ENGR 151

Fall 2021

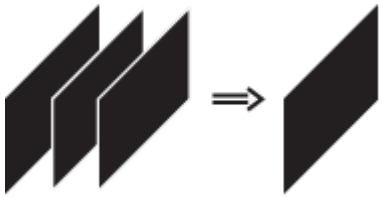
Lab 2: Working with Matrices of Image Data

Images in MATLAB

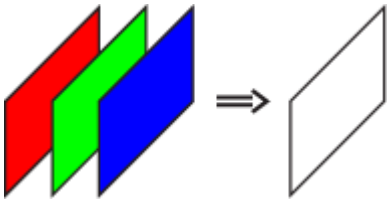
Before you begin, it will help to understand the structure of an image as it is stored in MATLAB. A digital **grayscale** image is stored as a 2-dimensional matrix of pixels. Each pixel has one value representing an intensity of light, stored as an 8-bit number and thus limited to the range from 0 to 255. The value 0 represents BLACK, 255 represents WHITE, and values in between represent various shades of GRAY.

A digital **color** image is often stored as a 3-dimensional matrix. Each pixel has three values representing the intensities of the primary colors of light: RED, GREEN, and BLUE (RGB). Similar to grayscale images, each intensity value ranges from 0 to 255. This time 0 represents zero intensity of a color and 255 represents the full intensity of that color. When calculated, there are $255^3 \approx 16.5$ million possible colors!

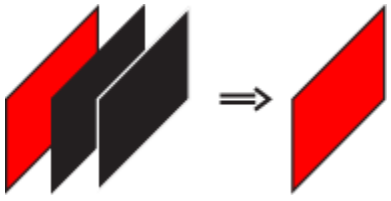
R 0, G 0, B 0



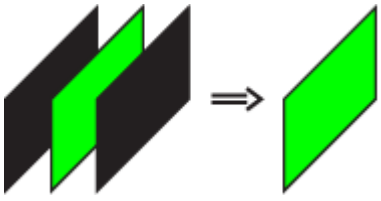
R 255, G 255, B 255



R 255, G 0, B 0

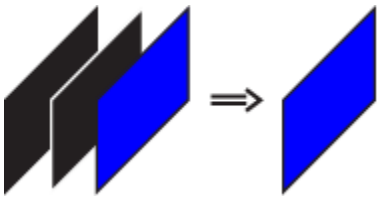


R 0, G 255, B 0



In a color image, a pixel is BLACK if the RGB values are **all** 0 and WHITE if they are **all** 255. The following is an example in code where we set the pixel at location (i,j) in the matrix A to be blue.

```
A(i, j, 1) = 0;   % Set RED intensity of the pixel
A(i, j, 2) = 0;   % Set GREEN intensity of the pixel
A(i, j, 3) = 255; % Set BLUE intensity of the pixel
```



Reading in and Saving Images in MATLAB

When working with images in MATLAB you can follow these steps.

- Read in an image
- Manipulate pixels
- Save or display new image

For example, as in the lab slides:

```
pic_orig = imread('balloons.jpg');      % Read in image
pic_dim = 0.8 .* pic_orig;              % Reduce intensity by 20%
imwrite(pic_dim,'dimmed.jpg');          % Write image to file
imshow(pic_orig);                       % Display image
```

Here are some useful functions when reading, writing, and displaying images in MATLAB:

Function	**Description**	**Example**
`imread(filename)`	Reads the image as a 3D RBG matrix	`imread('my_pic.jpg')`
`imwrite(C, filename)`	Saves the 3D matrix C as a new image with name filename	`imwrite(data,"new_pic.jpg")`
`imshow(C)`	Displays matrix C in a figure	`imshow(arr)`
`figure()`	Opens a new figure for the next plot	

`subplot(rows,cols,position)`	Allows you to plot multiple figures in same window; creates a window with rows*cols spots for figures and plots the next plot in the position-th spot	`subplot(1,3,2);`
		`imshow(my_img);`
		
		Plots my_img in the 2nd position of the figure

`close all`	Closes all open figures	

Exercise 1: Plot Histogram of RGB Image Intensities

An image histogram is a histogram that graphically represents the intensity distribution of an image. It plots the number of pixels for each intensity.

