ECE 280L Laboratory Digital Image Processing #1

Version 1.0, 19 October 2020

Gustafson, Collins, Knox, Putri, Fleeting, Mainsah, and Cai

Contents

1 Introduction

This lab is centered on the topic of Digital Image Processing. In this lab, you will extend your programming abilities in MATLAB® and your knowledge of convolution to explore simple digital image processing techniques. This handout and supporting web pages will teach you about the fundamentals of images and image formats as well as specific MATLAB commands used to load, manipulate, and save images. They will also teach you how to extend one-dimensional convolution to two dimensions. There is a Pundit page that accompanies this document and that has the example codes and images. See https://pundit.pratt.duke.edu/wiki/ECE_280/Imaging_Lab_1

2 Objectives

The objectives of this project are to:

- Become familiar with different types of digital images (Black/White, Grayscale, and Color) as well as the conversions between them,
- Become familiar with using MATLAB's Image Processing Toolbox and the toolbox's built-in functions,
- Load, create, manipulate, and save images, and
- Understand and use MATLAB's built-in 2-D convolution function to spatially filter images and perform edge detection.

3 Assignment Preamble

You will be submitting discussions, images, and codes for eight different exercises. Everything will be collected in a single PDF you will upload to Gradescope. Also, the scripts (.m files) will be submitted to a Sakai Drop Box Folder in your account that you will create and name `IP1`. The specific names for the images and scripts will be given in the exercises themselves as well as in the summary of the exercises at the end of this document. Make sure the names are *exactly* what are asked for here. Also, **every title of every figure you submit needs to have your NetID in it**, so if the problem says "Make the title **Blue**" that really means the title should be `Blue (NetID)`. If there is not a specified title, make up one that makes sense given the contents of the plot and include your NetID.

4 Background

The formal, encyclopedic definition of a digital image reads as follows: "A digital image is a representation of a real image as a set of numbers that can be stored and handled by a digital computer. In order to translate the image into numbers, it is divided into small areas called pixels (picture elements). For each pixel, the imaging device records a number, or a small set of numbers, that describe some property of this pixel, such as its brightness (the intensity of the light) or its color. The numbers are arranged in an array of rows and columns that correspond to the vertical and horizontal positions of the pixels in the image."[1] During this lab, we will look at three main types of image: black & white, grayscale, and color. In each case, the image will be stored as one or more arrays of values that map to what each pixel looks like. For color images, there will be three arrays storing all the information.

[1] "Digital Images," Computer Sciences, Encyclopedia.com, (July 10, 2020) https://www.encyclopedia.com/computing/news-wires-white-papers-and-books/digital-images

4.1 Black & White Images

As you can imagine from the title, a black and white image is just that - a collection of black pixels and white pixels. As a result, each pixel really only needs to know one thing: is it black or white? Typically, this means these images will be stored with each entry being either a 0 (black) or 1 (white).

EXAMPLE 1: Black & White Images

For example, run the following in MATLAB:

```
a = [ 1 0 1 0 0; ...
      1 0 1 0 1; ...
      1 1 1 0 0; ...
      1 0 1 0 1; ...
      1 0 1 0 1 ];
figure(1); clf
imagesc(a)
colormap gray; colorbar
```

This program creates a 5x5 matrix of 0 and 1 values, then opens a figure and clears it. Next, it uses MATLAB's `imagesc` program to view the matrix as a scaled image, meaning MATLAB will map the minimum value of the array to the first color and the highest value of the array to the last color. MATLAB will then change the colormap to grayscale which makes the first color black and the last color white. Finally, MATLAB adds a colorbar so that you can relate the colors to numerical values.

Note that by default for integers, `image` assumes values between 0 and 255 so using the `image` command here would make an image that is basically all black. For this image, depending on which color you focus on, you will either see "if" or "Hi" in the figure. With enough pixels, you can certainly create more intricate graphical representations - but if you add different shades of gray, you can do even more.

4.2 Grayscale

A grayscale image differs from a black and white image in that each pixel is allowed to have one of several values between pure black and pure white. Typically, grayscale images store an 8-bit number for each pixel, allowing for $2^8 = 256$ different shades of gray for each.

EXAMPLE 2: Simple Grayscale Images

If you run the following in MATLAB:

```
b = 0:255;
figure(1); clf
image(b)
colormap gray; colorbar
```

you will see an image that goes from black to white in 256 steps from left to right. Grayscale images have 8 times as much information as black and white and typically take 8 times as much memory to store.

