Armstrong Atlantic State University
Engineering Studies
MATLAB Marina – Image Processing Primer

Prerequisites

The Image Processing Primer assumes knowledge of the MATLAB IDE, MATLAB help, arithmetic operations, built in functions, scripts, variables, arrays, logic expressions, conditional structures, iteration, functions, debugging, characters and strings, cell arrays, structures, and file input and output. Material on these topics is covered in the MATLAB Marina Introduction to MATLAB module, MATLAB Marina Variables module, MATLAB Marina Arrays module, MATLAB Marina Logic Expressions module, MATLAB Marina Conditional Structures module, MATLAB Marina Iteration module, MATLAB Marina Functions module, MATLAB Marina debugging module, MATLAB Marina Character and Strings module, MATLAB Marina Cell Arrays module, MATLAB Marina Structures module, and MATLAB Marina File Input and Output module.

Learning Objectives

1. Be able to use MATLAB to load, save, and display digital images.
2. Be able to extract portions of images using indexing.
3. Be able to resize images using down and up sampling.
4. Be able to implement image processing operations such as lightening and contrast enhancement.

Terms

Image, digital image, down sample, up sample, lighten (darken), contrast enhancement, spatial filtering

MATLAB Functions, Keywords, and Operators

imread, imwrite, uint8, conv2, colormap, axis equal

Images

An image is a two-dimensional light intensity function $f(x,y)$. Point (x,y) gives the spatial location and $f(x,y)$ gives the light intensity. The light intensity can be in terms of a single value in some range for gray scale images, a 0 or 1 for a black and white image, or some measure of color intensity (values of each of RGB). In general small values for the intensity correspond to dark areas and large values of intensity correspond to light areas. An image has infinitely many spatial points (x,y) and infinitely many light intensity values $f(x,y)$. To obtain a digital image from an image we must discretize (digitize) the space variables and discretize the intensities.

Discretizing the (x,y) values is called image sampling. This gives a finite grid of points $\{(x_1,y_1),(x_2,y_2),\cdots\}$. The information for each point is smeared over an area around it. The induced regions (each member of the grid) are called a picture elements or pixels. The intensity is assumed to be constant throughout a given pixel.

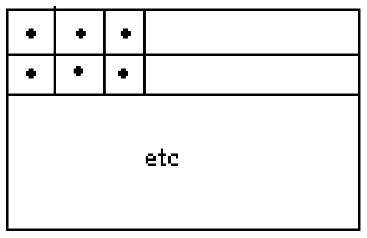

Figure 1a, Image Sampling

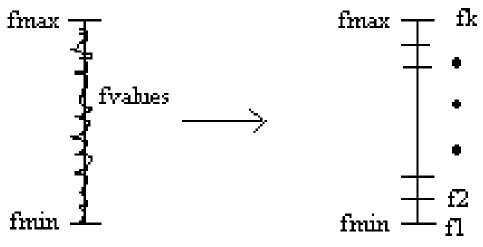

Figure 1b, Quantization

Discretizing the intensity values is called quantization. This means that we will only allow the intensity levels from a finite set of values $F = \{ f_1, f_2, \cdots, f_k \}$ (quantization means to allow only a finite number of values). If an intensity does not land on one of the f_i, then the closest f_i value to the actual value is used.

Higher quality pictures (pictures that are closer to the actual image) usually have a larger number of pixels and a larger number of intensity levels.

Digital Images

A color digital image (RGB) is an M by N by 3 array. A black and white or grayscale image is an M by N (M by N by 1) array. The first two subscripts are for specifying the pixel and the third subscript for color images is for selecting the red, green, or blue component of the pixel.

True color images store the intensity of the red, green, and blue (RGB) components of each pixel separately. True color images typically use either 8 or 16 bits to represent the intensity of each color. For example, bitmap color images (.bmp) use 8-bits for each color (24-bit total) giving an intensity range of 0 to 255 for each color, 0 being low intensity and 255 being high intensity. Color mapped images store a single value for each pixel (rather than 3 values). This value corresponds to a color from a stored set of colors. The colors are typically represented in 8 or 16-bits giving 256 or 65536 different colors.

Grayscale images are similar to color mapped images in that each pixel is a single value. The value is an intensity ranging from black to white. Typically the grayscale intensities are stored in 8-bits giving 256 different intensities, 0 being black, 255 being white, and in between are shades of gray. Black and white images represent each pixel with one of two values: black (0) or white (1). One can also represent black and white images using a color or gray scale representation. For color representations of black and white images RGB are all 0 for black and all 255 for white and for gray scale representations of black and white images black is 0 and white is 255.