In this exercise, you have to generate an image histogram and practice plotting and editing plot elements such as axes titles, plot title, plot color, and legend.

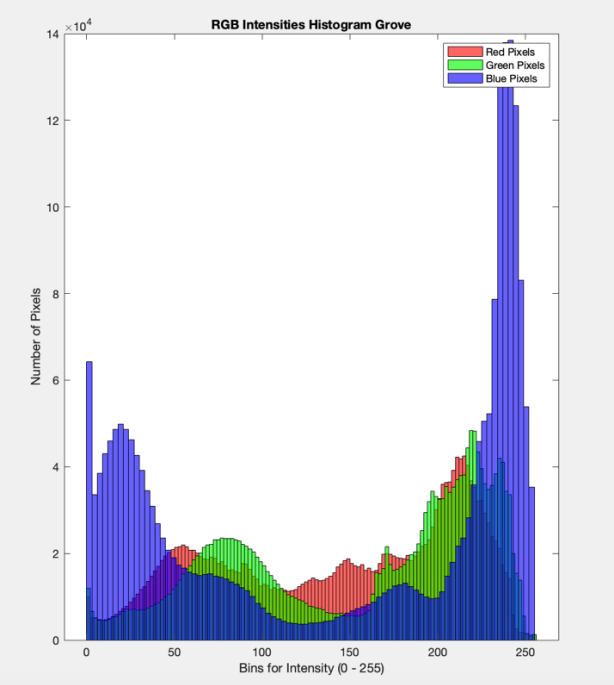
Your job is to write a **function** that reads in an image and displays a histogram of the image, showing the intensities of the three colors (R,G,B). You need to customize histogram so that the distribution of an image's red pixels is represented in red, the distribution of an image's blue pixels is in blue, and green pixels in green. You must also add chart title, axes titles, and a legend. The actual textual content of the legend and titles don't have to follow a specific format, just as long as the content is representative! (Think about how you would format a chart for a PowerPoint presentation!) See example below for what we expect your plot format to be. Save your generated figure as a file named **hist_output.jpg**.

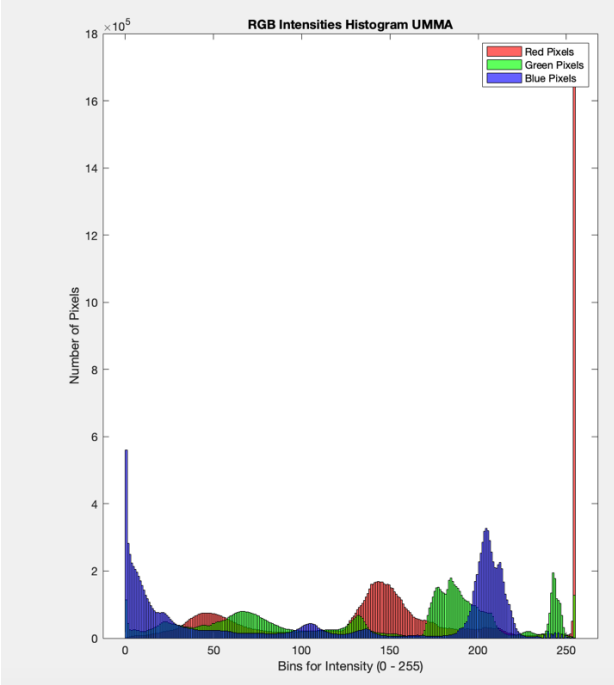
This code must be vectorized - no loops allowed!

Note: You **MUST** use the `histogram()` function introduced in lecture. The distributions of the three pixels for each image **must** be on the same plot. Use the `saveas()` function to save your plot to file.

