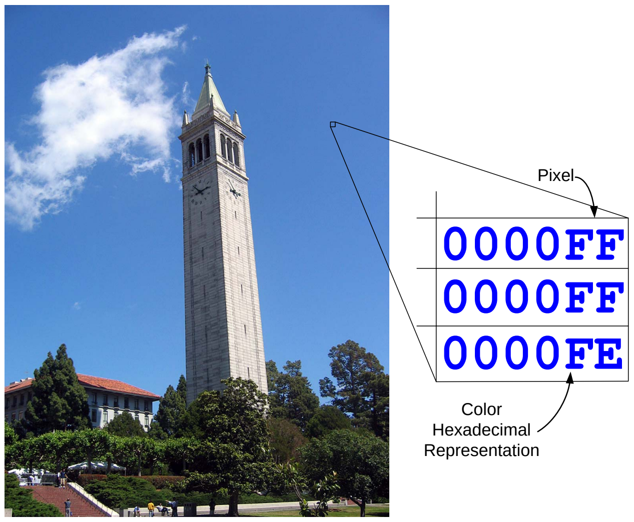
[2]Notice that this is a different kind of frequency than the one we normally discuss: when we talk about signals in the time domain, we are talking about **temporal frequency**; here, we are talking about **spatial frequency**.

few common colors in Figure 2. An illustration showing the TrueColor representations for a few pixels in an image is shown in Figure 3. This representation is also the standard for webpages: HTML, for example, utilizes the hexadecimal triplet.

**Figure 2** The TrueColor representation of a few colors.

| COLOR | RED | GREEN | BLUE |
|---|---|---|---|
| Red | FF | 00 | 00 |
| Blue | 00 | 00 | FF |
| Teal | 38 | 8E | 8E |

**Figure 3** The TrueColor representation of a few pixels in an image. (Note that the color values are not precise and are only present for demonstrative purposes.) [1]

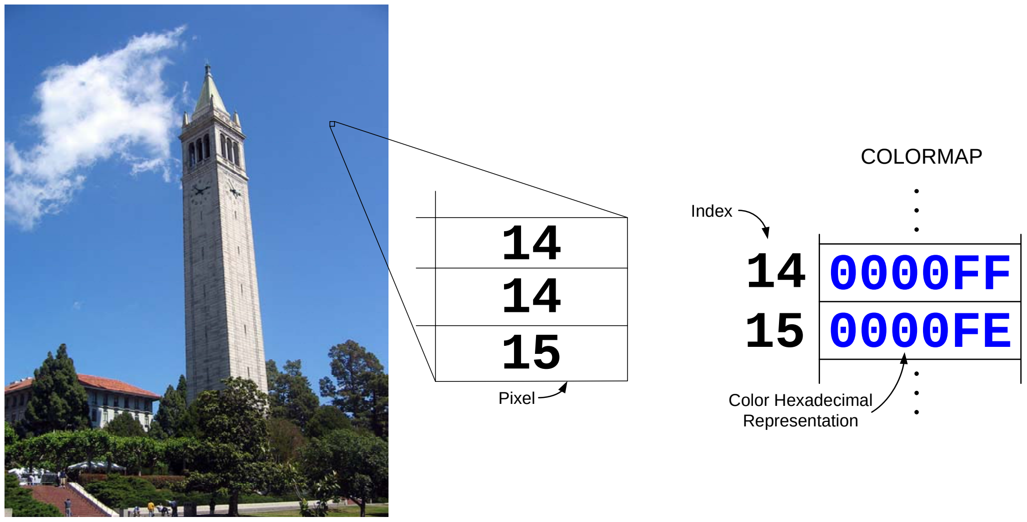


While the image blocks in LabVIEW use TrueColor by default, usually not all the colors are used. Hence, in order to conserve memory, **colormaps** are used. Colormaps provide a very simple form of image compression. A colormap is simply a table of color values. Images will thus no longer need to hold the entire color representation, but a smaller number that references a location in the colormap. This smaller number can then be used to index into the colormap to obtain the actual color representation. This representation is called the **bitmap** representation; image files with the extension BMP are represented in this fashion: a two-dimensional array along with a colormap. An illustration of this representation is shown in Figure 4; contrast this with the basic TrueColor representation illustrated in Figure 3.