EXAMPLE 3: Less Simple Grayscale Images

Here is a more complex example showing the variations in grayscale:

```
[x, y] = meshgrid(linspace(0, 2*pi, 201));
z = cos(x).*cos(2*y);
figure(1); clf
imagesc(z)
axis equal; colormap gray; colorbar
```

Once again note the use of `imagesc` instead of `image` to automatically map the values in the matrix to the 256 values in the given map. In this case, the minimum value of -1 gets mapped to the first color (black) and the maximum value of +1 gets mapped to the last color (white). Also, the image dimensions are equalized to make it look square regardless of the shape of the figure window.

4.3 Color

A color image allows you to not only adjust the dark or light level of a pixel but also the hue ("which color") and saturation ("how intense does the color appear / is it washed out?"). MATLAB stores color images by storing the red, green, a blue levels for each pixel. Each of these values is stored in its own 8-bit number, meaning a color image takes up 3 times as much space as a grayscale image and 24 times as much space as a black and white image! While there are other methods of representing colors of a pixel, we are going to stick with the RGB model in this lab. Generally, MATLAB expects either an integer between 0 and 255 or a floating point number between 0 and 1 for each component.

EXAMPLE 4: Building an Image

To see an example of how you can build an image from three different color layers, take a look at the results of the following code:

```
rad = 100;
del = 10;
[x, y] = meshgrid((-3*rad-del):(3*rad+del));
[rows, cols] = size(x);
dist = @(x, y, xc, yc) sqrt((x-xc).^2+(y-yc).^2);
venn_img = zeros(rows, cols, 3);
venn_img(:,:,1) = (dist(x, y, rad.*cos(0), rad.*sin(0)) < 2*rad);
venn_img(:,:,2) = (dist(x, y, rad.*cos(2*pi/3), rad.*sin(2*pi/3)) < 2*rad);
venn_img(:,:,3) = (dist(x, y, rad.*cos(4*pi/3), rad.*sin(4*pi/3)) < 2*rad);
figure(1); clf
image(venn_img)
axis equal
```

The code produces a *three layer* matrix with the number of rows and columns determined by the parameters `rad` and `del`. The `rad` value will be used to generate three circles of equal radius (2*`rad`) centered on points evenly spaced around a circle of radius `rad`. The `del` value will provide some space between the circles and the edges of the image.

The first layer will represent red, the second layer will represent blue, and the third will represent green. With this code, the red layer will be 1 for all the pixels within a total distance of 200 from the location (100, 0). The green layer will be 1 for all pixels within a total distance of 200 from the location (50, -86.6). The blue layer will be 1 for all pixels within a distance of 200 from the location (-50, -86.6). These three layers describe three overlapping circles, and when you make the plot, you can see eight different regions:

- (0,0,0) not in any of the circles (black)
- (1, 0, 0) for the red circle
- (0, 1, 0) for the green circle
- (0, 0, 1) for the blue circle
- (1, 1, 0) for the intersection of the red and green circle (yellow)
- (1, 0, 1) for the intersection of the red and blue circle (magenta)
- (0, 1, 1) for the intersection of the green and blue circle (cyan)
- (1, 1, 1) for the intersection of the green and blue circle (white)

This example shows the extremes where the components are either fully on or fully off. Since there are three dimensions (for the red, green, and blue level), it is difficult to portray the full range of colors MATLAB can show. In fact, since there are just over 16 million different possible colors, it is impossible for most computers screens to display all the colors at once given that most screen resolutions top out at or below 8 megapixels.

EXAMPLE 5: Exploring Colors

If you want to see how two of the three components interact while keeping the third constant, you can use something similar to the following code (set to see what happens when the red and green levels change if the blue level is set to half strength).

```
1 [x, y] = meshgrid(linspace(0, 1, 256));
2 other = 0.5;
3 palette = zeros(256, 256, 3);
4 palette(:,:,1) = x;
5 palette(:,:,2) = y;
6 palette(:,:,3) = other;
7 figure(1); clf
8 imagesc(palette)
9 axis equal
```

For this code, the red component increases from left to right and the green component increases from top to bottom. The very middle of the image will be gray since each component is at 50%.