Loading, Saving, and Displaying Images in MATLAB

The MATLAB functions `imread` and `imwrite` allow one to load and save most standard image formats. The `imread` function reads in a grayscale or color image from the specified file. The file format is specified by a string. If the format string is omitted, MATLAB will try and infer the file format. Some of the more common file formats that you may encounter are given in Table 1 along with their extensions.

File Type	Extension
Bitmap	.bmp
Graphics Interchange Format	.gif
Joint Photographic Experts Group	.jpg or .jpeg
Portable Network Graphics	.png
Tagged Image File Format	.tif or .tiff

Figure 2, Table of Common Image File Formats

Bitmap and jpeg images are the most commonly encountered images. Both file types can store gray scale or true color images and the images are typically read in as M by N by 3 arrays of 8-bit unsigned integers (`uint8`) although the pixel values may also be 16-bit unsigned integers (`uint16`) or real numbers (`double`). For the full list of bit depths supported for the file type see MATLAB's help on `imread`.

The `imwrite` function writes an image to a file in the format specified. For bitmap and jpeg images the image must be either grayscale (M by N) or true color (M by N by 3). The standard data types for writing to files are `uint8` numbers in the range 0 to 255 or real numbers in the range [0, 1]. The file format is specified by a string. If the format string is omitted, MATLAB infers the file format from the filename extension. For the full list of file types and options see MATLAB's help on `imwrite`.

The MATLAB function `image` displays an array as an image. Figure 3 shows a MATLAB code segment that reads in an image (jpeg), displays the image, and saves the image as a bitmap image.

```
im1 = imread('lucy.jpg', 'jpg');
figure(1), image(im1);
imwrite(im1,'lucy2.bmp','bmp');
```

Figure 3, MATLAB Code to Read, Write, and Display an Image

The image loaded from the file `lucy.jpg` is stored in the variable `im1`. The image data `im1` is an 1173 by 804 by 3 matrix of type `uint8`. For color mapped images, the color map is specified using the MATLAB `colormap` function before the image display operation. MATLAB has a color map editor for creating your own color maps.

Since the images are displayed in figures, the MATLAB functions `title`, `xlabel`, `ylabel`, etc. can be used to annotate images. Pay attention to the image dimensions when displaying images. Images are displayed in square figure windows by default but most images are not square so the image may be stretched in one or both dimensions. The MATLAB command `axis equal` sets the aspect ratio so that equal tick marks on the x and y axis are equal in size. This gives the correct aspect ratio in a displayed image. Whitespace is added to the smaller dimension to make the image plus whitespace square. The MATLAB command `axis image` is the same as `axis equal` except the plot box is tight around the image eliminating the whitespace.

Indexing and Slicing Images

Since images are 2D or 3D arrays, indexing operations can be performed on images to access individual pixels or portions of the image.

```
im1 = imread('lucy.jpg', 'jpg');
[M,N,P] = size(im1);
redPixel = im1(53,160,1);
greenPixel = im1(53,160,2);
bluePixel = im1(53,160,3);
upperHalfLucy = im1(1:1:586,:,:);
```

Figure 4, Indexing of Images

In the MATLAB code of Figure 4, `redPixel` is the red intensity of pixel (53, 160) in the `im1` image, `greenPixel` is the green intensity of pixel (53, 160) in the `im1` image, and `bluePixel` is the blue intensity of pixel (53, 160) in the `im1` image. The statement `upperHalfLucy = im1(1:1:586,:,:)` extracts the upper half of the image and saves it.

When using the size function to determine the dimensions of an image remember to give the appropriate number of return variables: two for grayscale images and three for true color images.

Up and Down Sampling Images

Down sampling an image decreases the number of pixels in the image. One can think of down sampling as either reducing the image size (fewer pixels but each representing same area as original pixels thus new image is smaller) or decreasing image resolution (fewer pixels representing image same size so each pixel represents a larger area than before). One potential problem with down-sampling is that aliasing might occur. To down sample by a factor of two, keep every other pixel. To down sample by some factor greater than one, keep pixels every factor apart. The down sampled image will have M/factor rows and N/factor columns. The MATLAB function of Figure 5 implements down sampling with vector operations.