COLORMAPS

BITMAP

**Figure 4** The bitmap representation of a few pixels in an image. (Note that the color values are not precise and are only present for demonstrative purposes.) [1]



### 2.4 RGB Ain't Got Nothin' On Me

Download the `ColorRepresentation` VI from the course lab page and run the VI. This VI obtains the corresponding TrueColor representation for a given color. Click on the color in `Color to Number` to change the color and see its TrueColor representation. Ignore the `Luminance` value.

For each of the colors below, determine the RGB values, and also explain briefly why the RGB values returned by the VI are intuitively the correct values. For example, green has the hexadecimal value $\texttt{00FF00}_{16}$; this makes intuitive sense because only the green component is present in the color.

1. White
2. Black
3. Orange
4. Purple

### 2.5 Phantom of the Colormap

The color black has no contribution from either red, green, or blue. If we increase the contribution of all of the colors equally, we will get progressively lighter shades of gray. While the hexadecimal triplet notation provides us with the ability to represent many colors, the in-lab session will utilize a grayscale colormap.

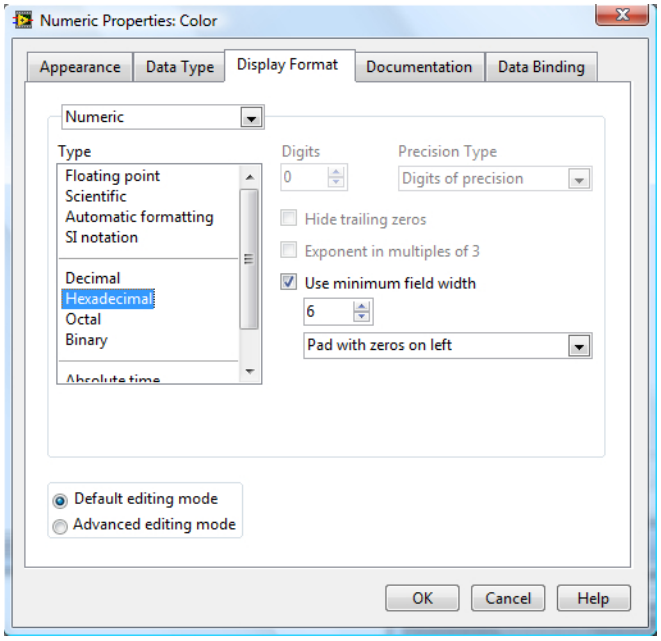
Grayscale images use an 8-bit colormap, which can store a maximum of 256 mappings (why?) between array locations and colors. In each location in a grayscale colormap, the red, green and blue components are all equally intense. The three components each vary from $0_{10}$ (or $0_{16}$) to $255_{10}$ (or $\texttt{FF}_{16}$), representing 256 variations of gray. As a result, the grayscale colormap will be a mapping between the $i^{\text{th}}$ index and the $i^{\text{th}}$ variation of gray.

1. Create an 8-bit grayscale colormap in a VI called `Colormap`. In order to accomplish this, create an array of 256 elements. Each element should contain a color, whose hexadecimal representation has the same red, green and blue values, such as $\texttt{000000}_{16}$, $\texttt{010101}_{16}$, $\texttt{020202}_{16}$, and so on. These colors should go up to $\texttt{FFFFFF}_{16}$. You may find the `RGB to Color` block, located under `Programming`

→ `Graphics and Sound` → `Picture Functions`, useful; this block accepts the red, green and blue components separately, and concatenates them to produce the corresponding color. Call this array `Colormap`.

You will want to create an indicator for this array on the front panel. However, the indicator will, by default, choose to display the values in the array in the decimal number system. To change this, we need to first change the datatype of the array elements to be unsigned 32-bit integers. Right-click on an element, select `Representation`, and then change the datatype to `U32`. Following this, right-click again on an element in the array and select `Display Format...`. Play around with the options under the `Display Format` tab, as shown in Figure 5 so that finally, the front panel representation of each element in the array resembles a hexadecimal triplet.

