ECE 280L Fall 2024

Laboratory 5:
Image Processing 1

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 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 2D arrays of values that map to what each pixel looks like. For color images, there will be three 2D arrays storing all the information.

3.1 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

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

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.

3.2 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:

```
1 b = 0:255;
2 figure(1); clf
3 image(b)
4 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:

```
1 [x, y] = meshgrid(linspace(0, 2*pi, 201));
2 z = cos(x).*cos(2*y);
3 figure(1); clf
4 imagesc(z)
5 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.

3.3 Color

A color image allows you to adjust not only 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, and blue levels for each pixel. Each of these values is stored in its own 8-bit number, meaning a color image takes up to 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 is used to generate three circles of equal radius ($2 \times$ `rad`) centered on points evenly spaced around a circle of radius `rad`. The `del` value provides some space between the circles and the edges of the image.

The first layer represents red, the second layer represents blue, and the third represents green. With this code, the red layer is 1 for all the pixels within a total distance of 200 from the location (100, 0). The green layer is 1 for all pixels within a total distance of 200 from the location (50, -86.6). The blue layer is 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)

Pre-lab Deliverable (1/6): Which lines of code in Example 4 would you change to most directly change the intensity of each color?

1. Line 3
2. Lines 5-6
3. Lines 7-11
4. Lines 12-14

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%.

Pre-lab Deliverable (2/6): What would you change line 4 of the Example 5 code to so that the red component is 0 along the line x = y and gets brighter as we move further away?

1. `palette(:,:,1) = abs(x+y)`
2. `palette(:,:,1) = abs(x-y)`
3. `palette(:,:,1) = 1`
4. `palette(:,:,1) = abs(x/y)`

3.4 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:

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

The result will be:

```
1 y =
2      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

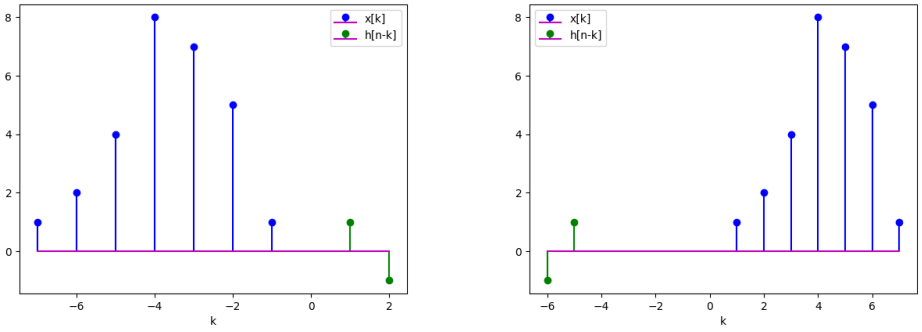
  and $y(4)$ will be the sum of that last row, or 4.

- 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.

Pre-lab Deliverable (3/6): Which of the images in Fig. 1 accurately visualizes the setup of the convolution described above, after h has been flipped but before it has been aligned?



(a) (b)

(c) (d)

Figure 1. Example 6, x[k] and h[n-k] (?)

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:

```
1 x = [1, 2, 4, 8, 7, 5, 1]
2 h = [1, -1]
3 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 that the 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!

3.5 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:

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

3.5.1 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. First, the web page has the row and column indices flipped, and it also 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.

3.5.2 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 average filter you created in Lab 2. 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:

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

Pre-lab Deliverable (4/6): Which of the following lines of code could be substituted for line 2 in Example 8 to create a 2×50 blurring kernel?

1. `h = ones(50, 2)/100;`
2. `h = ones(2, 50)/2^2;`
3. `h = ones(2, 50)/100;`

3.5.3 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,

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

The result will be:

```
1 y =
2     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.

3.5.4 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.