Up sampling involves increasing the number of pixels in an image. The "new" pixels can either be copies of adjacent existing pixels or one can interpolate between pixels to get the values for the "new" pixels.

```
function outImage = downsample(inImage, factor)
% obtain the image size
[M,N,P] = size(inImage);
% down sample the image
outImage = inImage(1:factor:M,1:factor:N,:);
```

Figure 5, Down Sampling Function

Image Processing

Image processing involves applying some algorithm to an image generating a new image. Image processing is commonly used to improve images for human interpretation (clean up photos, x-rays), improve image for autonomous machine perception (automatic target recognition), or modify information in image for storage (compression/decompression). Image enhancement covers processes that improve the appearance of an image. These are often heuristic approaches that take advantage of the human visual system. Image restoration involves removing (undo) damage of an image. Restoration involves making assumptions on the type of damage down and fixing this by inverting the process. Noise removal for instance can use frequency select filters or averaging filters to try and eliminate the noise.

Typically processing of color images is performed on HSI images (hue, saturation, intensity). For HSI images, the intensity is decoupled from the color and processing the intensities does not alter the color. RGB images can be converted to HSI images, processed, and the processed image converted back to a RGB image for storage. We will focus only on RGB color, black and white, and grayscale images.

MATLAB operations and functions that are defined for 2D and 3D arrays can be used to operate on images. For example, subtraction of images can be used to detect changes between two still images and averaging several images of the same scene can be used to reduce noise in the images. However, many MATLAB operations and functions are not defined for unsigned integers (`uint8` and `uint16`). To account for this, the pixel values of images are often converted to real numbers for processing and then back to unsigned integers for display and storage.

The MATLAB arithmetic operations `+`, `-`, `.*`, ./, and `.^` are defined for unsigned integer data and `+`, `-`, `*`, /, and ^ are defined of arrays of unsigned integers. Operations done on data of type `uint8` that exceed the range of 0 to 255 will saturate on overflow (be set to either 0 or 255). MATLAB comparisons such as equal (==) and greater than or equal (>=) can be performed on unsigned integer data. Recall that the data type name is also a function that can be used for type conversion. For example, to convert an image from type `uint8` to type `double` could be done using the statement `im2 = double(im1)`. When converting data to type `uint8`, keep in mind that values less than 0 will be set to 0 and values greater than 255 will be set to 255.

In general to perform operations on an image one needs a double or triple nested loop. Two nested loops are needed to cycle though the rows and columns of the image allowing one to do an operation on each pixel. Sometimes another nested loop is needed to cycle through the RGB colors. MATLAB's array operations can often be used instead of one or more of the loops. The template for an operating on an image using iteration is shown in Figure 6.

```
function outimage = functionname(inimage, other parameters)
% allocate space for result
[M, N, P] = size(inimage);
outimage = zeros(M,N,P);

% image processing operation
for k1 = 1: 1 : M
    for k2 = 1 : 1 : N
     % instructions to process image

    end
end

end
```

Figure 6, Image Processing Function Template

Depending on the type of operation performed on the image, the image array may need to be converted from type uint8 to type double to do the work, and back to type uint8. Most MATLAB operations are not defined on data of type uint8.

Black and White and Gray Scale Images

Color images can be converted to gray scale images by replacing each RGB pixel value by the average intensity of the pixel colors. Gray scale or color images can be converted to black and white images using thresholding. Pixel intensities less than the threshold are converted to black and pixel intensities greater than the threshold are converted to white. The threshold is typically chosen to either be the middle value of intensity (128) or something lower or higher based on how dark or light the image is.

Figure 6a shows a color image and Figure 6b the grayscale image resulting from processing with the `colortogray` function. Figure 7a shows the `colortogray` function and Figure 7a shows a version using loops instead of array operations.