**Figure 5** The `Display Format` dialog box.


Numeric Properties: Color

Tabs: Appearance | Data Type | Display Format | Documentation | Data Binding

Numeric

Type: Floating point, Scientific, Automatic formatting, SI notation, Decimal, Hexadecimal (selected), Octal, Binary, Absolute time

Digits: 0

Precision Type: Digits of precision

Hide trailing zeros [ ]

Exponent in multiples of 3 [ ]

Use minimum field width [✓]

6

Pad with zeros on left

Default editing mode [●]

Advanced editing mode [ ]

OK | Cancel | Help

2. Convert this VI into a sub-VI with no inputs, but with the array generated in step 1 as the sole output.
3. Create a new VI called `Phantom`. In this VI, create a two-dimensional array of arbitrary intensities of gray. The array should have dimensions 200 $\times$ 200 and have values ranging from 0 to 255. If you implement this array using `MathScript`, you may find the function `rand` useful. You can also choose to generate this two-dimensional array using other blocks that involve matrix operations.
4. The `Draw Unflattened Pixmap` block converts a two-dimensional array into a picture, using a colormap to determine what color each element in the two-dimensional array corresponds to. We will now convert the two-dimensional array that we created in step 3 to an image using the colormap that we generated in step 2. Bring a `Draw Unflattened Pixmap` block to the block diagram, and feed this array in to the `data` terminal of the `Draw Unflattened Pixmap` block.

   Make use of the in-built search functionality to determine where the `Draw Unflattened Pixmap` block is located.
5. Set the `Draw Unflattened Pixmap` block to an 8-bit representation (under `Select Type` on the right-click menu) and wire the grayscale colormap to the `color table` port on the block.

   The colormap should be explicitly specified as 8-bit, or else LabVIEW will assume that the colormap contains more values than it actually does.

6. Create an indicator for the `image` port on the front panel, and run the virtual instrument. You may need to change the size of the picture indicator to show the entire image.

## 2.6 Submission Rules

1. Submit your files *no later than* 10 minutes after the beginning of your next lab session, during the week of **April 25, 2011**.
2. Late submissions will *not* be accepted, except under unusual circumstances.
3. If the pre-lab exercises are not performed, you will get an immediate zero for the entire lab.
4. These exercises should be done *individually.*
5. Keep your work safe for further usage in the in-lab sections.

## 2.7 Submission Instructions

1. Log on to bSpace and click on the `Assignments` tab.
2. Locate the assignment for `Lab 9 Pre-Lab` corresponding to your section.
3. Attach the following files to the assignment:
   (a) A text document called `Responses`, containing your responses to the questions in section 2.1 and section 2.4.
   (b) The `Colormap` VI.
   (c) The `Phantom` VI.

**Figure 6** Love in hex. [3]

# 3 In-Lab Section

In the pre-lab sections, we had a chance to retrieve the RGB values for different colors and we also had a little taste of creating an image. For the most part, this lab guide relies on your creativity in implementation, so go crazy and perform the tasks in whatever manner you please, but do not forget to maintain clarity and optimality.

## 3.1 Do You Have Your 2D Glasses?

As we move through the lab, we will take advantage of an important and exciting idea: if we consider one row (or equivalently, one column) of an image, we are essentially looking at a discrete-time signal having various values at different points in time! Of course, we are not working in the discrete-time domain at all, but we *are* working in a discretized domain, where the domain is the set of integers; in this case, the domain consists of pixels. As a result, we can utilize the ideas that we have learned through our excursions in digital signal processing to implement solutions to interesting image processing problems.

There is, however, an added complexity that comes with working with images: the *whole* image (and not just one row/column) is actually a signal that works in *two* dimensions, not one. Nonetheless, as we shall see, concepts in one-dimensional signal processing extend neatly into two-dimensional signal processing.

## 3.2 It's a Sine of Things to Come

1. Make a new copy of the `Phantom` VI that you created in step 3 of pre-lab section 2.5, and save it as the `Sinusoids` VI.

2. Modify the arbitrary 200 $\times$ 200 pixel image that you generated to produce an image exactly as that shown in Figure 1.

   - We know that the value of each pixel varies from `0` to `255`, reflecting the number of colors represented in the grayscale colormap. Which value corresponds to white and which value corresponds to black?
   - If done correctly, your image should match Figure 1. If the middle patch of black is too wide, this might be because the values for some of the pixels are either going below `0` or above `255`, in which case such pixels are rendered as completely black.
   - We observed that the intensities of the pixels in Figure 1 vary sinusoidally in the vertical direction from white to black, and then back to white. Using this fact, determine the mathematical function $I(i)$ that best relates row $i$ to the intensity $I$ of the pixels in that row. This will be the signal that you have to implement.
   - There are, of course, several ways to approach this question. However, as in previous labs, we encourage you to consider not using `For Loops`, either as structures or as constructs in `MathScript Nodes`. There are more elegant (and faster!) methods that exploit the matrix operations present in LabVIEW.
   - Again, you may find the `repmat` function useful. Also, `A'` produces the transpose of matrix `A`.