```
1 [x, y] = meshgrid(linspace(-1, 1, 200));
2 z1 = (.7<sqrt(x.^2+y.^2)) & (sqrt(x.^2+y.^2)<.9);
3 z2 = (.3<sqrt(x.^2+y.^2)) & (sqrt(x.^2+y.^2)<.5);
4 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, [-512, 512]);
axis equal; colormap gray; colorbar; title('Vertical Edges')

hy = hx';
edgey = conv2(zimg, hy, 'valid');
figure(3); clf
imagesc(edgey, [-512, 512]);
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')
```

Pre-lab Deliverable (5/6): Which of the following options, based on Example 10, would most effectively detect horizontal edges?

1. $[2, 2; -2, -2]$
2. $[2, -4; 4, -2]$
3. $[2, 2; 2, 2]$
4. $[-2, 2; -2, 2]$

3.5.5 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. Also note that the *normalized* values for the colorful and absolute edges are presented with `image` versus `imagesc` since they are, in a word, normalized.

```
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]);
```

```
13      axis equal; colormap gray; colorbar
14  end
15  %%
16  yx = (y+max(abs(y(:)))) / 2 / max(abs(y(:)));
17  figure(8); clf
18  image(yx); axis equal
19
20  yg = sqrt(y(:,:,1).^2 + y(:,:,2).^2 + y(:,:,3).^2);
21  ygs = abs(yg) / max(yg(:)) * 255;
22  figure(9); clf
23  image(ygs); colormap gray; axis equal
```

Pre-lab Deliverable (6/6): How would you change line 5 of the code in Example 12 to make it into a Sobel operator that detects when a color changes as you go from top to bottom? You may find the information at this Wikipedia page helpful: https://en.wikipedia.org/wiki/Sobel_operator

1. `h = [-1 0 1; -2 0 2; -1 0 1]`
2. `h = [1 2 1; 0 0 0; -1 -2 -1]`
3. `h = [-1 -2 -1; 0 0 0; 1 2 1]`
4. `h = [1 1 1; 0 0 0; -1 -1 -1]`

4 Pre-Laboratory Assignment

For your prelab, provide answers on Gradescope to each of the (6) Pre-lab Deliverables based on the Background information provided above.

The questions are listed below for your convenience:

1. Which lines of code in Example 4 would you change to most directly change the intensity of each color?

   (a) Line 3
   (b) Lines 5-6
   (c) Lines 7-11
   (d) Lines 12-14

2. What would you change line 4 of the Example 5 code to so that the red component is 0 along the line x = y and gets brighter as we move further away?

   (a) `palette(:,:,1) = abs(x+y)`
   (b) `palette(:,:,1) = abs(x-y)`
   (c) `palette(:,:,1) = 1`
   (d) `palette(:,:,1) = abs(x/y)`

3. Which of the images in Fig. 1 accurately visualizes the setup of the convolution described in Example 6, after h has been flipped but before it has been aligned?

4. Which of the following lines of code could be substituted for line 2 in Example 8 to create a 2×50 blurring kernel?

   (a) `h = ones(50, 2)/100;`
   (b) `h = ones(2, 50)/2^2;`
   (c) `h = ones(2, 50)/100;`

5. Which of the following options, based on Example 10, would most effectively detect horizontal edges?

   (a) $[2, 2; -2, -2]$
   (b) $[2, -4; 4, -2]$
   (c) $[2, 2; 2, 2]$
   (d) $[-2, 2; -2, 2]$

6. How would you change line 5 of the code in Example 12 to make it into a Sobel operator that detects when a color changes as you go from top to bottom? You may find the information at this Wikipedia page helpful: https://en.wikipedia.org/wiki/Sobel_operator

   (a) `h = [-1 0 1; -2 0 2; -1 0 1]`
   (b) `h = [1 2 1; 0 0 0; -1 -2 -1]`
   (c) `h = [-1 -2 -1; 0 0 0; 1 2 1]`
   (d) `h = [1 1 1; 0 0 0; -1 -1 -1]`

5 Instructions

5.1 Exercise 1: Color Blending

1. Create a new MATLAB script and title it `IP1_EX1.m`. Copy the code currently specified in Example 4 into the script.

2. Adjust the code that makes the Venn diagram of colors such that the amount of color for each component is determined by the formula:

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

   In this formula:

   - (x, y) are the x and y values for a particular pixel.
   - k is an index (1,2, or 3 for red, green, or blue, respectively).
   - p determines how quickly a color fades as you move away from its center.
   - $(x_{c,k}, y_{c,k})$ is the location where a particular component's intensity should be at its maximum, i.e. the center of the circle in the Venn diagram.

   (a) Set the values of $x_{c,k}$ and $y_{c,k}$ for each circle based on the following table:

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$\cdot\sin(4\pi/3)$

   Checkpoint (1/10): Show your modified code to your TA. Discuss how you anticipate changing p to affect the resulting Venn diagram.

3. Set $p = 0.05$ and export the resulting image as follows:

   (a) Set the title of the image to be `Venn Diagram p = 0.05 (NetID)`

   (b) Save the image as `IP1_EX1_Plot1.png`.

4. Repeat the same process with $p = 0.005$, changing the title accordingly and saving the image as `IP1_EX1_Plot2.png`.

   Deliverable (1/16): In your report, describe the image output by the code in this exercise. How did changing p affect the image?

   Deliverable (2/16): Include the following files in your report:

   - `IP1_EX1.m`
   - `IP1_EX1_Plot1.png`
   - `IP1_EX1_Plot2.png`

5.2 Exercise 2: Random Colors

1. Create a new script and title it `IP1_EX2.m`.
2. In this new script, generate a $200 \times 200 \times 3$ array of random numbers between 0 and 1 and save this to the variable X.
3. Display the matrix using `equal`.
   (a) Use `axis equal` and set the title to be `Random Color Image (NetID)`

   Discussion (1/3): Why does the image look like this? Does the distribution of colors in the image look roughly uniform?
4. Run your code several times, exporting the image from any one run as `IP1_EX2_Plot1.png`.

   Deliverable (3/16): In your report, describe the output image and how it changes between runs.

   Deliverable (4/16): Include the following files in your report:

   - `IP1_EX2.m`
   - `IP1_EX2_Plot1.png`

5.3 Exercise 3: 1D Moving Average

1. Create a new script titled `IP1_EX3.m` and copy in 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
```