EXERCISE 1: Color Blending

Write a new script based on the one above called `IP1_EX1.m`. Change the code that makes the Venn diagram of colors such that the amount of color for each component is related to how far away the pixel is from a particular location (different for each location). The amount should be determined by the formula:

$$\text{Component}_k(x,y) = \frac{1}{1 + p\sqrt{(x - x_{c,k})^2 + (y - y_{c,k})^2)}}$$

where (x, y) is the value of (x, y) for a particular pixel, k is an index (1, 2, or 3 for red, green, or blue), $(x_{c,k}, y_{c,k})$ represents the location where a particular component should be its maximum, the p represents a parameter to determine how quickly a color fades as you move away from its center. The centers will be:

k	Component	$x_{c,k}$	$y_{c,k}$
1	Red	rad	0
2	Green	rad$\cdot \cos(2\pi/3)$	rad$\cdot \sin(2\pi/3)$
3	Blue	rad$\cdot \cos(4\pi/3)$	rad$\cdot \sin(4\pi/3)$

Create an image (using `image` since the values are floating point numbers between 0 and 1 for each layer) with $p = 0.05$ and another with $p = 0.005$. Use `axis equal` for each. Save the first file as `IP1_EX1_Plot1.png` and the second as `IP1_EX1_Plot2.png`. The images and the code will go in your lab report and you will also upload the code for this to your Sakai Drop Box. In the lab report, describe how changing p changed the images.

EXERCISE 2: Random colors

Write a script called `IP1_EX2.m` that will generate a 200x200x3 array of random numbers between - and 1 and display it. You can achieve this either by first creating a 200x200x3 matrix of zeros, then replacing each layer with a 200x200 matrix of uniformly distributed random numbers between 0 and 1, or by using MATLAB's ability to create 3D random matrices. Display the matrix using `image`. Use `axis equal` so the image part looks square. Run the code several times to see how the colors change. Save the image from one run as `IP1_EX2_Plot1.png`

The image and the code will go in your lab report and you will also upload the code for this to your Sakai Drop Box. In the lab report, discuss what you see in the image as well as how it changes from run to run.

5 1D Convolution

You have already learned about the process of convolution for both discrete and continuous signals. Digital images very much fall into the discrete category, but they are a bit more complicated than what you have seen so far in that they are two dimensional with the row and column index providing the two independent directions. Furthermore, with images, we are less concerned about the concept of a system response and more interested in how we might use a weighted average of pixel values to produce a new image. 2D Convolution will allow us to perform that task.

Let us first look at the issue from a 1D perspective. Imagine you have some discrete signal $x[n]$ and you want to find a signal that is based on the difference between the value of the signal at n and the value of the signal at $n-1$. You are therefore interested in creating a signal $y[n]$ where:

$$y[n] = x[n] - x[n-1]$$

The impulse response of this signal is:

$$h[n] = \delta[n] - \delta[n-1]$$

In MATLAB, you can perform this convolution with the `conv` command, where the arguments will be the discrete values of your original signal $x[n]$ and the discrete values of your impulse response $h[n]$.

EXAMPLE 6: 1D Convolution

Imagine that we have some set of $x[n]$ values:

$$x[n] = [1, 2, 4, 8, 7, 5, 1]$$

and we want to find the differences between those values. We can define h from its first non-zero value to its last non-zero value:

$$h[n] = [1, -1]$$

and then we can ask MATLAB to do the convolution:

```
x = [1, 2, 4, 8, 7, 5, 1]
h = [1, -1]
y = conv(x, h)
```

The result will be:

```
y =
     1     1     2     4    -1    -2    -4    -1
```

Among other things, notice that y is longer than either x or h! What happened here is the following (using parenthetical arguments to map to MATLAB):

$$y(n) = \sum_{m=1}^{m=7} x(m) \cdot h(n - m + 1)$$

which is the discrete version of convolution (with the +1 in the h argument to account for MATLAB being 1-indexed and also assuming that x or h at undefined index values will be considered 0). Here is a step-by-step examination:

- MATLAB flipped h to create [-1, 1]
- MATLAB aligned the far right of the flipped version of h with the far left of x, multiplied overlapping terms ($x(1) \cdot h(1)$, meaning the 1 from x and the 1 from the flipped h) and stored the result in the first element of y; which is to say, if $n = 1$ to calculate the first value of y,

m		1	2	3	4	5	6	7	
$x(m)$		1	2	4	8	7	5	1	
$h(n-m+1)$	-1	1							
$x(m) \cdot h(n-m+1)$	0	1	0	0	0	0	0	0	0

and $y(1)$ will be the sum of that last row, or 1.