3. Duplicate the vertically varying image that you created in step 2 and modify it so that the sinusoidal variations are horizontal rather than vertical. Additionally, vary the frequency so that there are four cycles of the sinusoidal variations instead of one.

4. Duplicate the horizontally varying image that you created in step 3, and add the necessarily logic that enables a horizontal slider on the front panel to vary the frequency of the image, such that the number of cycles of sinusoidal variations can be set to any real value from `1` to `20`. Label this horizonal slider `Cycles`.

## 3.3 Image Processing 101

Now that we have created custom images, the next step is to manipulate images, and along the way, obtain a taster of basic image processing.

### 3.3.1 Living on the Edge

Continuing with our analogy of images as two-dimensional signals, we now attempt to analyze the "frequency content" of an image, although in the case of images, we talk about spatial frequency, instead of temporal frequency. We make the observation that parts of an image where pixel intensities vary gradually are regions of "low frequency", since the image does not vary much in these regions, whereas parts where pixel intensities vary suddenly are regions of high frequency. With this in hand, we then have the stunning idea that we can apply filters to images to extract images of different frequency contents, just as we apply filters to discrete-time signals to extract signals of different frequency contents!

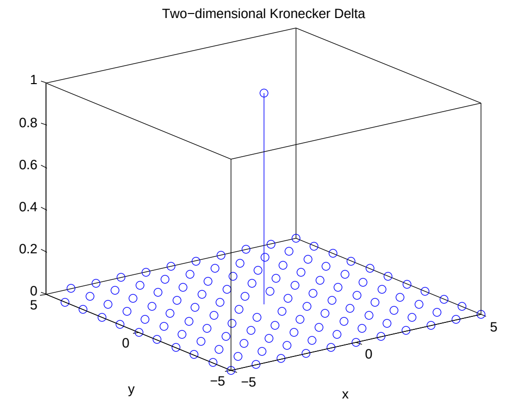
In particular, we can apply these ideas on to the sharp edges in an image, which are, by their very nature, areas that are rich in the high frequencies. In this section, we will see that, depending on the kind of filter we use, we can either blur an image or perform basic edge detection.

### 3.3.2 2D Convolution

For one-dimensional signals, we know that every LTI system can be uniquely represented by its impulse response. We also know the impulse response proves useful in determining the output signal of the system, because the output signal is merely the input signal convolved with the impulse response.

Very similar concepts arise in the analysis of two-dimensional signals. For instance, we define the **two-dimensional impulse (Kronecker delta)** as the signal $\delta(m, n)$ that is 1 when $m$ and $n$ are *both* zero, and 0 otherwise. A three-dimensional plot of the two-dimensional Kronecker delta is shown in Figure 7. 2D IMPULSE

**Figure 7** The two-dimensional Kronecker delta.



The impulse response of an LTI system H operating on two-dimensional signals is then defined as its response to the two-dimensional Kronecker delta, denoted by $h(m, n)$. Indeed, we can now go so far as to define **two-dimensional convolution**. If we have two 2D signals, $x(m, n)$ and $h(m, n)$, then the convolution of the two signals is defined as 2D CONVOLUTION

$$(x * h)(m,n) = \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} h(k,\ell)\, x(m-k, n-\ell) = \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} x(k,\ell)\, h(m-k, n-\ell).$$

This is, of course, the output of the LTI system H in response to the input $x(m,n)$. However, we will not attempt to manually perform 2D convolution; the details are beyond the scope of the course. We will use functionality already present in LabVIEW to perform 2D convolution. Nonetheless, we need to determine what signals we need to convolve, as we will shortly.

#### 3.3.3 Getting an Edge Start [2]

1. Download the `ImgData` VI from the lab 9 resources on the course lab page. This VI will provide the image data (the two-dimensional 8-bit array) you will be using.
2. Create a new VI called `Image Processing`. In the block diagram, place the `ImgData` VI as a sub-VI.
3. At the `Data` output port of the `ImgData` sub-VI, you can retrieve the byte array representing an image. Using this array output, show the image on the front panel and label it as the `Original Image`. Resize the picture indicator on the front panel to be about 250 pixels by 250 pixels. (Make sure that you are using your `Colormap` VI with a `Draw Unflattened Pixmap` block, and also make sure that the `Draw Unflattened Pixmap` block is specified as 8-bit, or else your image will appear blue.)