   Checkpoint (2/10): Run the code and show the output to your TA. Discuss with your TA what the relationship is between `xc` and `xd`.
2. Create a vector h of length 2, where each entry in h is equal to $\frac{1}{2}$. This will be used to calculate the 2-point moving average of `xd`.
   - **NOTE:** In general, you can calculate the "N-point moving average" of a data set by convolving the data set with an h of length N where all the entries in h are equal to $1/N$.
3. Calculate the 2-point moving average of `xd` by convolving h with xd and save this to the variable `y2`.
   - Refer to the code in Example 7 for reference on how to perform convolution while maintaining dimensionality of your input.

4. Repeat steps 2 and 3 to calculate the 5-point moving average and save it to the variable `y5`.
5. Create a figure that contains the following and save it as `IP1_EX3_Plot1.png`:
   (a) The original plots of `xc` and `xd`
   (b) A plot of the 2-point moving average with red circles connected by solid lines
   (c) A plot of the 5-point moving average with green circles connected by solid lines
   (d) The title `Moving Averages (NetID)`

   Deliverable (5/16): In your report, discuss the differences between the 2-point and 5-point moving averages. In particular, answer the following questions:

   (a) Does one of the smoothed curves look more or less symmetrically-smoothed than the other?
   (b) Which one and why do you think that is?

   Deliverable (6/16): Include the following files in your report:

   - `IP1_EX3.m`
   - `IP1_EX3_Plot1.png`

5.4 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 $[1, 0, -1]/(2\Delta t)$.

1. Create a new script titled `IP1_EX4.m` and copy in the following code:

```
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
```

```
20      plot(tc(1:10:end), twopointdiff(1:10:end), 'bo')
21      hold off
22      title('Change Me')
```

Checkpoint (3/10): Run the code and show your TA the two figures that were produced. What is `twopointdiff`?

