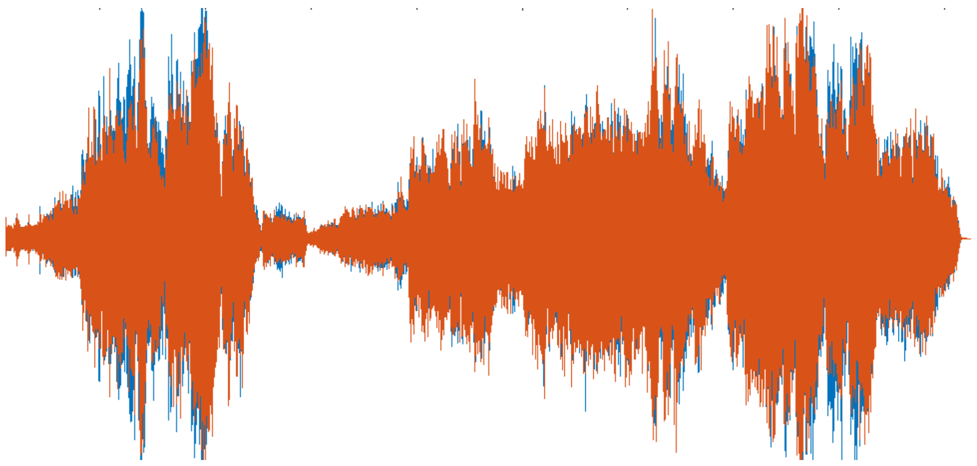
Digital Signal Processing using MATLAB



Electrical and Computer Engineering 5

Exploring Digital Signal Processing

Developed by Joshua Jain and Professor Truong Nguyen

Professor John Eldon | Professor Vikash Gilja | Professor Drew Hall | Professor Truong Nguyen

What You Will Need

Materials:

- Provided Image and Sound files.

Equipment:

- Desktop computer / Laptop

Software Tools:

- MATLAB suite with valid license

Part 2: Image Processing

The second portion of this lab will be on Image Processing. Image processing may sound like a difficult topic, however many of you may have already done some basic image processing without even realizing it. For those of you who are familiar with applications such as Instagram may have seen how dramatically an image’s appearance can be altered using such applications. With the simple press of a button, you are able to greatly enhance the aesthetics of your photos.

This is one of the goals what image processing attempts to accomplish. While Instagram filters are used to add artistic effects to an image, tools such as Photoshop can perform manipulations to images to remove defects and improve the quality of the image. All of this can fall under the general category of *Image Enhancement*, and while there are other fields of image processing such as Feature Recognition and Computer Graphics, we will focus on improving the quality of images by enhancement.

We will explore how images are constructed from *pixels*, and we will see how we can rearrange the *pixel values* using an *image histogram*. By the end of this lab, you will understand exactly what each of these italicized terms mean.

Challenge #1: Pixels and Matrices

As mentioned in lecture, the raw voltage output from a CCD can be digitized and transformed into a digital signal by sampling it. The numeric values of that signal are interpreted as the intensity of light striking various pixels of the image. In order to keep track of all of these values, the computer usually stores your image signal in a structure called a matrix, which is just a collection of numbers arranged in columns and rows. For an image, each numeric value represents a *pixel* and the position, which row and column within the matrix, of that numeric value corresponds to the position of the pixel in the overall image.

The following examples are meant to get you accustomed to and familiar with working with images and matrices in MATLAB. Please run each of the following code snippets on your computers.

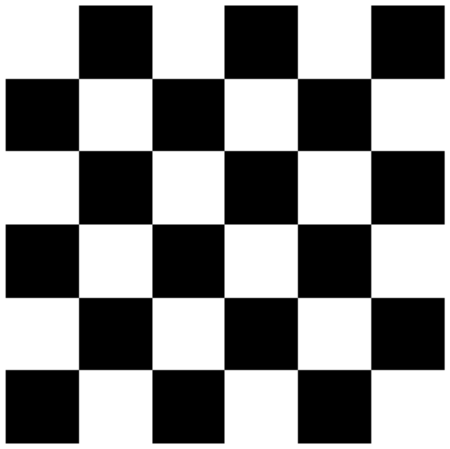
To better understand the way pixels work, we can use MATLAB to develop an 8-bit greyscale image. To do this we can define a matrix filled with numeric values within the range of 0 and 255. 0 corresponds to black and 255 corresponds to white and the pixel definition varies linearly between white and black between 0 and 255, hence the image being greyscale.

```
% 2 dimensional (3x3) matrix A

A =   [0 130 255;
       0 130 255;
       0 130 255;]
```