   If everything is done correctly, the image should display a message. Try following what it says.

#### 3.3.4 All a Bit of a Blur

Since sharp edges have rich content in the high frequencies, we can blur these edges simply by amplifying the lower frequency content of the image and diminishing its higher frequency content. If we perform this across all pixels in the image, every pixel with content in higher frequencies should be affected, not just edges, resulting in the entire image becoming blurred. The filter that should now spring to mind is the low-pass filter, whose one-dimensional variant we have already seen:

$$y(n) = \frac{1}{2}\left(x(n) + x(n-1)\right).$$

**Another Way of Looking at a Low-Pass Filter** Notice that, from the LCCDE for a low-pass filter, the value of the output signal at every point in time is the average of the value of the input signal at that point in time, and the value of the input signal at the previous point in time; in other words, the low-pass filter essentially performs a **moving average**. With this observation, we realize that we can blur an image simply by performing a moving average over that image. For this lab, we will perform a 5×5 moving average on a given image, with the result that the intensity of each pixel in the output image is the average of the intensities of the (at most) 25 pixels in the 5×5 square centered at the corresponding pixel in the input image. Intuitively, a moving average smoothens out sharp changes in the intensities of different pixels.

**How To Perform the Moving Average** We will perform this moving average through 2D convolution: we will convolve our two-dimensional signal (the image) with a two-dimensional impulse response to produce another two-dimensional signal (the blurred image). The task is now to create this impulse response.

As discussed previously, the value of the output signal at any given point $(m,n)$ is the average of the surrounding pixels in the original image within a 5×5 window:

$$y(m,n) = \frac{1}{25} \sum_{k=-2}^{2} \sum_{\ell=-2}^{2} x(m-k,\, n-\ell),$$

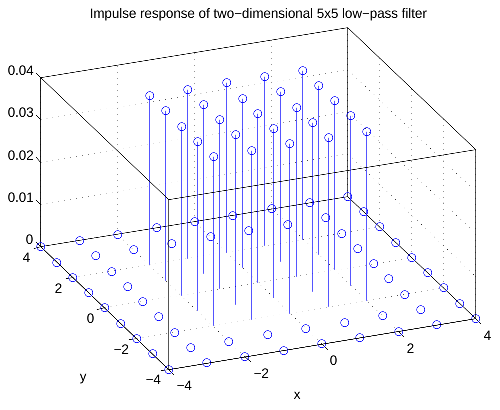
and in general, for an $N \times N$ window, where $N$ is odd:

$$\begin{aligned} y(m,\, n) &= \frac{1}{N^2} \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} x(m-k,\, n-\ell) \\ &= \frac{1}{N^2} \sum_{k=-\lfloor \frac{N}{2} \rfloor}^{\lfloor \frac{N}{2} \rfloor} \sum_{\ell=-\lfloor \frac{N}{2} \rfloor}^{\lfloor \frac{N}{2} \rfloor} x(m-k,\, n-\ell). \end{aligned}$$

where $\lfloor \cdot \rfloor$ is the flooring function.

**So What Did You Want Me To Do?** Determine the impulse response $h_{\mathsf{L}}(m,n)$ of a system $\mathsf{H_L}$ that performs the $5{\times}5$ moving average of its input, with the help of the equations from the previous paragraph. As a hint, remember how you would often find the impulse response of LTI systems acting on one-dimensional signals: what special signal would you set $x(n)$ to be? If derived correctly, a plot of the resulting impulse response should match the one shown in Figure 8.

**Figure 8** Impulse response of the two-dimensional $5{\times}5$ low-pass filter $\mathsf{H_L}$.



Then, create a matrix that represents this impulse response. Note that even though the impulse response is theoretically infinite in span, you only need to create a matrix for its **support**, the region where the impulse response is nonzero; the zeros elsewhere are implied. In this case, for example, you will only need to create a $5{\times}5$ matrix, where the value of the impulse response at $(0,0)$ is at the center: SUPPORT