2. Perform a same-size convolution of a vector `h = [1,0,-1]/2/deltatd` with `xd` to get an approximation for the derivative of `humps`.
3. Overlay a plot of the result of the convolution onto figure 2 using magenta circles.
4. Title the graph `Derivative Approximation (11 points) (NetID)` and save it as `IP1_EX4_Plot11.png`

   Discussion (2/3): How do you think adding more points to `td` will affect the relationship between `twopointdiff` and our derivative approximation?

5. Change the code where `td` is generated so that there are 51 points, rather than 11, and rerun the code.

   (a) This time, title your figure `Derivative Approximation (51 points) (NetID)` and save it as `IP1_EX4_Plot51.png`.

   Deliverable (7/16): In your lab report, discuss the differences between the derivative approximations using 11 points and using 51 points. Clearly identify the calculations you believe have unacceptable error and provide an explanation for why those errors occur.

   Deliverable (8/16): Include the following files in your report:

   - `IP1_EX4.m`
   - `IP1_EX4_Plot11.png`
   - `IP1_EX4_Plot51.png`

5.5 Exercise 5: Boxes and Rectangles and Voids (Oh My!)

1. Create a new script called `IP1_EX5.m` that starts with the code in Example 8 and save the blurred figure as `IP1_EX5_Plot1.png`.
2. Edit the definition of `h` to blur the image using a 2 row, 50 column blur and run the code.

   Checkpoint (4/10): Show your TA the output of your 2×50 blur.

3. Title the image `2x50 Blur (NetID)` and save it as `IP1_EX5_Plot2.png`.
4. Edit `h` to make a 50×2 blur and display the result in another figure. Title that image `50x2 Blur (NetID)` and save it as `IP1_EX5_Plot3.png`

Deliverable (9/16): In your lab report, discuss the differences between the three different blurs used (10x10, 2x50, and 50x2), as well as any interesting "artifacts" you see in the images.

Deliverable (10/16): Include the following files in your report:

- `IP1_EX5.m`
- `IP1_EX5_Plot1.png`
- `IP1_EX5_Plot2.png`
- `IP1_EX5_Plot3.png`

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

1. Create a new script called `IP1_EX6.m` and load in the same coin image as used in Exercise 5.
2. Referencing the code from the previous exercise, write code to blur the image using a 10×10 blur, but with the `` `valid' `` option instead of `` `same' ``.
   - **NOTE:** See Example 9 for an in-depth explanation of the difference between the `` `valid' `` and `` `same' `` options for `conv2`.

   Checkpoint (5/10): Show your output to your TA. What differences do you see between this output image and the corresponding one from exercise 5?
3. Title the image `Valid 10x10 Blur (NetID)` and save it as `IP1_EX6_Plot1.png`.
4. Repeat steps 2 and 3 for a 2×50 blur and a 50×2 blur, adjusting the figure titles accordingly and saving them as `IP1_EX6_Plot2.png` and `IP1_EX6_Plot3.png`, respectively.

   Deliverable (11/16): In your lab report, discuss the differences between the `` `same' `` and `` `valid' `` versions of the convolution.

   Deliverable (12/16): Include the following files in your report:

   - `IP1_EX6.m`
   - `IP1_EX6_Plot1.png`
   - `IP1_EX6_Plot2.png`
   - `IP1_EX6_Plot3.png`

5.7 Exercise 7: Fun with Convolution

For this exercise you will be exploring different kernels, or filter masks. Each kernel will be based on the description available on the Wikipedia page at https://en.wikipedia.org/wiki/Kernel_(image_processing).

1. Create a script called `IP1_EX7.m` and import the same coin image from the last few exercises.

2. Based on the example in the table of different image kernels from the Wikipedia page, generate a 5×5 Gaussian blur approximation kernel.
3. Convolve the image with your kernel and display the result with `image`.