With matrix A defined in our MATLAB workspace we can now use the *imshow()* function to display / plot the matrix as an image. (Note: (1) the function *uint8()* is used to ensure the variable type is recognizable by the function which requires an unsigned 8 bit integer to display this image correctly (2) *truesize()* and *set()* functions and their inputs will resize pixels to appear larger and enlarge the figure window respectively. See following example.)

```
imshow(uint8(A))
truesize(gcf,[100 100]) % This function is only so pixels appear larger
set (gcf, 'Units', 'normalized', 'Position', [0.1,0.1,0.7,0.7]);
```

Exercise 1a: After understanding the development of a 2-dimensional matrix and functions used to display the defined image, develop a 6x6 checkerboard image using MATLAB as shown here:



More definition can be added to your image by defining a matrix with more rows and columns. Below is a 12x11 matrix. The code illustrates one way to develop matrix A and, then, an equivalent but easier way. MATLAB makes developing and manipulating matrices quick and easy. For example:

```
A = [1     1     1     1     1     1     1     1     1     1     1;
     1     1     1     1     1     1     1     1     1     1     1;
     1     1     1     1     1     1     1     1     1     1     1;
     1     1     1     1     1     1     1     1     1     1     1;
     0    25    50    75   100   125   150   175   200   225   250;
     0    25    50    75   100   125   150   175   200   225   250;
     0    25    50    75   100   125   150   175   200   225   250;
     0    25    50    75   100   125   150   175   200   225   250;
   255   255   255   255   255   255   255   255   255   255   255;
   255   255   255   255   255   255   255   255   255   255   255;
   255   255   255   255   255   255   255   255   255   255   255;
   255   255   255   255   255   255   255   255   255   255   255;  ]
```

is equivalent to:

```
A =   [1*ones(4,11);
      0:25:250;
      0:25:250;
      0:25:250;
      0:25:250;
      255*ones(4,11);]
```

The colon operator ":" and ones() function are used here instead of typing out individual elements of the matrix, this can be easily scaled for your image to include thousands of pixels if desired. And to display the image defined with matrix A we can input the following code:

```
A = uint8(A);
figure(1)
imshow(A)
truesize(gcf,[100 100])
set (gcf, 'Units', 'normalized', 'Position', [0.1,0.1,0.4,0.7]);
title('Image from Matrix A')
```

A simple "transpose" operation may be performed on the matrix by using the " ' " transpose operator. By running the following lines of code you can visually understand how a transpose alters an image in a new figure.

```
figure(2)
imshow(A') % Transpose A
truesize(gcf,[100 100])
set (gcf, 'Units', 'normalized', 'Position', [0.5,0.1,0.4,0.7]);
title('Image from Matrix A'' (transposed)')
```

So far, we have been working with greyscale images, where pixel intensity values from 0 to 255 represent black, white, and various shades of grey. How about color images? How are they different from greyscale images? First off, color will add depth to your image as well as to the matrix producing it. **By adding color, there will be 3 dimensions to your matrix** (MxNxD). The additional 3rd dimension (denoted by D), added to the 2-dimensional case seen prior as greyscale, accounts for the RGB layers (Red, Green, Blue) of the color image. Each pixel now has 3 components, so, we will have 3 matrices to describe the R and G and B pieces of the colored image. Here is an example where you will be able to recognize many similarities to the previous greyscale code:

```
% Color: RGB 3-Dimensional Matrix
  % Overlay 3 Matricies
  clear all; clc; close all;

  A_R = [ 255 0 0;
          255 0 0;
          255 0 0;]
  A_G = [0 255 0;
         0 255 0;
         0 255 0;]
  A_B = [0 0 255;
         0 0 255;
         0 0 255;]

     A = zeros(3,3,3);
     A(:,:,1) = A_R;
     A(:,:,2) = A_G;
     A(:,:,3) = A_B;

  imshow(uint8(A))
  truesize(gcf,[100 100])
  set (gcf, 'Units', 'normalized', 'Position', [0.1,0.1,0.7,0.7]);
```

Here is an example of a larger image where colors overlay and are seen in an additive sense:

```
% https://en.wikipedia.org/wiki/Additive_color
% Overlay 3 Matrices
clear all; clc; close all;

   A_R = [ 255*ones(6,4) zeros(6,1) zeros(6,1)];
   A_G = [ zeros(6,1) 255*ones(6,4) zeros(6,1)];
   A_B = [  zeros(6,1) zeros(6,1) 255*ones(6,4)];

      A = zeros(6,6,3);
      A(:,:,1) = A_R;
      A(:,:,2) = A_G;
      A(:,:,3) = A_B;

   imshow(uint8(A))
   truesize(gcf,[100 100])
   set (gcf, 'Units', 'normalized', 'Position', [0.1,0.1,0.7,0.7]);
```

Exercise 1b: After understanding the development of a 3-dimensional matrix to produce an RGB image and functions used to display the defined image, create a color pattern of your choice with at least a 6 x 6 x 3 matrix.