- MATLAB slid the flipped version of h one space to the right, multiplied the overlapping terms (so $x(2) \cdot h(1)$ and $x(1) \cdot h(2)$) and stored the result in the second element of y; if $n = 2$:

m		1	2	3	4	5	6	7	
$x(m)$		1	2	4	8	7	5	1	
$h(n-m+1)$		-1	1						
$x(m) \cdot h(n-m+1)$	0	-1	2	0	0	0	0	0	0

  and $y(2)$ will be the sum of that last row, or 1.

- MATLAB slid the flipped version of h one more space to the right, multiplied the overlapping terms (so $x(3) \cdot h(1)$ and $x(2) \cdot h(2)$) and stored the result in the third element of y; if $n = 3$:

m		1	2	3	4	5	6	7	
$x(m)$		1	2	4	8	7	5	1	
$h(n-m+1)$			-1	1					
$x(m) \cdot h(n-m+1)$	0	0	-2	4	0	0	0	0	0

  and $y(3)$ will be the sum of that last row, or 2.

- MATLAB slid the flipped version of h one more space to the right, multiplied the overlapping terms (so $x(4) \cdot h(1)$ and $x(3) \cdot h(2)$) and stored the result in the fourth element of y; if $n = 4$:

m		1	2	3	4	5	6	7	
$x(m)$		1	2	4	8	7	5	1	
$h(n-m+1)$				-1	1				
$x(m) \cdot h(n-m+1)$	0	0	0	-4	8	0	0	0	0

  and $y(4)$ will be the sum of that last row, or 4.

- MATLAB repeated this process until $x(7)$ overlaps with $h(2)$, which is to say, if $n = 8$:

m		1	2	3	4	5	6	7	
$x(m)$		1	2	4	8	7	5	1	
$h(n-m+1)$								-1	1
$x(m) \cdot h(n-m+1)$	0	0	0	0	0	0	0	-1	0

  and $y(8)$ will be the sum of that last row, or -1.

EXAMPLE 7: 1D Convolution Using 'same'

MATLAB also provides the option to only return the "central" values of the convolution such that the resulting vector is the same size as x. The way to make that happen is to add the `'same'` option:

```
x = [1, 2, 4, 8, 7, 5, 1]
h = [1, -1]
y = conv(x, h, 'same')
```

The result will be:

```
y =
     1     2     4    -1    -2    -4    -1
```

Assuming h has N_h terms, and further assuming that N_h is smaller than the number of terms in x (N_x), using the `same` option means that MATLAB will not return results at the extreme edges of the convolution. The way MATLAB trims the convolution depends on how large h is as a total of $N_h - 1$ terms need to be removed.

If N_h is even, $N_h/2$ terms are removed from the beginning and $(N_h - 2)/2)$ are removed from the end. If N_h is odd, $(N_h - 1)/2)$ terms are removed from both the beginning and the end. This process tries to make $y(1)$ the result of "centering" the flipped version of h on the first value in x and similarly makes the last value $y(N_x)$ the result of centering the flipped version of h on the last value of x. If h has an even number of values, it cannot be perfectly centered so the flipped value is shifted one space to the right. This means convolution being performed is:

$$y(n) = \sum_{m=1}^{m=Nx} x(m) \cdot h(n - m + \lceil N_h/2 \rceil)$$

with $n = [1, N_x]$ and where the $\lceil\ \rceil$ operator represents rounding up. Fortunately you do not have to worry about that calculation - MATLAB will do that!

EXERCISE 3: 1D Moving Average

You can calculate the "moving average" of a data set by convolving the data set with an h of length N_h where all the entries in h are equal to $1/N_h$. Write a script called `IP1_EX3.m` that starts with the following code:

```
tc = linspace(0, 1, 101);
xc = humps(tc);
td = linspace(0, 1, 11);
xd = humps(td);
figure(1); clf
plot(tc, xc, 'b-')
hold on
plot(td, xd, 'bo')
hold off
```

This will generate two data sets using the built-in `humps` command. The `tc` and `xc` values will represent a refined data set using 101 points to show the overall shape of the `humps` command over the domain. The `xd` values will represent a set of 11 "discrete" points we can to smooth using convolution. Note that below you will only be convolving with `xd`.

Use convolution to create one vector that stores the 2-point moving average of `xd` (with the same number of data points as `xd`) and another vector with the 5-point moving average of `xd` (again, with the same number of data points as `xd`). Create a figure that has the original plot above (solid blue "continuous" line with blue circles at "discrete" points) and then overlays overlays the 2-point moving average with red circles connected by solid lines along with the 5-point moving average with green circles connected by solid lines. Title the graph "Moving Averages." Save the figure as `IP1_EX3_Plot1.png`. The image and the code will go in your lab report and you will also upload the code for this to your Sakai Drop Box. In the lab report, discuss the differences between the 2-point and 5-point moving averages. Be sure to at least answer the following two questions: (1) Does one of the smoothed curves look more or less symmetrically-smoothed than the other and (2) because the answer to that should be yes, which one and why do you think that is?