Example:
The images I'm reading in:


The histogram outputs get:





Coding Guidelines

1. The input to the function is a string with the name of the file. Use that name to read the image file. (i.e. imread(image_name))
2. Vectorize your code, do not use loops!
3. Generate all histograms on one plot.
4. Save red, blue, green histograms to variables hist_red, hist_green, hist_blue (i.e. red_hist = histogram(...)) (Make sure these are exact, they are returned by the function).
5. Remember to add labels for x axes, y axes, title, and legend.
6. Save your image at the end with saveas(gcf,'hist_output.png'). ‘gcf’ tells Matlab to grab the current figure and is then passed as an argument to ‘saveas’ to store it in the specified filename.

Exercise 2: Find the hidden message

In this exercise, you will use color filtering and logical arrays to find a hidden message. The image contains a message which you will have to find by keeping certain colors and washing out others.

Extract the message in the image “color_code_rgb.jpg”. The colors you are to keep are red, green, and blue. Below is an example of what the decoded message would look like. Pixels of non-red, green, or blue letters become WHITE, ie. you have to erase them from the original image!

You do not know the exact pixel value of each color, so you will have to look for relative values. For example, to find red letters, check if the red value of a pixel is greater than 3 times the intensity of the blue value as well as the 3 times the intensity of the green value. Do the same to find the green and blue letters. Notice that the colors that remain in the exercise are either **red, green** or **blue**! That’s because we filtered out we tried to find the **most red/green/blue pixels!**

Your job is to write a **script** that reads in the image and outputs an image where only the letters of the specified colors are visible. If you wish to view both images at the same time, use the **subplot** function.

Again, this code must be vectorized - no loops allowed!

Note: you cannot index a 3D matrix with a 2D matrix. To create a logical matrix of the correct dimension, you may want to use the built-in MATLAB `cat()` function which concatenates (glues) multiple matrices together. Use help `cat()` to learn about how to use it.

Code Submission Guidelines

- Your job is to write a script that reads in the image and outputs an image where only the letters of the specified colors are visible. Save the image of your extracted message as 'color_decoded.jpg'
- For your code submission, make sure you use the sample image 'color_code_rgb.jpg' available on canvas. Save it with the same name in the same folder as your code.
- Again, this program must be vectorized. (no loops allowed!)
- Use 'imshow' to view your image
- Use the variable 'pic_decoded' in your code to store the final decoded image before saving it as a file.

Ideas for a fun exercise - make your own Photoshop effect

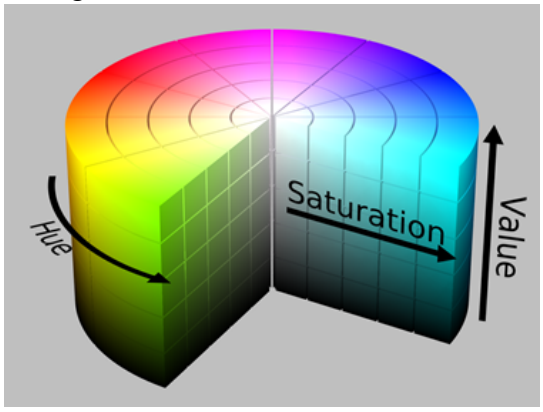
THIS IS NOT GRADED. But, you can do this, and upload your awesome creations to share with your classmates on Piazza!

For this last exercise, we want you to get creative! Play around with your own pictures and invent your own original effects. Make sure your idea is not too simplistic to get enough credit for this exercise. It also doesn't have to be overly complex. As a guideline, you should **either use logical array indexing or explore a new concept**. Below are a few ideas to get the juices flowing.

Option A: Working in the HSV color space

How is this possible??