   Checkpoint (6/10): Show your TA the resulting, blurred image.
4. Title the image `Coin Gaussian Blur (NetID)` and save it as `IP1_EX7_Plot1.png`.
5. Reference the Wikipedia page at https://en.wikipedia.org/wiki/Prewitt_operator to generate a 3×3 Prewitt operator that will detect edges as the gray level changes from left to right.
6. Convolve the original image with this new kernel. Save the result to a variable called `CoinsEdgeX` and display this using `imagesc`.
7. Title the image `Coin Vertical Edges (NetID)` and save it as `IP1_EX7_Plot2.png`.
8. Now, generate a 3×3 Prewitt operator that will detect edges as the gray level changes from top to bottom.
9. Repeat steps 6 and 7 using your new kernel, saving the convolved result to `CoinsEdgeY`, titling the image `Coin Horizontal Edges (NetID)`, and saving it as `IP1_EX7_Plot3.png`.

   Checkpoint (7/10): Show your TA the two plots generated via convolution with Prewitt operators. Explain the difference between the two of them.

   Discussion (3/3): What about the Prewitt operator influences which direction it detects edges in? How can you relate this to the derivative approximation you calculated in Exercise 4?
10. Find the (element based) square root of the sum of the (element based) squares of `CoinsEdgeX` and `CoinsEdgeY` and display the result with `imagesc`.
11. Title the resulting image `Coin Edges (NetID)` and save it as `IP1_EX7_Plot4.png`.
12. Finally, convolve the original image with the following kernel:

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

13. Title the image `Alternate Coin Edges (NetID)` and, displaying the result with `imagesc`, save it as `IP1_EX7_Plot5.png`.

    Checkpoint (8/10): Show your last two images to your TA and discuss the similarities/differences between the two.

Deliverable (13/16): In your lab report, discuss:

- how each process changed the image, and
- the difference between the last two images in terms of how well you feel each found the edges

Deliverable (14/16): Include the following files in your lab report:

- `IP1_EX7.m`
- `IP1_EX7_Plot1.png`
- `IP1_EX7_Plot2.png`
- `IP1_EX7_Plot3.png`
- `IP1_EX7_Plot4.png`
- `IP1_EX7_Plot5.png`

5.8 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. **NOTE: Make sure to use the `valid` option for convolution throughout this exercise.**

1. On the Wikipedia page linked above, click on the picture of the PM5544 test pattern and download it.
2. Create a script called `IP1_EX8.m` and import the image.
3. Display the original image using `image`. Title the graph `Original Test Card (NetID)` and save this as `IP1_EX8_Plot0.png`.

   Checkpoint (9/10): Show your TA your displayed image.
4. Convolve the image with a normalized 21×21 box blur kernel.
5. Convert the result (say, `y1`) to unsigned 8 bit integers with the command `uint8(y1)` and display the result with `image`.
6. Title the image `Box Blur Test Card (NetID)` and save it as `IP1_EX8_Plot1.png`.
7. Convolve the image with a 5×5 Gaussian blur approximation kernel.
8. Repeat steps 5 and 6, titling your image `Gaussian Blur Test Card (NetID)` and saving it as `IP1_EX8_Plot1.png`.

   Checkpoint (10/10): Show your TA the two blurred images and discuss the differences between the two blurs.
9. Repeat this process for the following kernels, using the information in Example 12 as well as at the Wikipedia page https://en.wikipedia.org/wiki/Sobel_operator:

   (a) A 3 Sobel operator that will detect edges as colors change from left to right.

   - Title: `Sobel Vertical Edges Test Card (NetID)`

- Save: `IP1_EX8_Plot3.png`

(b) A 3 Sobel operator that will detect edges as colors change from top to bottom.

- Title: `Sobel Horizontal Edges Test Card (NetID)`
- Save: `IP1_EX8_Plot4.png`

10. Display the normalized square root of the sum of the squares of the results for the two Sobel operators with `image`.
11. Title the image `Sobel Edges Test Card (NetID)` and save it as `IP1_EX8_Plot5.png`.