EXERCISE 4: Derivative Approximations

You can calculate a numerical approximation to the first derivative of a data set by convolving the data set with an h of $[1, 0, -1]/(2\Delta t)$. Start a program called `IP1_EX4.m` with:

```
tc = linspace(0, 1, 101);
xc = humps(tc);
deltatc = tc(2)-tc(1);
td = linspace(0, 1, 11);
xd = humps(td);
deltatd = td(2)-td(1);

figure(1); clf
plot(tc, xc, 'b-')
hold on
plot(td, xd, 'bo')
hold off
title('Values')

figure(2); clf
twopointdiff = diff(xc)/deltatc;
twopointdiff(end+1)=twopointdiff(end);
plot(tc, twopointdiff, 'b-')
hold on
plot(tc(1:10:end), twopointdiff(1:10:end), 'bo')
hold off
title('Change Me')
```

This will again generate two data sets using the built-in `humps` command. The first figure shows the function as well as the eleven points along that function we are using for the data set. The second figure contains a two-point-forward approximation for the first derivative of the data set containing 101 points along with circles at the eleven values where we know `xd`. Use convolution to create a vector that stores the same-size convolution of `xd` with `[1, 0, -1] / 2 / deltatd`, then overlay a plot of this derivative as a function of `td` onto figure 2 using magenta circles. Title the graph "Derivative Approximation (11 points)" (rather than the current value of `Change Me`. Save this figure as `IP1_EX4_Plot11.png`. Once you have saved this figure, you are going to run the code again but with a different number of points in `xd`, a different title for the graph, and a different file name. Change the code where `td` is created so that there are 51 points, change the title to "Derivative Approximation (51 points)" and change the plot's filename to `IP1_EX4_Plot51.png`. Re-run the code for these parameters.

The images and the code will go in your lab report and you will also upload the code for this to your Sakai Drop Box. You only need to document one version of the code (either for the 11-point or 51-point approximation). In your lab report you will discuss the differences between the derivative approximations using 11 points and the derivative approximations using 51 points. Be sure to clearly identify the calculations you believe have unacceptable error (and there are some) and provide an explanation for why those errors occur. You do not need to include either of the `figure(1)` from these scripts but you may reference them in your discussion.

6 2D Discrete Convolution

Now we can apply the concept of convolution to an image. You will basically be taking a 2D matrix $h(r, c)$, flipping it both vertically and horizontally, shifting it relative to some matrix $x(r, c)$, multiplying the values that now overlap, and adding those products together. The general form of 2D discrete convolution is:

$$y(r, c) = \sum_i \sum_j x(i, j)\, h(r - i + 1, c - j + 1)$$

6.1 2D Convolution Graphical Example (complete with caveats)

There is an excellent example of 2D convolution (with caveats) at: http://www.songho.ca/dsp/convolution/convolution2d_example.html **Note:** that web page does two things that are different from how MATLAB works: the web page has the row and column indices flipped and the web page uses 0-indexing instead of 1 indexing. The images are well worth visiting the page, however! The example has:

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

If you flip h both vertically and horizontally, you get:

$$\text{flipped } h = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

If you are performing 2D convolution with the `'same'` option, $y(1, 1)$ will be the result of centering the flipped version of h on $x(1, 1)$, multiplying the overlapping values, and adding those products together. This is what the web page calls $y[0, 0]$. The gray square represents the flipped and shifted h matrix. To get $y(1, 2)$ you would move the flipped and shifted h matrix one space to the right (what the web page would call y[1, 0] since it reverses rows and columns from our perspective). If you look at the remaining examples, you get a new entry in y by centering the flipped h on a new location in x.

6.2 Basic Blurring

Now imagine that you want to blur an image. One way to do this is to replace each pixel with the average of itself and its neighbors. A 2D blur is very similar to the moving averages above. You can create an R_hxC_h matrix h whose entries are all equal to $1/(R_h C_h)$, and then you can convolve an image with that h to produce a blurred version of that image. MATLAB has a built-in `conv2` command to do just that. The `conv2` command has the same `'same'` option as `conv` to produce a matrix the same size as x.