RGB is just one way to represent colors. Another color space is called HSV (hue, saturation, value). An image in HSV format is also 3D, but the 3 values represent hue/saturation/value of the color instead of the red/green/blue components of a color. The image below gives you a good idea of what effect each of these factors has on the pixel. For example, to make a color black, change its value to 0. To make a color white, change its saturation to 0.

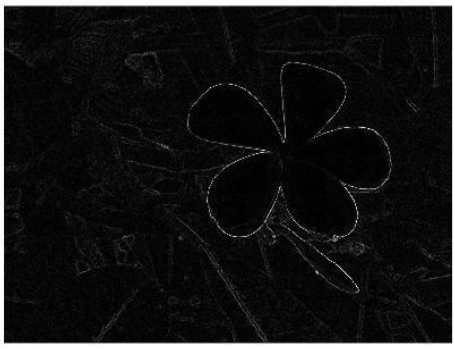


Note: to work in HSV space, you can and should use the built-in function `rgb2hsv`.

Trying playing around with pictures in HSV format to get effects that would be hard to do in RGB format! After manipulating your image, you will ***need to convert back*** using the **`hsv2rgb`** function before using **`imshow`**, which expects color images in RGB format.

Option B: Edge Detection*

*this option is more involved, but the instructions will walk you through the steps of the algorithm



This option is more involved, but is a very useful image processing technique. It is used in many real-world applications. For example, an autonomous car may use edge detection to identify lines on the road or the outline of street signs.

Edge detection sounds complex, but the idea behind it is pretty simple: You want to look for places where the color changes suddenly. Generally in an image, you will find groupings of similar colors within objects. When neighboring pixels are different, we see this as an edge.

To do this, you will use what is called a Sobel filter to detect the change around a pixel. This filtering technique will use a 3 x 3 matrix that calculates the gradient of color change across a particular pixel.

Imagine that the 3 x 3 matrix is overlayed on top of the image array with its center over the pixel being considered as shown below. Multiply each value of the 3 x 3 matrix with the value under it. These 9 values will be used to calculate a new value for the center pixel.

$$A = \begin{bmatrix} 1 & 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 & 10 \\ 11 & 13 & 15 & 17 & 19 \\ 12 & 14 & 16 & 18 & 20 \\ 21 & 23 & 25 & 27 & 29 \end{bmatrix}$$

The 3x3 filter matrices we will use are:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Where G_x finds the change in the x direction and G_y finds the change in the y direction.

Use the equation below to find the total magnitude of change for each pixel. G will be the new pixel value.

$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

For example, consider the matrix A above. A(2,2) has value 4. Let's calculate it's G value:
G_x = 1*A(1,1) + 2*A(2,1) + 1*A(3,1) - 1*A(1,3) - 2*A(2,3) - 1*A(3,3) = -16

G_y = 1*A(1,1) + 2*A(1,2) + 1*A(1,3) - 1*A(3,1) - 2*A(3,2) - 1*A(3,3) = -40

thus, G = sqrt(G_x^2 + G_y^2) = ~43

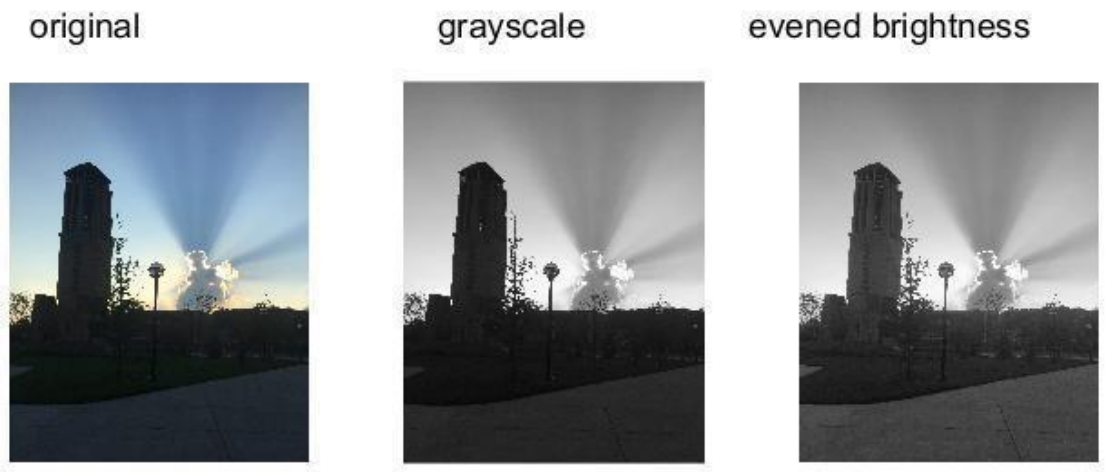
You will calculate this based on a 2D (grayscale) image so you may need to use your grayscale code. Your code may be vectorized but does not have to be.

