MTRE 2610 Engineering Algorithms and Visualization – Dr. Kevin McFall

Laboratory – Image Processing

Introduction

The goal of this laboratory exercise is to use become familiar with digital representation of images and using basic image processing techniques to distinguish colors in an image.

Image Pixels

It may not be obvious in the modern age with high resolution cameras, but digital images are composed of discrete picture elements called pixels. Identifying individual pixels is possible with repeated digital zooms or in images such as the one in Figure 1 from Minecraft. The resolution of an image is the number of pixels in each direction of the image, i.e. a 480×640 image contains pixels organized in 480 rows and 640 columns. Figure 2 shows a small portion of a grayscale image with pixels of varying shades of gray and their numerical representation. Pixels with zero value are black and white pixels are assigned the largest value, which is 255 in this example. The bit depth of an image is not related to its resolution but rather the number of different values a pixel can take. The maximum pixel value of 255 in Figure 2 uses 8 bits since $2^8 = 256$. Somewhat counter-intuitively, image rows and columns are measured down and to the right from the top left corner.

Figure 1: Image from Minecraft with its purposely pixelated graphics.

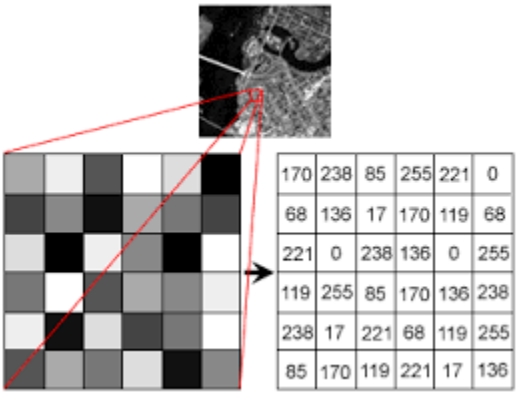

Figure 2: Digital zoom on a grayscale image showing individual pixels and their numerical values.

Several methods exist for representing color images, with the most common considering red/green/blue (RGB) components separately. An RGB image consists of three matrices with the same resolution stacked in layers, one each for strengths of the red, green, and blue components in each pixel. The grayscale equivalent of a color image is created simply by taking the average of the RGB values for each pixel.

In Python, `im = cv2.imread('pic.jpg')` reads the `pic.jpg` file, returning a 3-dimensional Numpy array `im` with the lengths of its first two dimensions indicating the image resolution. The third dimension has a length of 3, which each layer holding a color channel. The `cv2` module provides numerous functions for image processing from the open source library OpenCV. Interestingly, OpenCV images are stored in the order blue/green/red (BGR) rather than RGB. This is important to keep in mind when using OpenCV in combination with other modules such as `matplotlib`. How color channels are encoded in the three layers of an image appear in Table 1.

Layer of image `im`	OpenCV (BGR)	matplotlib (RGB)
`im[:,:,0]`	Blue	Red
`im[:,:,0]`	Green	Green
`im[:,:,0]`	Red	Blue

Table 1: Encoding of color channels in OpenCV and matplotlib.

For example, when an image is read with `cv2.imread` and displayed with `matplotlib.pyplot.imshow`, the red and blue channels are switched as appears in Figure 3 (left) where the folders are actually yellow, green, blue, orange, and purple. Colors appear correctly, Figure 3 (right), after rearranging the color channels to RGB and using `imshow`. The code producing Figure 3 is

```
import cv2
import matplotlib.pyplot as plot
BGR = cv2.imread('folders03.jpg');
RGB = cv2.cvtColor(BGR, cv2.COLOR_BGR2RGB)
plot.figure(1)    # Figure 3 (left)
plot.clf()
plot.imshow(BGR)
plot.pause(0.001)
plot.figure(2)    # Figure 3 (right)
plot.clf()
plot.imshow(RGB)
plot.pause(0.001)
```

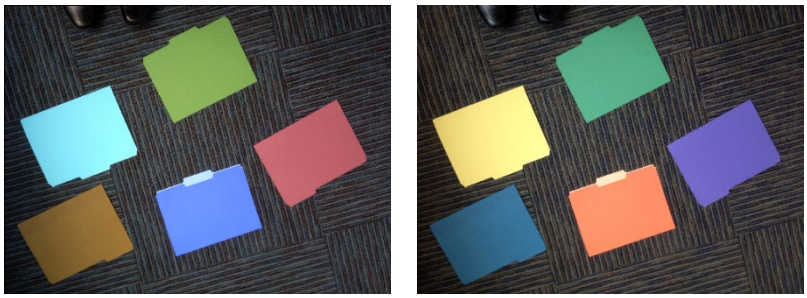

Figure 3: Image where red and blue channels are switched (left) and

Notice that the green folder in Figure 3 does not change much swapping red and blue channels, and that since yellow is a combination of red and green, the actually yellow folder appears cyan (combination of blue and green) when the red and blue channels are switched. The call to `cv2.cvtColor` using the constant `cv2.COLOR_BGR2RGB` converts images from BGR to RGB. Other conversion constants include `cv2.COLORBGR2GRAY` and `cv2.COLORBGR2HSV` for conversion to other color spaces.