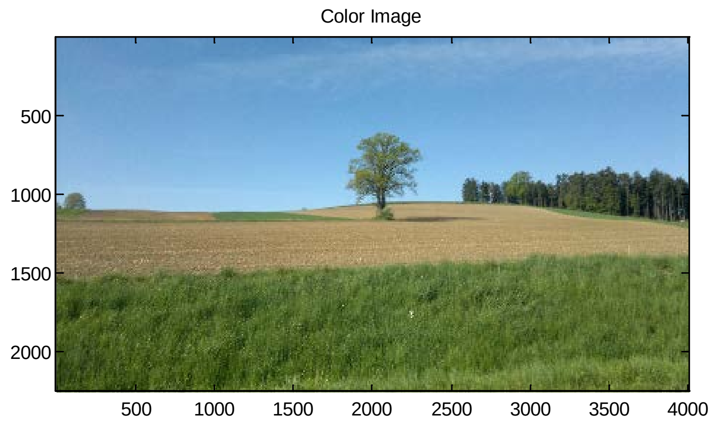

Figure 6a, Color (RGB) and Grayscale Image

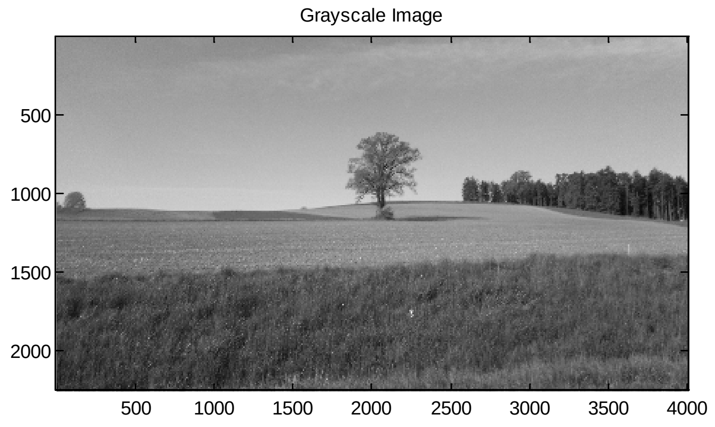

Figure 6b, Corresponding Grayscale Image

```
function outimage = colortogray(inimage)
% allocate space for the gray-scale image
[M, N, P] = size(inimage);
outimage = zeros(M,N,3);

% convert image to type double for operations
tempim = double(inimage);

% convert to gray scale by setting RGB values equal to
% average RGB intensity
outimage(:,:,1) =(tempim(:,:,1)+tempim(:,:,2)+tempim(:,:,3))/3.0;
outimage(:,:,2) = outimage(:,:,1);
outimage(:,:,3) = outimage(:,:,1);
% convert resulting image to type uint8
outimage = uint8(outimage);

end
```

Figure 7a, colortogray function

```
function outimage = colortograyLoop(inimage)
% allocate space for the gray-scale image
[M, N, P] = size(inimage);
outimage = zeros(M,N,3);

% convert image to type double for operations
tempim = double(inimage);

% convert to gray scale by setting RGB values equal to
% average RGB intensity
for k1 = 1: 1 : M
    for k2 = 1: 1 : N
        outimage(k1,k2,1) = (tempim(k1,k2,1) + tempim(k1,k2,2) +
tempim(k1,k2,3))/3.0;
        outimage(k1,k2,2) = outimage(k1,k2,1);
        outimage(k1,k2,3) = outimage(k1,k2,1);
    end
end

% convert image back to type uint8
outimage = uint8(outimage);

end
```

Figure 7b, colortograyLoop function

Lightening and Darkening
An image can be lightened or darkened by increasing or decreasing each pixel's intensity by a set amount or a percentage. When performing intensity transforms on color images one must be careful that the transform does not alter the color. One must also ensure that the resulting intensities do not go out of range of the legal pixel intensity values. To lighten RGB images, typically the intensity of each of the R, G, and B components of each pixel is increased or decreased by the same amount.

$f_{lightened}\left(x,y\right)=f\left(x,y\right)+\Delta$, where delta is the amount to lighten or darken
$f_{lightened}\left(x,y\right)=f\left(x,y\right)\cdot\left(1+\Delta\right)$, where delta is the percentage to lighten or darken

If delta is negative or a negative percentage, then the image is darkened.

To map the resulting intensity range $\left[r_1,r_2\right]$ after a transformation to the appropriate range $\left[s_1,s_2\right]$ (for `uint8`, 0 to 255)

$$f_{mapped}\left(x,y\right)=s_1+\left(\frac{s_2-s_1}{r_2-r_1}\right)\left(f_{transformed}\left(x,y\right)-r_1\right)$$

A simpler mapping is to set all values below the minimum of the range to the minimum and all values above the maximum of the range to the maximum, however this is a nonlinear mapping (saturation) and may not be desirable depending on the application.