Since the pixels on the border of the image will not have enough neighbors to do the above calculation, you should only calculate the new value for rows and columns 2 to end-1.

You may use any image you like, but not all images will work as well if they don't have clear edges. Normally when edge detection is done, some pre-processing is done (i.e. blurring, noise reduction, etc.).

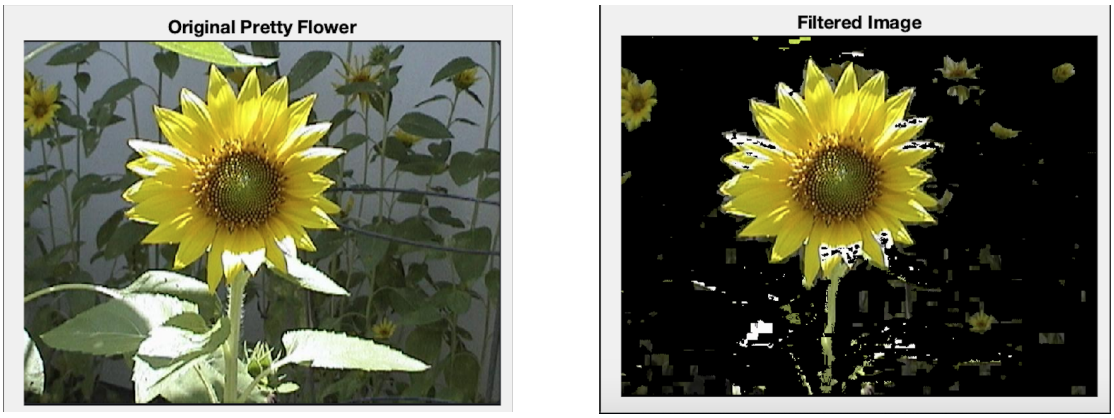
Option C: More Logical Arrays (Selective Brightness Example)

Try using logical arrays in other ways besides how you used them in Exercise 2. In the example image below, notice how part of the image is very dark. The image was altered such that only pixels below a certain threshold get brightened while the light parts of the image stay the same.



Option D: Color Slicing

Color slicing is a technique in Digital Image Processing to locate parts of an image based on regions of color. We'll be using this to our advantage to highlight certains parts of an image like below!



Your task will be to read in an image, convert it into HSV format (this simplifies a lot of filtering as it's easier to visualize and we can simply only adjust the hue parameter to get decent results). Set a range for the HSV parameters (i.e. .2 <= hue <= .8). Using this, create a logical mask for pixels that fall within this range. Then, set all pixels (note we're back in 3D here) to black (zero) or any other constant color value. Last, be creative! Use your own images, filter out different parts of the image, or even better tune the parameters to create a better filter :). Good luck!

Files to turn in

This lab assignment is due before the beginning of your next lab. Please turn in following on Canvas under Lab 2 assignments:

- Exercise 1 code copied and pasted to L2: Exercise 1
- Exercise 2 code copied and pasted to L2: Exercise 2

Any additional image filtering artwork you want to share with us to Piazza!