EXAMPLE 8: 10x10 Blurring

Use the following code to load one of MATLAB's built-in grayscale images, convolve it with a 10x10 matrix of 0.01's, and then display the result in a new figure:

```
x = imread('coins.png');
h = ones(10, 10)/10^2;
y = conv2(x, h, 'same');
figure(1); clf
image(x)
axis equal; colormap gray; colorbar
title('Original')
figure(2); clf
image(y)
axis equal; colormap gray; colorbar
title('10x10 Blur')
```

EXERCISE 5: Boxes and Rectangles and Voids (Oh My!)

Write a program called `IP1_EX5.m` that starts with the code above and save figure 1 (the blurred one) as `IP1_EX5_Plot1.png`. Then add code that will blur the original image using a properly scaled 2 row, 50 column blur and display it using `image` in figure 2. Title the image "2x50 Blur" and save it as `IP1_EX5_Plot2.png`. Next, write code that will blur the image using a properly scaled 50 row, 2 column blur and display it using `image` in figure 3. Title the image "50x2 Blur" and save it as `IP1_EX5_Plot3.png`. The images and the code will go in your lab report and you will also upload the code for this to your Sakai Drop Box. In the lab report, discuss the differences between the 10x10, 2x50, and 50x2 blurs as well as any interesting "artifacts" you see in the images.

6.3 A Better Convolution Option

While the `'same'` option can work when we want the output to have the same number of entries as the input, it may be problematic to have MATLAB assume an image is 0 for locations where part of the h matrix does not overlap with part of the x matrix.

EXAMPLE 9: Make No Assumptions

Given that, there is another option for `conv` and `conv2` known as `'valid'`. In this case, the only values returned will be for entries where flipping and shifting the h matrix results in full overlap with x. For example, in 1D,

```
x = [1, 2, 4, 8, 7, 5, 1]
h = [1, -1]
y = conv(x, h, 'valid')
```

The result will be:

```
y =
     1     2     4    -1    -2    -4
```

which is different from using the `'same'` option in that there is one fewer value. The seventh value with the `'same'` option assumed there was an extra 0 on the end of x. That can be convenient for making comparative plots, but can also lead to misinformation when it comes to blurring and other operations with images. As a result, for image processing, we will stick with the `'valid'` option for `conv2`. We may end up with a smaller image, but the information presented will not have to make any assumptions about "missing" x values and generally the h matrix is so small relative to the x image that not much is cropped. We will, however, generally want to use *square* h matrices so that the same amount is cropped vertically and horizontally. Soon you will see that sometimes we will want to combine two or more convolved images and those must all be the same size.

EXERCISE 6: Same Song, Different Verse, These Images Aren't Quite As Big As At First

Using the same coin image as above, write a program in a script called `IP1_EX6.m` that will blur the image using a properly scaled 10 row, 10 column blur, but with the `'valid'` option instead of `'same'`. Title the image "Valid 10x10 Blur" and save it as `IP1_EX6_Plot1.png`. Repeat the process for the 2x50 blur in `IP1_EX6_Plot2.png` and the 50x2 blur in `IP1_EX6_Plot3.png`. In each case use `axis equal` and a gray colormap. Include a colorbar. The images and the code will go in your lab report and you will also upload the code for this to your Sakai Drop Box. In the lab report, discuss the differences between the `'same'` and `'valid'` versions of the convolution. **From this point forward in the assignment, always use the `'valid'` option with `conv2`**

6.4 Basic Edge Detection

In addition to blurring an image, you can use convolution to perform *edge detection* - that is, to determine where there are quick changes in grayscale or color. Since you want to find where there is a large *difference* between values, the h matrix will generally have some positive and some negative entries.