Gray Level Slicing
Gray level slicing involves highlighting specific intensity levels and either eliminating or preserving the background. Gray level slicing is often used to pick out objects of interest from an image (works if the object is of a consistent intensity level). For example, to highlight the range of $\left[r_1,r_2\right]$ and eliminate everything else in an image $f\left(x,y\right)$

$$f_{graylevelsliced}\left(x,y\right)=\begin{cases}0 & if\ f\left(x,y\right)<r_1\\ s_1 & if\ r_1\le f\left(x,y\right)\le r_2\\ 0 & if\ f\left(x,y\right)>r_2\end{cases}$$

Or to highlight the range of $\left[r_1,r_2\right]$ and preserve the rest of the image $f\left(x,y\right)$

$$f_{graylevelsliced}\left(x,y\right)=\begin{cases}f\left(x,y\right) & if\ f\left(x,y\right)<r_1\\ s_1 & if\ r_1\le f\left(x,y\right)\le r_2\\ f\left(x,y\right) & if\ f\left(x,y\right)>r_2\end{cases}$$

Where s_1 would typically be much lighter or darker than the rest of the image so the highlighted portion stands out.

Contrast Enhancement
Images with poor contrast have their pixel intensities clustered in a limited range rather than spread through the entire range. Low contrast means that only a small proportion of the

intensities in the range are being used. The image can be light or dark or in between and have poor contrast. Image quality can often be improved by improving the contrast of the image.

Contrast stretching increases the range of intensity levels in the image. One method of contrast stretching is to scale the intensities so that they fall over the entire range or some new range. This can be done the same way as correcting intensities that are out of range after lightening/darkening. To map pixel intensities in the range $\left[r_1, r_2\right]$ to the range $\left[s_1, s_2\right]$ (for uint8, 0 to 255)

$$f_{contrast}\left(x,y\right) = s_1 + \left(\frac{s_2 - s_1}{r_2 - r_1}\right)\left(f\left(x,y\right) - r_1\right)$$

This is a linear transformation for contrast enhancement but the transformation can also be piecewise linear or nonlinear

Log transforms and power transforms are also used for contrast enhancement and are also are commonly used to correct intensity discrepancies between image capture and display devices (monitors have intensity responses that are power functions of the input). When used to correct intensity discrepancies in display devices, the contrast enhancement using a power transform is called gamma correction.

$f_{contrast}\left(x,y\right) = c \cdot \left(f\left(x,y\right)\right)^{\gamma}$, where c and γ are constants

When $\gamma < 1$, the lower (dark) values are expanded (increases detail in dark areas) and when $\gamma > 1$ the higher (lighter) values are expanded (increases detail in bright areas, eliminate washed out appearance). Choosing gamma too low can give a washed out appearance (all light) and choosing gamma too high can give the image too dark of a cast. The constant c is chosen to keep the transformed intensities within the appropriate range, or 8bit intensities, $c = 255/\left(255^{\gamma}\right)$.

Histograms

The histogram $h\left(r_k\right)$ of a digital image is a discrete function that gives the frequency of occurrence of the intensity levels in an image.

$$h\left(r_k\right) = n_k$$

where r_k is the kth intensity level in the image and n_k is the number of pixels having the intensity of r_k in the image

The histogram is often normalized by the total number of pixels yielding the probability of occurrence of the intensity level r_k

$$p\left(r_k\right) = \tfrac{n_k}{M \cdot N}$$

M and N are the number of rows and columns in the image and M times N is the total number of pixels in the image

Histogram plots (typically bar plots) show either the frequency or probability of occurrence of each intensity level.

Images with low contrast will have histograms that are narrow (only cover small range) whereas images with high contrast have histograms that are evenly spread throughout the intensity range (uniformly distributed). A transformation function that yields an evenly distributed histogram will give good contrast enhancement.

Histogram Equalization
Given a digital image with pixels in the intensity range $[0, L\text{-}1]$, histogram equalization will spread out the intensity levels in the digital image resulting in a transformed image with a more uniform histogram. Histogram equalization generally improves the contrast of an image.

Assuming that the intensity level r_k occurs with probability $p(r_k) = \frac{n_k}{M \cdot N}$ then the transformation for histogram equalization is given by

$$s_k = (L-1)\sum_{\ell=0}^{k} p(r_\ell)$$

Where r_k is the intensity in the range $[0, L\text{-}1]$ in the original image and s_k is the intensity in the transformed image.

For a true color (RGB) image, histogram equalization could be done on each component of the color image, i.e. on each of the three color images R, G, and B. This will improve the contrast but will also change the colors of the image. The resulting color image may look quite strange.