Loading Images into MATLAB

The previous section was to get you familiar with using MATLAB to create and manipulate your own images. You should now understand that images are stored as matrices in MATLAB. Throughout this lab, we will be making use of the prebuilt functions in MATLAB's Image Processing Toolbox. If you decide to take an upper division course in Image Processing/Computer Vision later in your studies, you will learn about the algorithmic implementation of many of these functions. For now, we will just use just use the available functions. Provided is a link to function descriptions for reference:

http://www.mathworks.com/help/images/functionlist.html

Now let's take an existing image and see how we can manipulate it. Before we can do this, we must first load the image into MATLAB. In other words, the image must be placed in the MATLAB "workspace," a special area where MATLAB stores data for use in functions or scripts. There are a number of ways you can do this. Simply dragging the image into your workspace will work is one option. However, we are going to do this programmatically. Loading an image into MATLAB can be performed using a single function.
Namely, use the following function to load your image downloaded from *Canvas* ('test.jpg'):

```
IMG = imread('test.jpg');
```

Note that in order to use this method, your image must be in your Current Folder in order for MATLAB to recognize and load it into your Workspace. Here is an alternative method to load an image from anywhere on your computer.

```
IMG = imread('path');
```

Here you must specify the path to the image in the parenthesis of the *imread()* function. Either method should produce the same result, a variable called IMG which stores your image.

Optional Exercise: Do a google images search and download a color JPG format image. Use the above methods to place your image into the MATLAB Workspace. Execute the *imread()* command and store your image as a variable called "IMG".

Notice that after you execute this command, a variable called IMG will be created. Notice that this variable is actually a matrix of numbers, as explained in the first section of this lab. Take a second to recall the significance of this. MATLAB and pretty much all digital image processing software tools store images as just huge matrices of numbers. From mathematics, we can perform mathematical operations on matrices with ease. Many of the effects we will explore when we get to the section on filtering involve simple mathematical operations on matrices. Keep this in mind as you progress through the lab.

Now there are a few questions you should ask yourself at this point. From our previous discussion the answers should hopefully be clear by now. One, why is the image sized the way it is, why does it have three dimensions. Two, what do the values of the numbers mean? Before proceeding to the next exercise, be sure to spend some time thinking about these questions.

Challenge #2: Exploring the Image Structure

Notice that your dot JPG image is stored as a MxNx3 dimensional matrix (M denotes the number of rows, N the number of columns, and 3 is the number of *layers*). You know that you can represent any color as a combination of Red, Green, and Blue. These layers specify the exact content of each of these RGB building blocks per pixel. To get a better understanding of this, you should break the image up into components and display each layer as a separate image.

If I have an image matrix which is composed of three RGB layers, to get the R layer, I would have to tell MATLAB to slice off the other layers. To do this, if I have an image called "IMG", I would call the following command:

```
R = IMG(:,:,1);
```

Remember that each layer of the image is MxN, so the colons tell MATLAB to preserve all the pixel elements in the rows and columns. The number 1 in the third position tells MATLAB to access the 1st layer, the layer which corresponds to R (red).

Exercise 2a: Separate your image into RGB layers. Execute the previously mentioned command for each layer (R, G, and B, or the 1st, 2nd, and 3rd layers) of the image. Save your results in variables R, G, and B for the next exercise.

In order to display these layers individually, try calling imshow() on your variables for R, G, and B individually. When you call imshow() on the red layer, one would expect to see an image comprised entirely of red pixels? Unfortunately, due to the way MATLAB interprets an RGB image, a bit more work is required.

Look at the dimensions on each of these images, you will notice that they are only MxN and they are missing the third dimension. This is because you have in effect taken the single 3-layered image and have split it up into three single layered images. MATLAB will display Color RGB images if the images have three layers, otherwise it will display the image as greyscale. In order to get around this issue, for each image layer (R,G, and B), we will recreate the other two layers. We will use all zeroes for the pixel values of these reconstructed layers, this will in effect only show the RGB layer of interest when we call imshow() (an RGB image with a nonzero Red layer and zero Blue and Green layers). Here is the code which will accomplish all of the above.

```
a = zeros(size(IMG, 1), size(IMG, 2));

just_red = cat(3, R, a, a);
just_green = cat(3, a, G, a);
just_blue = cat(3, a, a, B);
```

Exercise 2b: Look at the original image; can you guess which of the RGB layers seems to be of most importance or is most prevalent? (Does the image have a redder tone to it? You could say that the Red layer is most prevalent). Display each of the RGB layers using imshow().