EXAMPLE 10: Basic Edge Detection and Display

Use the following code to create a grayscale image with some different gray level rings in it, convolve the image with matrices meant to detect left-right and up-down edges, then display the results in new figures. Note that because the edge values may be positive or negative, we are using the *imagesc* command to see a scaled version of the image along with `colorbar` to get an idea of what the shades mean. Furthermore, because we will be "losing" columns or rows based on how wide or tall h is (due to using the `'valid'` option, we are using a square h matrix that is adding together the differences for *two* rows or columns at once so that the vertical and horizontal edge detections are the same size. This is necessary for when we combine the different directions in the last image.

```
[x, y] = meshgrid(linspace(-1, 1, 200));
z1 = (.7<sqrt(x.^2+y.^2)) & (sqrt(x.^2+y.^2)<.9);
z2 = (.3<sqrt(x.^2+y.^2)) & (sqrt(x.^2+y.^2)<.5);
zimg = 100*z1+200*z2;
figure(1); clf
image(zimg); axis equal; colormap gray; colorbar; title('Original')

hx = [1 -1; 1 -1];
edgex = conv2(zimg, hx, 'valid');
figure(2); clf
imagesc(edgex); axis equal; colormap gray; colorbar; title('Vertical Edges')

hy = hx';
edgey = conv2(zimg, hy, 'valid');
figure(3); clf
imagesc(edgey); axis equal; colormap gray; colorbar; title('Horizontal Edges')

edges = sqrt(edgex.^2 + edgey.^2);
figure(4); clf
imagesc(edges); axis equal; colormap gray; colorbar; title('Edges')
```

EXERCISE 7: Fun With Convolution

For this exercise you will be exploring different kernels, or filter masks. Go to the Wikipedia page at: https://en.wikipedia.org/wiki/Kernel_(image_processing) and specifically note the table of different image kernels. The convolution formula given is a little different from the one that we use – later in the page it mentions that asymmetric kernels have to be flipped to perform convolution. Note that there are a few different examples of kernels that blur or smooth the image and a few different examples that detect edges. Given that, using the built-in grayscale coin image from above, and always using the options to make the axes equal, using a gray colormap, and including a colorbar, write a script called `IP1_EX7.m` that does all the following:

- Convolve the image with a 5x5 Gaussian blur approximation kernel. Display the result with `image`. Title the graph "Coin Gaussian Blur." Save this as `IP1_EX7_Plot1.png`
- Convolve the image with a 3x3 Prewitt operator that will detect edges as the gray level changes from left to right. You will need to do some research to find out what a Prewitt operator is. Call this matrix `CoinsEdgeX` and display the normalized absolute value of the results with `imagesc`. Title the graph "Coin Vertical Edges." Save this as `IP1_EX7_Plot2.png`
- Convolve the image with a 3x3 Prewitt opeartor that will detect edges as the gray level changes from top to bottom. Call this matrix `CoinsEdgeY` and display the normalized absolute value of the results with `imagesc`. Title the graph "Coin Horizontal Edges." Save this as `IP1_EX7_Plot3.png`
- Find the (element based) square root of the sum of the (element based) squares of `CoinsEdgeX` and `CoinsEdgeY` and display the normalized value of the results with `imagesc`. Title the graph "Coin Edges." Save this as `IP1_EX7_Plot4.png`
- Convolve the image with a 3x3 kernel on Wikipedia that looks at differences in all eight directions; specifically, use the third edge detection kernel on the Kernel page,

$$h = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

  Display the normalized absolute value of the results with `imagesc`. Title the graph "Alternate Coin Edges." Save this as `IP1_EX7_Plot5.png`

The images and the code will go in your lab report and you will also upload the code for this to your Sakai Drop Box. In the body of the lab report, discuss how each process changed the image and then discuss the difference between the last two images in terms of how well you feel each found the edges.

6.5 2D Convolution in Color

So far, the examples of 2D convolution have been for grayscale images. You can also use convolution with color images - you simply have to use the convolution on each *layer*. First, you should look at the different layers.

EXAMPLE 11: Chips!

The following code will load an image into a three-layer array and then it will display each layer separately first as a gray scale image and then using a colormap made especially to show the red, green, and blue components in their own colors.

```
img = imread('coloredChips.png');
figure(1); clf
title('Original')
image(img); axis equal
vals = (0:255)'/255;
names = {'Red', 'Green', 'Blue'}
for k = 1:3
    figure(k+1); clf
    image(img(:,:,k)); axis equal
    colormap gray; colorbar
    title(names{k}+" as Gray")
    figure(k+4)
    image(img(:,:,k)); axis equal
    cmap = zeros(256, 3);
    cmap(:,k) = vals;
    colormap(cmap); colorbar
    title(names{k}+" as "+names{k})
end
```

Note the use of `image` instead of `imagesc` here. We do not want the individual components to have their levels "stretched out" to the full range of the colorbar; instead, we want the value between 0 and 255 to be represented by the gray or color level between 0 and 255. The `cmap` matrix is changed each time to provide color codes for each of the 256 possible values within each *layer*. For red, only the first component will be nonzero; for green the second; for blue the third.

EXAMPLE 12: Chip Edges!