$$\begin{bmatrix} h_{\mathsf{L}}(-2,2) & h_{\mathsf{L}}(-1,2) & h_{\mathsf{L}}(0,2) & h_{\mathsf{L}}(1,2) & h_{\mathsf{L}}(2,2) \\ h_{\mathsf{L}}(-2,1) & h_{\mathsf{L}}(-1,1) & h_{\mathsf{L}}(0,1) & h_{\mathsf{L}}(1,1) & h_{\mathsf{L}}(2,1) \\ h_{\mathsf{L}}(-2,0) & h_{\mathsf{L}}(-1,0) & h_{\mathsf{L}}(0,0) & h_{\mathsf{L}}(1,0) & h_{\mathsf{L}}(2,0) \\ h_{\mathsf{L}}(-2,-1) & h_{\mathsf{L}}(-1,-1) & h_{\mathsf{L}}(0,-1) & h_{\mathsf{L}}(1,-1) & h_{\mathsf{L}}(2,-1) \\ h_{\mathsf{L}}(-2,-2) & h_{\mathsf{L}}(-1,-2) & h_{\mathsf{L}}(0,-2) & h_{\mathsf{L}}(1,-2) & h_{\mathsf{L}}(2,-2) \end{bmatrix}$$

When creating your matrix, replace the entries in the matrix above with the corresponding actual values.

Finally, blur the image that you obtained in section 3.3.3 by convolving it with this impulse response, and display the output image as the `Blurred Image`. The output image should have the same dimensions as the input image.

When done, explore the blurring effect with different window sizes: 3×3, 7×7, and 9×9. You can be ambitious and include a slider that changes $N$, the size of the blurring window, although bear in mind that the operation becomes slower with larger blurring windows.

**Helpful Tools** We recommend using `MathScript Node`s to achieve the intended result, although there are blocks under `Programming` $\rightarrow$ `Array` available that are equally capable of performing the job. You may find the following `MathScript` constructs and functions useful. Use LabVIEW Help to get familiar with the functions:

1. The `conv2d` function.
2. The `zeros` and `ones` functions.
3. Recall from lab 04 that if we have $n$ one-dimensional arrays `A1`, `A2`, ..., `An`, we can concatenate these arrays into one one-dimensional array `A` using the construct

   ```
   A = [ A1   A2   A3   ...   An ];
   ```

   Alternatively, if all of the one-dimensional arrays are of the same dimension, then we can stack these arrays one above another to create a matrix `A` using the construct

   ```
   A = [ A1;   A2;   A3;   ...   An ];
   ```

   In this context, the `;` character separates the different rows of the matrix.

### 3.3.5 Playing Detective

Since sharp edges have rich content in the high frequencies, we can detect these edges simply by amplifying the higher frequency content of an image and diminishing its lower frequency content[3]. If we perform this across all pixels in the image, every pixel with content in higher frequencies should be affected, not just edges, resulting in an output image where only the edges are visible. The filter that should now spring to mind is the high-pass filter, whose one-dimensional variant we have already seen:

$$y(n) = \frac{1}{2}\left(x(n) - x(n-1)\right).$$

**Another Way of Looking at a High-Pass Filter** Notice that, from the LCCDE for a high-pass filter, the value of the output signal at every point in time is merely the difference between the value of the input signal at that point in time, and the value of the input signal at the previous point in time; in other words, the high-pass filter essentially performs a **moving difference**. With this observation, we realize that we can detect the edges in an image simply by performing a moving difference over that image: we determine the difference between two adjacent pixels, and if the *absolute* difference exceeds a certain (changeable) threshold, we 'let that pixel through'. This is the approach that we will be following in this section.

[3]Déjà vu?

**How To Perform the Moving Difference** We will compare the intensity of each pixel to the intensity of the pixels *to its east and to its north* (where applicable), and if *either* of the absolute differences (either the difference between the current pixel and the pixel to its east, or the difference between the current pixel and the pixel to its north) exceeds a certain threshold, the corresponding pixel should be made black; else, it will be set to be white. We will also need to use a slider on the front panel to determine the threshold to be used: **what are the bounds on this slider?**

**So What Did You Want Me To Do?** Perform the basic edge detection algorithm presented above on the image that we obtained in section 3.3.3. Again, we recommend using `MathScript Node`s to achieve the intended result, although there are blocks under `Programming` $\rightarrow$ `Array` available that are equally capable of performing the job. Please be careful when performing the steps below; each step depends on the correctness of the previous.