After you have identified which layer you think is of most importance, let’s see what happens to the image if we alter that particular layer.

Exercise 2c: Scale this layer of your image by some factor which is less than 1. For example, using the following line of code, I can scale down the red layer of my image by a scale factor of 0.2, and then add it back to my original image:

```
new_R = (0.2 .* R);
new_image = cat(3, new_R, G, B);
```

Exercise 2d: Display this new image and compare it to the original image, would you say your image looks better or worse? Comment.

Exercise 2e: Teeth Whitening.

Whitening toothpaste often achieves its goal by taking advantage of a phenomenon known as an optical shift. Basically, a blue pigment contained in the toothpaste is applied to stained teeth during regular brushing. A thin layer of this pigment is deposited which alters the optical qualities (how the light reflected off of the surface of the tooth is perceived by the brain) of the stained tooth and makes the tooth appear visibly whiter. In this exercise, we will simulate the effects of whitening toothpaste on some yellowed teeth using MATLAB and the concepts you have learned so far. Complete the following steps using the provided image, ‘stained_teeth.jpg’.

Step 1) Import the image “stained_teeth.jpg” into your workspace.

Step 2) Separate the blue layer from this image. Amplify this layer by scaling it up by some number *greater* than 1.

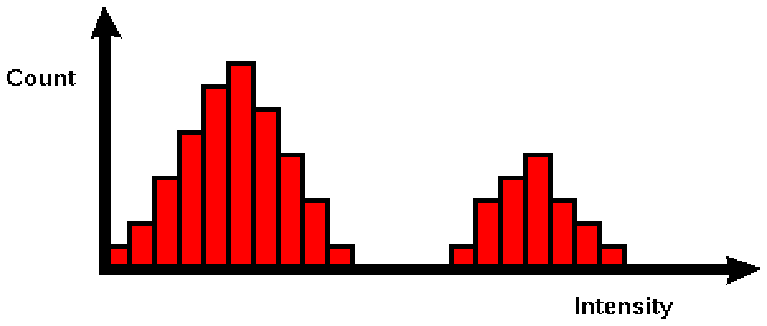
Step 3) Take your new blue layer, and combine it with the original red and green layers of the “stained_teeth” image. (Hint: You can make use of the *cat()* function to do this.)

Step 4) Show the new image next to the old image using *subplot().* Do the teeth appear whiter? If there is no noticeable effect, try increasing your scale factor.

Step 5) Try different scaling factors for the blue layer, display two new images with differently scaled blue layers.

Challenge #3: Image Histograms

The histogram is an indispensable tool to any engineer working in image processing. By just looking at a histogram, we can get an idea of how the overall image will look, and what adjustments should be made to improve its quality. A histogram is simply a graph which plots pixel intensity values (X axis) and their frequency of occurrence (Y axis). Below is an example of an image intensity histogram:



Source: http://homepages.inf.ed.ac.uk/rbf/HIPR2/histgram.htm

Looking at this particular histogram, how do we interpret it? Well the easiest way is to keep in mind that the horizontal X axis represents pixel intensity values, and that vertical Y axis represents how often these intensity values occur in the image. Knowing that low pixel intensity values represent darker shades, and that high pixel intensity values represent brighter shades, we can infer that the image this histogram represents would have a large distribution of dark pixels and a few very bright pixels, but there would be few midrange gray pixels.

Here is a real example of an image and its associated histogram:

Image with very low contrast, Source: https://visart1.wordpress.com/

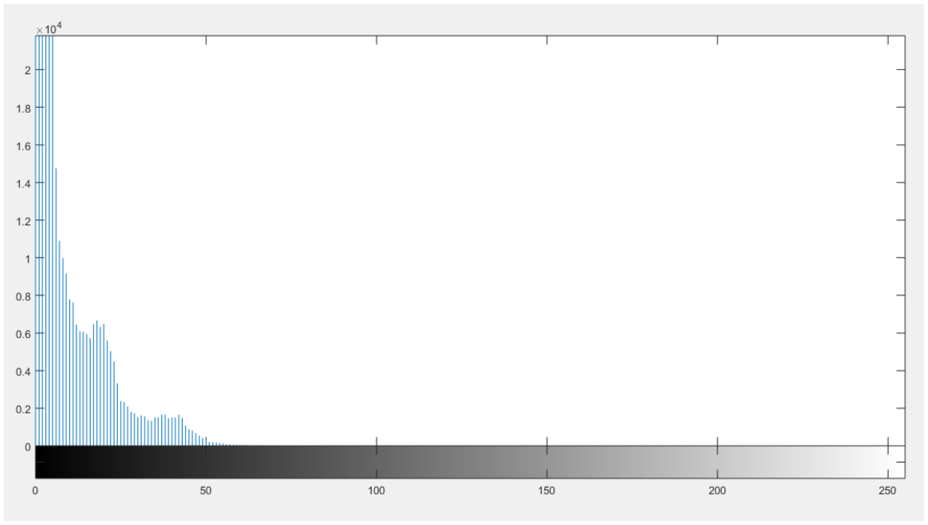
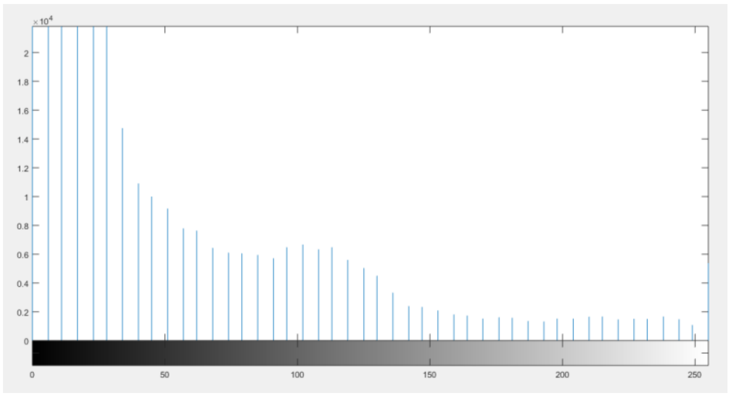
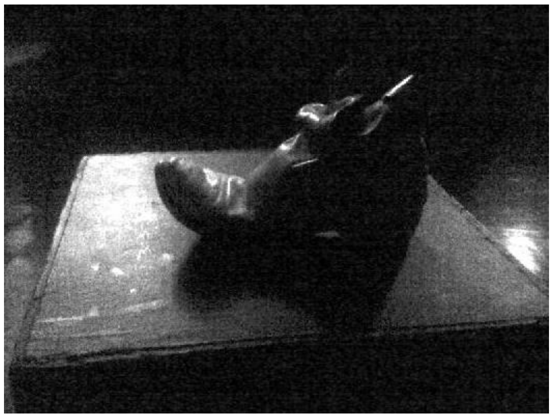


Image histogram showing most pixel intensity values are concentrated at the darker tones/lower intensity values. This explains the darkness of the image.

The histogram shows that most, if not all of the pixels have low intensity values. Because of this, the image has very low contrast and would not be very useful. Thus, methods for improving the contrast of these images are of interest to engineers and photographers. Two simple methods which we will explore here are called *Histogram Equalization* and *Contrast Stretching*.

Contrast stretching attempts to spread the pixel values along the entire intensity range (1-255). Applying contrast stretching to the previous image, we obtain a much better spread over the intensity range as seen in the following figures:





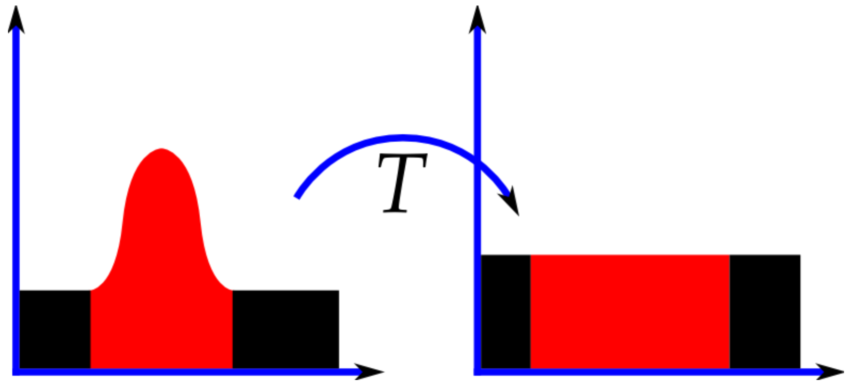
Lightened "Boot" and its associated Histogram.

Contrast Stretching can be achieved by using the *imadjust()* and *stretchlim()* functions (refer to online documentation) and a single line of code.

```
C = imadjust(IMG,stretchlim(IMG),[]);
```

The *imadjust()* function takes 3 arguments. The first is an input image "IMG", and it redistributes the pixel intensity values according to the 2nd and 3rd arguments provided (use the *stretchlim()* function in the 2nd argument, and keep the 3rd argument as default by using empty square brackets).