Histogram equalization transforms are invertible if the original histogram probabilities are available.

Spatial Filtering
For spatial filtering, a filter mask is moved point to point over the digital image and the resulting pixel value for the filtered image is calculated based on the mask coefficients and the values of the pixel and its neighbors covered by the mask. If the filtering is linear, the then result is the sum of the products of the mask coefficients and the image pixels in the area under the mask. Spatial filtering is similar to convolution, thus the masks are sometimes called convolution masks or convolution kernel.

For the 3 by 3 linear filter shown in Figure 8, the result of filtering with the mask centered on pixel (x, y) of f is

$$\begin{aligned} f_{filtered}(x,y) &= w(\text{-}1,\text{-}1)\cdot f(x\text{-}1,y\text{-}1) + w(\text{-}1,0)\cdot f(x\text{-}1,y) + w(\text{-}1,1)\cdot f(x\text{-}1,y+1) \\ &+ w(0,\text{-}1)\cdot f(x,y\text{-}1) + w(0,0)\cdot f(x,y) + w(0,1)\cdot f(x,y+1) \\ &+ w(1,\text{-}1)\cdot f(x+1,y\text{-}1) + w(1,0)\cdot f(x+1,y) + w(1,1)\cdot f(x+1,y+1) \end{aligned}$$

In general, the linear filtering of an M by N image f(x,y) by an m by n mask is given by

$$f_{filtered}(x,y)=\sum_{i=-a}^{a}\sum_{j=-b}^{b}w(i,j)\cdot f(x+i,y+j)$$

where $a=(m\text{-}1)/2$ and $b=(n\text{-}1)/2$ (assuming m and n are odd) and this must be done for all pixels (x,y) of the image f(x,y).

$w(-1,-1)$	$w(-1,0)$	$w(-1,1)$
$w(0,-1)$	$w(0,0)$	$w(0,1)$
$w(1,-1)$	$w(1,0)$	$w(1,1)$

Figure 8, 3 by 3 spatial mask

Blurring an image makes it fuzzier. Sharpening an image makes it clearer. However, sharpening an already sharp image can degrade the visual quality of the image.

Smoothing filters are used for blurring and noise reduction. A smoothing filter averages the pixels under the mask. Smoothing filters reduce sharp transitions in intensity levels of an image (noise often shows up as sharp transitions in intensity levels). However, edges in an image also correspond to sharp transitions in intensity levels, so removing noise can have the effect of blurring the edges in an image. Figures 9a and 9b show two examples of 3 by 3 smoothing filters. Larger masks correspond to higher order filters. In general a higher order averaging filter will blur an image more than a lower order averaging filter.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Figure 9a, 3 by 3 Smoothing Filter

1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

Figure 9b, 3 by 3 Smoothing Filter

Sharpening filters enhance detail that has been blurred or highlight fine detail in an image. Sharpening can be accomplished using spatial differentiation (spatial differences). Differentiation detects changes, and the response of the differential operator is proportional to the degree of change. Thus spatial differentiators can be used to enhance edges and other discontinuities in an image. Second order differentiation functions are commonly used for sharpening. They do a better job of enhancing fine detail like small points and thin lines than first order differentiation functions. First order differentiation functions are commonly used for edge detection or used in combination with second order differentiation functions for image enhancement. Masks corresponding to first and second order differentiation filters are shown in Figures 10a through 10d.

The filters of Figure 10a and 10b are Sobel operators and are commonly used for edge detection. The Sobel masks are typically used together, one to perform the gradient in vertical direction and the other in the horizontal direction. The masks for Figures 10c and 10d are Laplacian operators. Laplacian operators are commonly used for detail enhancement (it is a second order differentiation operator). Two other Laplacian masks can be obtained by negating the masks of Figures 10c and 10d. Since the Sobel and Laplacian masks have some negative coefficients it is possible to get negative intensities as a result (these should be shifted to within the legal range).

-1	-2	-1
0	0	0
1	2	1

Figure 10a, 3 by 3 Sobel Mask (vertical direction)

-1	0	1
-2	0	2
-1	0	1

Figure 10b, 3 by 3 Sobel Mask (horizontal direction)

0	1	0
1	-4	1
0	1	0

Figure 10c, 3 by 3 Laplacian Mask

1	1	1
1	-8	1
1	1	1

Figure 10d, 3 by 3 Laplacian Mask extended to diagonal neighbors

Last modified Friday, November 01, 2013