1. Determine the impulse response $h_{\mathsf{HN}}(m,n)$ of a system $\mathsf{H}_{\mathsf{HN}}$ that, for each pixel, determines the difference in intensities of that pixel and of the pixel to its north. In other words, determine the impulse response of the system
$$y(m,n) = x(m,n) - x(m,n+1).$$
Also, determine the impulse response $h_{\mathsf{HE}}(m,n)$ of a similar system $\mathsf{H}_{\mathsf{HE}}$ that, for each pixel, determines the difference in intensities of that pixel and of the pixel to its east. In other words, determine the impulse response of the system
$$y(m,n) = x(m,n) - x(m+1,n).$$

2. Generate two 3×3 matrices that represent these impulse responses, similar to the matrices that you generated for the impulse response of the two-dimensional low-pass filter. Thus, each matrix should have the form
$$\begin{bmatrix} h(-1,1) & h(0,1) & h(1,1) \\ h(-1,0) & h(0,0) & h(1,0) \\ h(-1,-1) & h(0,-1) & h(1,-1) \end{bmatrix},$$
where $h$ is either $h_{\mathsf{HN}}$ or $h_{\mathsf{HE}}$.

3. Convolve the input signal (the input image) with each of these impulse responses to produce two different output signals (of the same dimensions).

4. Modify the resulting signals, remembering that we need the *absolute* differences in intensities. Call these new signals $y_{\mathsf{HN}}(m,n)$ and $y_{\mathsf{HE}}(m,n)$.

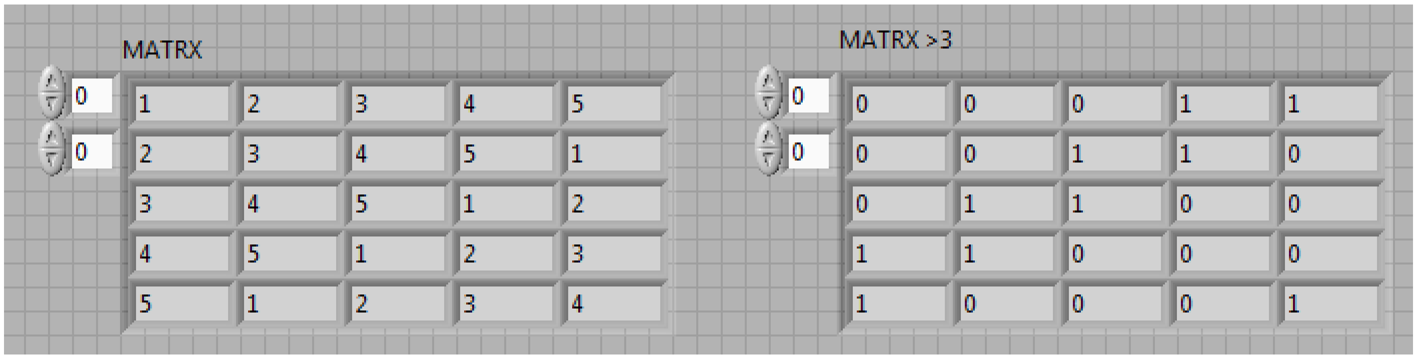
5. Create a slider to manipulate the threshold, with the bounds that you determined in the previous section.

6. Using the signal $y_{\mathsf{HN}}(m,n)$, generate another signal $z_{\mathsf{HN}}(m,n)$ that is `1` wherever $y_{\mathsf{HN}}(m,n)$ is larger than the provided threshold, and `0` otherwise. See figure 9 for an example showing the output of $MATRX > 3$.

   The `MathScript` command `M > T` will check whether each element in the matrix `M` is greater than the scalar `T`: if so, the resulting matrix will have a `1` in the corresponding position; else, the resulting matrix will have a `0` in the corresponding position.

   After this point, we have a signal that is `1` wherever the intensity of the corresponding pixel in the original image differed from the intensity of its neighboring pixel in the north by a value beyond the selected threshold. Generate a similar signal $z_{\mathsf{HE}}(m,n)$ from $y_{\mathsf{HE}}(m,n)$.