Another method for improving image contrast is histogram equalization. This method attempts to flatten the distribution of pixel intensity values along the entire intensity range. Ideally, histogram equalization applied to an image would transform the images histogram in the manner of the following image:



Source: https://en.wikipedia.org/wiki/Histogram_equalization

The histogram on the left is flattened to the histogram on the right. This is how histogram equalization attempts to improve the overall contrast of the image.

Plotting an image histogram in MATLAB is a single function call, the following line of code will plot the histogram for an image IMG:

```
imhist(IMG);
```

Exercise 3a: Use histogram equalization on the provided image (dark.jpg) to improve its contrast. To do this, make use of the MATLAB *histeq()* function. Plot the original image and new image histograms.

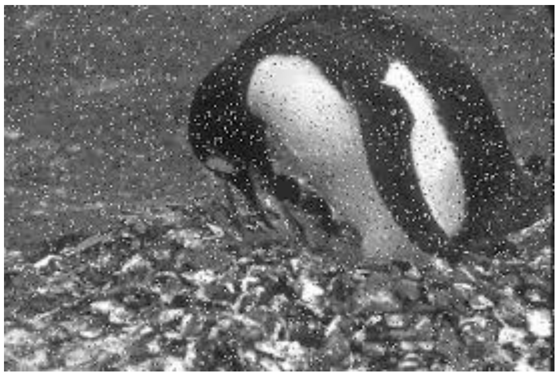
Exercise 3b: Perform Contrast Stretching on the same image (see previous example). Plot the new image histogram.

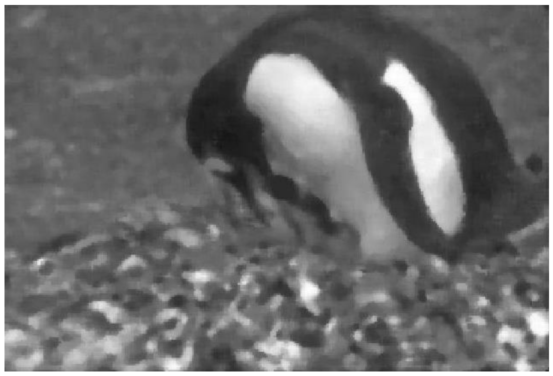
Exercise 3c: Compare the original image, the image after Contrast Stretching, and the image after Histogram Equalization. Which looks better?

Challenge #4: Image Filtering/Manipulation using Kernel Matrices

By now, you know that images are just large matrices of numbers. You also know how to do basic manipulations to those matrices to achieve some desired effect. You have also covered a fundamental tool for image analysis, the intensity histogram. We are now in a position to start exploring more advanced effects such as image filtering. Understand that the word "filtering" is used in the common sense, in that a filter is just something which removes undesirable components and lets the good components pass. In this section, we will start off with image filtering to remove undesirable effects, formally known as "noise."

Noise is just defined as any undesirable visual component of the image. Images can become corrupted by noise in a wide variety of ways; typically it is a result of the sensors we use to capture the image. Since noisy images are not very useful, Engineers and Photographers are interested in developing techniques to remove such undesirable noise from images. For example, take a look at the following images:





The image on the left has been corrupted by "salt and pepper" noise (unwanted black and white pixels, visually similar to common table salt and pepper). The image on the right is the same image with the noise removed. The most common way to remove this particular form of noise is by using what is called a *median filter*. This filter belongs to a broader class of filters known as Kernel Filters, which we shall explore here.

Kernel Filters are basically small matrices which when applied to a larger matrix (such as the matrix representing your image), produce some useful effect. In formal terms, you may hear engineers speak of the *convolution* of a kernel filter with a matrix. Convolution is just a mathematical operation which explains what happens when the image is filtered; it is an intermediate topic which will be treated in detail in your further courses.

Exercise 4a: Navigate to this link which contains a very useful visualization of what a Kernel Filter is and how it affects an image: http://setosa.io/ev/image-kernels/

Spend some time applying different kernel filters to the provided example image on the site. Pick your favorite Kernel Filter and save a copy of the resulting image after you apply that filter. Be sure to label the image with the name of the filter you used.

We will now use functions in MATLAB to achieve these same effects.

Exercise 4b: Noise removal using kernel filters.

Step 1) Import provided image Penguin.jpg into your workspace.

Step 2) Plot the image and its histogram. The image contains two forms of noise, White Gaussian Noise and Salt and Pepper Noise.

Step 3) Remove the Salt and Pepper noise using a median filter (*medfilt2* function). Plot the new image and its histogram. Compare it to the original histogram.

Step 4) In order to remove the Gaussian White Noise, we will use a mean averaging filter. In order to do this, you will need to use the *fspecial*() function to create your Kernel Matrix, and the *imfilter()* function to apply it to your image. See the following link for examples on how to do this:

http://www.mathworks.com/help/images/ref/fspecial.html (scroll down to input arguments)

Step 5) Plot the image with all noise removed and its histogram. Again, compare this histogram to the original.