Deliverable (15/16): In your lab report, discuss:

- the differences between the different blurs, and
- what you see in the edge detection images

Deliverable (16/16): Include the following files in your lab report:

- `IP1_EX8.m`
- `IP1_EX8_Plot0.png`
- `IP1_EX8_Plot1.png`
- `IP1_EX8_Plot2.png`
- `IP1_EX8_Plot3.png`
- `IP1_EX8_Plot4.png`
- `IP1_EX8_Plot5.png`

6 Lab Report

This lab assignment is a little different from others in that there is no full lab report with an introduction, discussion, conclusion, etc.

Instead, you should complete the 16 deliverables listed in the body of this document. There is a LaTeX skeleton available on Canvas that has the infrastructure of the lab document already done.

Overall, the deliverables cover the following components:

6.1 Discussion (25 points)

In your lab report, respond to the following prompts:

1. Describe the image output by the code in Exercise 1. How did changing p affect the image? (2 points)
2. Describe the output image from Exercise 2 and how it changed between runs. (2 points)
3. Discuss the differences between the 2-point and 5-point moving averages in Exercise 3. In particular, does one of the smoothed curves look more or less symmetrically-smoothed than the other? Which one, and why do you think that is? (2 points)
4. Discuss the differences between the derivative approximations using 11 points and using 51 points in Exercise 4. Clearly identify the calculations you believe have unacceptable error and provide an explanation for why those errors occur. (4 points)
5. Discuss the differences between the three different blurs used in Exercise 5 (10x10, 2x50, 50x2), as well as any interesting "artifacts" you see in the images. (3 points)
6. Discuss the differences between the `` `same' `` and `` `valid' `` versions of the convolution in Exercise 6. (2 points)
7. Discuss how each process in Exercise 7 changed the image, as well as the difference between the last two images in terms of how well you feel each found the edges. (5 points)
8. Discuss the differences between the different blurs in Exercise 8, as well as what you see in the edge detection images. (5 points)

6.2 Graphics (67 points)

Include the following images in your lab report:

1. Exercise 1
   - IP1_EX1_Plot1.png (2 points)
   - IP1_EX1_Plot2.png (2 points)
2. Exercise 2
   - IP1_EX2_Plot1.png (2 points)
3. Exercise 3
   - IP1_EX3_Plot1.png (2 points)

4. Exercise 4
    - IP1_EX4_Plot11.png (2 points)
    - IP1_EX4_Plot51.png (2 points)
5. Exercise 5
    - IP1_EX5_Plot1.png (3 points)
    - IP1_EX5_Plot2.png (3 points)
    - IP1_EX5_Plot3.png (3 points)
6. Exercise 6
    - IP1_EX6_Plot1.png (3 points)
    - IP1_EX6_Plot2.png (3 points)
    - IP1_EX6_Plot3.png (3 points)
7. Exercise 7
    - IP1_EX7_Plot1.png (3 points)
    - IP1_EX7_Plot2.png (3 points)
    - IP1_EX7_Plot3.png (3 points)
    - IP1_EX7_Plot4.png (3 points)
    - IP1_EX7_Plot5.png (3 points)
8. Exercise 8
    - IP1_EX8_Plot0.png (2 points)
    - IP1_EX8_Plot1.png (4 points)
    - IP1_EX8_Plot2.png (4 points)
    - IP1_EX8_Plot3.png (4 points)
    - IP1_EX8_Plot4.png (4 points)
    - IP1_EX8_Plot5.png (4 points)

6.3 Code (8 points)

Include the code from the following files in your report (1 point each):

- IP1_EX1.m
- IP1_EX2.m
- IP1_EX3.m
- IP1_EX4.m
- IP1_EX5.m
- IP1_EX6.m
- IP1_EX7.m
- IP1_EX8.m

7 Revision History

- October 2024: Added a prelab; made multiple adjustments based on recommendations of Jenny Green (Pratt '25), Eduardo Bortolomiol (Pratt '26), and Adam Davidson.
- Spring 2024: Converted to a standalone document
- Fall 2020: First published version