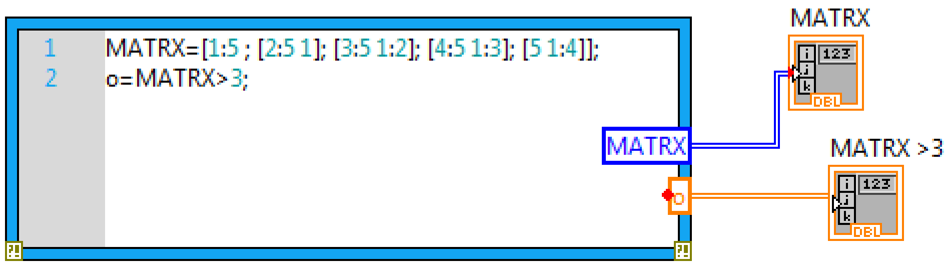
**Figure 9** *Thresholding*: Example of how to use the greater than operator using a matrix and a scalar


MATRX

|  |  |  |  |  |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 3 | 4 | 5 | 1 |
| 3 | 4 | 5 | 1 | 2 |
| 4 | 5 | 1 | 2 | 3 |
| 5 | 1 | 2 | 3 | 4 |

MATRX >3

|  |  |  |  |  |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |


```
MATRX=[1:5 ; [2:5 1]; [3:5 1:2]; [4:5 1:3]; [5 1:4]];
o=MATRX>3;
```

7. Finally, generate a signal $z$ which is `1` at pixel $(m, n)$ if either $z_{\text{HE}}(m, n)$ or $z_{\text{HN}}(m, n)$ is `1`.

   Boolean operations can also be performed pointwise between two matrices. For example, the command `M | N` performs the pointwise logical `OR` of the two matrices `M` and `N`. The logical `OR` of `0` with a nonzero value is `1`, while the logical `OR` of `0` with itself is `0`.

8. We now arrive at a small technicality: the color `255` represents white, whereas the color `0` represents black. Currently, the signal $z$ is `0` where it should be white, and `1` where it should be black. Modify the signal $z$ to produce $z'$, such that $z'(m, n)$ is `1` wherever $z(m, n)$ is `0`, and `0` wherever $z(m, n)$ is `1`. You can do this by subtracting the signal $z$ from a well-chosen scalar.

9. Finally, scale $z'(m, n)$ by `255` to obtain the edge-detected output image. Display the output image as the `Edges in Image` and find an appropriate threshold for the image provided.

**Applications of Edge Detection** Edge detection is often the first step in **image understanding**, which is the automatic interpretation of images. A common application of image understanding is **optical character recognition** or **OCR**, which is the transcription of printed documents into computer documents. Edge detection for different purposes, however, is still very much an open problem and an active area of research.

## 4 Parting Words

Now that we are at the end of the lab-based portion of Electrical Engineering 20N (affectionately known as "EE20N"), let us take stock of what you have managed to accomplish so far. You started the semester learning about mathematical constructs and the basic arithmetic of complex numbers, and from that you extrapolated into determining frequency responses. With these simple ideas, you constructed filters, and with these filters, you did some amazing things: you filtered an ECG signal, you created a guitar string and even made a song, you manipulated voices and filtered frog voices, and finally, you just performed basic image processing. You wrestled with the idea and implementation of convolution, and you wrestled with the ideas and implementations of various filters. With simple ideas, you tackled some of the biggest real-world problems; hopefully, we have now intrigued you, even just a teeny bit, to the power of digital signal processing. All things considered, however, it has been a pleasure having you aboard for EE20N this semester. The teaching staff hopes that you have learned a lot about signal processing and system analysis through this course, and should you wish to continue in this branch of Electrical Engineering, we suggest taking EE120 and EE123 after this course. We would also like to thank you because your input, however small or large, was considered in an attempt to make these labs all the better, more relevant, more interesting, and more fun for you to complete. Thank you, and good luck. ⌣̈

## 5 Acknowledgments

Special thanks go out to the teaching assistants (TAs) of the Spring 2009 semester (Vinay Raj Hampapur, Miklos Christine, Sarah Wodin-Schwartz), of the Fall 2009 semester (David Carlton, Judy Hoffman, Mark Landry, Feng Pan, Changho Suh), and of the Spring 2010 semester (Xuan Fan, Brian Lambson, Kelvin So) for providing suggestions, ideas, and fixes to this lab guide. This lab guide was based on, although substantially modified from, the "Images" laboratory exercise as presented in the book `Structure and Interpretation of Signals and Systems`, written by Edward A. Lee and Pravin Varaiya (ISBN 0201745518).

## References

[1] Roy Tennant. FreeLargePhotos.com. `http://www.freelargephotos.com`, 2009.

[2] Andrew Toulouse. *Babak Orama*. `http://www.facebook.com`, 2008. *I Got Babakued* Facebook group.

[3] `http://scienceblogs.com/retrospectacle/all%20my%20base%20cake.bmp`.