Optional

In theory, one could create their own kernel matrix and apply it to an image of their choice. Typically, one uses a prebuilt Kernel to perform basic manipulations, but for those interested, here is the skeleton of the code to build and apply a custom Kernel “h” to any image “x”.

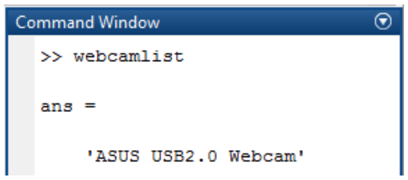
```
%Image manipulation using Kernels
%Writing your own kernels

close all

h = [-2 -1 0; -1 1 1; 0 1 2];       %My Kernel Matrix
x = rgb2gray(imread('Barbara.jpg')); %My image
y1 = (filter2(h,x));                %Applying my Kernel Matrix

figure; subplot(1,2,1)
imshow(x, [])
title('Original Image')
subplot(1,2,2)
imshow(y1, [])
title('Altered Image')
set(gcf,'Position', get(0,'Screensize'));
```

Challenge #5: Webcam → Photostrip

Developed by Camille Lee

1. Download the MATLAB USB Webcams Support Package if not installed already.
   a. Home → Add-Ons → Get Hardware Support Packages → Install from Internet → Search for "USB Webcams" → Next → Accept terms/conditions and install

2. In the command window type, "webcamlist". It will reply back with the name(s) of the webcams installed or attached.



```
>> webcamlist

ans =

    'ASUS USB2.0 Webcam'
```

3. Set the webcam to be an object.

```
>> cam = webcam(1)
```

4. Test to see that your webcam is on and working by using the *preview()* function

```
>> preview(cam)
```

   Up top on the preview window you can see the camera name. On the bottom part of the window you can see the camera name, resolution, frame rate, and the timestamp (in seconds).

   You can close the preview with the function *closePreview().*

5. Take a picture!
   To take an image with your webcam use the function *snapshot().* Use the function *imshow()* to display your image.

```
>> img = snapshot(cam);
>> imshow(img)
```

6. To save your image use the function *imwrite().*

```
>> imwrite(img, 'myfirstimage.jpg');
```

7. Close the webcam using the function *delete().*

8. Photobooth Challenge!
   As in the prior challenges to Lab 3, you can now take that image and do some signal processing! Take the image from your webcam and create your very own photobooth reel! Use your knowledge from lab 3 to change each photo in the reel showing different types of effects. For instance, Photo 1 can be black and white, Photo 2 can be the original image but flipped, Photo 3 create your own!

Check out this link for some interesting signal processing ideas:
http://blogs.mathworks.com/steve/category/special-effects/

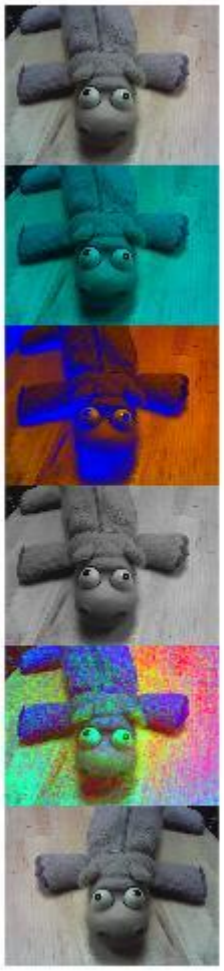
Here is example code you may use to better understand how each of the functions above work. Since each step is separated by "%%" you should "run section" one section at a time instead of running the entire code at once.

```
%% Photobooth Challenge for ECE 5
% The purpose of this program is to take a photo from a webcam and do some
% signal processing with the image. We will be manipulating the pixels by
% various means such as changing the color or flipping the image.

% Resources - Much of the code can be found on the previous part of Lab 3
% or http://blogs.mathworks.com/steve/category/special-effects/

%% Clears figure windows, variables, commands, etc.
clear all; close all; clc;

%% Shows what webcams are connected to the computer
webcamlist

%% Creates webcam as an object and tests to see that webcam is working.
cam = webcam(1); % input maybe 2 or 3 if using attached webcam
preview(cam);

%% Take an image with webcam and saves as JPG
image = snapshot(cam);
imwrite(image, 'imagetesting.jpg');
imshow('imagetesting.jpg');     % displays the image

%% Closing windows
closePreview(cam);      % closes the preview window
delete(cam);            % closes the webcam
```