Now that you can see the individual layers, all you need to do to perform convolution in color is to perform convolution on each layer and then combine the layers to make an image again. Look at the following example code to load a built-in image, convolve it with a Sobel operator to detect vertical edges (i.e. when a color changes as you go from left to right), display each color's edges in grayscale, display the absolute value of each color's edges in grayscale, display a color edge, and display the normalized 2-norm of the color edge. Note the `imagesc` command based on what the maximum and minimum possible values are for edge detection. With this operator, the maximum edge for each color will be 4*255.

```
img = imread('coloredChips.png');
figure(1); clf
image(img); axis equal

h = [1 0 -1; 2 0 -2; 1 0 -1]
for k=1:3
    y(:,:,k) = conv2(img(:,:,k), h, 'valid');
    figure(1+k); clf
    imagesc(y(:,:,k), [-1020, 1020]);
    axis equal; colormap gray; colorbar
    figure(4+k); clf
    imagesc(abs(y(:,:,k)), [0, 1020]);
    axis equal; colormap gray; colorbar
end
%%
yx = (y+max(abs(y(:)))) / 2 / max(abs(y(:)));
figure(8); clf
image(yx); axis equal

yg = sqrt(y(:,:,1).^2 + y(:,:,2).^2 + y(:,:,3).^2);
ygs = abs(yg) / max(yg(:)) * 255;
figure(9); clf
image(ygs); colormap gray; axis equal
```

EXERCISE 8: Putting It All Together

You will now apply everything you have learned to process a full-color image known as a "test card." Go to the Wikipedia page at: https://en.wikipedia.org/wiki/Test_card and read a bit about the purpose and history of test cards. Click on the picture of the PM5544 test pattern (created by Mediawiki user Zacabeb and released to the public domain) and download it to your computer wherever you are saving the programs for this lab. Write a program that will perform the tasks listed below. Note in all cases that you should clear the figure first and use `axis equal`. Use the `'valid'` option for convolution in every case. You will not need colorbars for any of these as they will all be in three layer color.

- Display the original image using `image`. Title the graph "Original Test Card." Save this as `IP1_EX8_Plot0.png`.
- Convolve the image with a normalized 21x21 box blur kernel. Convert the result (say, `y1`) to unsigned 8 bit integers with the command `image(uint8(y1))` and display the result with `image`. Title the graph "Box Blur Test Card." Save this as `IP1_EX8_Plot1.png`. If you do not convert the values to unsigned integers, the fact that there are floating point values will make `image` assume the component values should be between 0 and 1 versus 0 and 255, resulting in a mostly-white image.
- Convolve the image with a 5x5 Gaussian blur approximation kernel. Convert the result to unsigned 8 bit integers and display the result with `image`. Title the graph "Gaussian Blur Test Card."Save this as `IP1_EX8_Plot2.png`
- Convolve the image with a 3x3 Sobel operator that will detect edges as colors change from left to right. Display the normalized absolute value of the results with `image`. Title the graph "Sobel Vertical Edges Test Card."Save this as `IP1_EX8_Plot3.png`
- Convolve the image with a 3x3 Sobel operator that will detect edges as colors change from top to bottom. Display the normalized absolute value of the results with `image`. Title the graph "Sobel Horizontal Edges Test Card."Save this as `IP1_EX8_Plot4.png`
- Display the normalized results of taking the square root of the sum of the squares of the results for the two Sobel operators with `image`. Title the graph "Sobel Edges Test Card." Save this as `IP1_EX8_Plot5.png`

The images and the code will go in your lab report and you will also upload the code for this to your Sakai Drop Box. Also discuss the differences between the different blurs as well as what you see in the edge detection images.

7 Assignment Summary

This lab assignment is a little different from others in that there are eight targeted exercises to do rather than a full lab report with an introduction, discussion, conclusion, etc. Also, there is no extension - correctly completing and documenting the eight exercises will earn 100 for this assignment.

The discussions for each will be in the body of a lab document. The codes and images will be included in your lab document; there is a LaTeX skeleton available that has the infrastructure of the lab document already done. You will also be uploading your scripts to a Dropbox folder you create on Sakai; this will make it easier for the TAs to troubleshoot codes if they have the `.m` files. When you are getting ready to submit the codes for the lab, please go to the Sakai page **for the lab** - which is to say the ECE 280L9.01 (the "9" is the main thing to look for) page, go to the Drop Box link in the navigation bar at left, and in the `ACTIONS` drop-down box to the right of your name, select `Create Folder`. Make the folder name `IP1` and make sure you upload your codes with the appropriate names to that folder.