Cyan

```
% Creates different filter colors by changing the R, G, or B values
 R = img(:,:,1);
 G = img(:,:,2);
 B = img(:,:,3);

 z = zeros(size(img, 1), size(img, 2));

 cyan = cat(3, z, G, B);
```

Original



Color

```
% Contrast Enhancement of the image
 SE = strel('Disk', 18);
 imgEnhanced = imadjust(img,[0.20 0.00 0.09; 0.83 1.00 0.52],...
     [0.00 0.00 1.00; 1.00 1.00 0.00], [1.10 2.70 1.00]);
```

Black and White

```
% Creating a black and white image
 x = rgb2gray(imread(fileName));
 black_and_white = uint8(zeros(480, 640, 3));
 black_and_white(:,:,1) = x;
 black_and_white(:,:,2) = x;
 black_and_white(:,:,3) = x;
```

Flipped Image

```
% Flipping an image
a = fliplr(img(:,:,1));
b = fliplr(img(:,:,2));
c = fliplr(img(:,:,3));

flippedImage = cat(3,a,b,c);
```

Decorrelation

```
% Decorrelation streching - useful for visulaizing image by reducing
% inter-plane autocorelation levels in an image.
 colorImg = decorrstretch(img);
 colorImg = imadjust(colorImg,[0.10; 0.79],[0.00; 1.00], 1.10);
```

Other Filter

```
just_red = cat(3, R, z, z);
yellow = cat(3, R, G, z);
just_green = cat(3, z, G,a);
just_blue = cat(3, z, z, B);
purple = cat(3, z, a, z);
```

Other Saturation

```
imgEnhanced1 = imadjust(img,[0.00 0.00 0.00; 1.00 0.38 0.40],...
    [1.00 0.00 0.70; 0.20 1.00 0.40], [4.90 4.00 1.70]);
imgEnhanced2 = imadjust(img,[0.13 0.00 0.30; 0.75 1.00 1.00],...
    [0.00 1.00 0.50; 1.00 0.00 0.27], [5.90 0.80 4.10]);
imgEnhanced3 = imadjust(img,[0.20 0.00 0.00; 0.70 1.00 1.00],...
    [1.00 0.90 0.00; 0.00 0.90 1.00], [1.30 1.00 1.00]);
```

```
%% Altering the photos and plotting as photoreel
fileName = 'myfirstimage.jpg';       % Replace with the name of your image
pic = imread(fileName);
img = imresize(pic, [480 640]);         % Resizes the image

% Creates different filter colors by changing the R, G, or B values
 R = img(:,:,1);
 G = img(:,:,2);
 B = img(:,:,3);

 z = zeros(size(img, 1), size(img, 2));

 cyan = cat(3, z, G, B);

% Creating a black and white image
 x = rgb2gray(img);
 black_and_white = uint8(zeros(480, 640, 3));
% uint8() fixes the matrix size so it can go in photoreel
 black_and_white(:,:,1) = x;
% stores the black and white image into different layers
 black_and_white(:,:,2) = x;
 black_and_white(:,:,3) = x;

% Contrast Enhancement of the image
 SE = strel('Disk', 18);
 imgEnhanced = imadjust(img,[0.20 0.00 0.09; 0.83 1.00 0.52],...
     [0.00 0.00 1.00; 1.00 1.00 0.00], [1.10 2.70 1.00]);

% Decorrelation streching - useful for visulaizing image by reducing
% inter-plane autocorelation levels in an image.
 colorImg = decorrstretch(img);
 colorImg = imadjust(colorImg,[0.10; 0.79],[0.00; 1.00], 1.10);

 % Flipping an image
 a = fliplr(img(:,:,1));
 b = fliplr(img(:,:,2));
 c = fliplr(img(:,:,3));

 flippedImage = cat(3,a,b,c);

%% Creating the final photobooth strip.
 % list image name in desired order
 CompositeImage = [img; cyan; imgEnhanced; black_and_white;...
                   colorImg; flippedImage];
 imshow(CompositeImage);
```

Some of the useful image processing functions used above include rgb2gray, imadjust, and decorrstretch. Use the references and help menu to explore these functions further. The new images are combined at the end in a single column for the photobooth reel. This works because each matrix is the same size in the orientation they are concatenated, in this case, having the same number of columns since they are combined one on top of the other.

References

In addition to the web links provided in the lab, here are some classic resources on Image Processing which give both a practical and theoretical treatment of the subject.

Pratt, William K. Digital Image Processing. New York: John Wiley and Sons, 2001.

Gonzales, Rafael K. Woods, Richard E. Digital Image Processing 3E. New Jersey: Prentice Hall 2007.

Updated 1/5/2020 – Victor Miranda, Payal Singh, Brea Torquato