Image segmentation

The process of separating pixels from objects and bakgound in an image is called segmentation. This could be identifying road boundaries for a self-driving algorithm or locating the eyes for facial recognition. Segmentation can be accomplished by searching for edges, shapes, colors, textures, etc. and is often a challenging image processing task. Segmentation in this assignment will be accomplished by the user manually clicking on the image to define a region of interest, but automatic segmentation will be covered in the next laboratory exercise. An image can be displayed with the `imshow` command and then `matplotlib.pyplot.ginput(4)` will return a Numpy array with four rows where each contains the x and y pixel indices where in the image the user clicks with the mouse. These four indices are to mark a 4-sided polygon bounding the image region to be considered as indicated by the blue area in Figure 4. The polygon boundary edge line

$$y = mx + b \tag{1}$$

passes through any two consecutive indices such as (x_1,y_1) and (x_2,y_2) where

$$m = \frac{y_2 - y_1}{x_2 - x_1} \text{ and } b = y_1 - mx_1 \tag{2}$$

Note that the y-axis in Figure 4 points down to be consistent with `imshow` axes. The inequality

$$y_c > mx_c + b \tag{3}$$

holds for point (x_c,y_c) as long it lies on that side of the line defined by m and b whereas

$$y_c < mx_c + b \tag{4}$$

would be true if (x_c,y_c) fell on the other side, i.e. the red region in Figure 4.

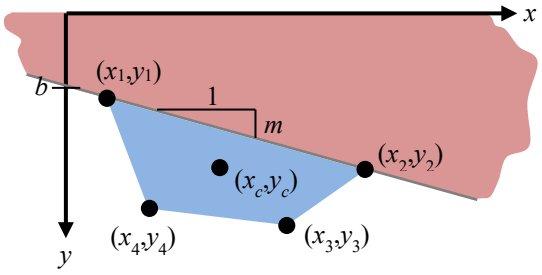

Figure 4: Geometry of a line defined by two points in relation to other points in the domain.

To identify the blue area in Figure 4 bounded by the four clicked points, first create a `numpy.meshgrid` of `x` and `y` integer values from 0 up to one less than the resolution of the image, i.e. the `shape` of `x` and `y` should match the image size. The expression `y<m*x+b` returns a Boolean Numpy array where pixels in the red area in Figure 4 are true. All the RGB intensities at these locations should be set to 0 to eliminate pixels outside the blue bounded area. Repeat this process for the other three lines comprising the polygon to remove everything but the blue region. The choice of using `>` or `<` in the inequality will depend on which side of the line the center point lies. The appropriate choice is the opposite of whether y_c is greater or less than $mx_c + b$. The area center point x_c and y_c is located by taking the average of the four x_i and y_i points, respectively.

Automatically identifying colors

The task in this exercise is for the user to choose one of the colored file folders in Figure 5 by clicking on its corners in the image, and for the program to return the folder color, either yellow, blue, green, orange, or purple. Once the image is segmented, extract all nonzero entries for each of the RGB components and display a `histo`gram of their values in a single figure with three `matplotlib.pyplot.subplots` as in Figure 6 for the green folder. Histograms plot the frequency a given value occurs in series of data. For an image, values will range from 0 to 255 for an 8 bit-depth. The green component is the strongest in Figure 6 with an average value around 175 compared with the red and blue averages which are both less than 125.

The `matplotlib.pyplot.hist` command accepts an array of values, whereas the pixel values are in matrix form where only the non-zero values are of interest. One option is to use logical indexing to find the values in the matrix meeting a particular criteria. For example, the code

```
import numpy as np
x = np.array([[1, 2, 3],
              [4, 5, 4],
              [3, 2, 1]])
print(x[x>3])
```

displays `[4 5 4]`, which is an array of all the values in `x` greater than 3. Note that `hist` has an optional input parameter `color` accepting a string with the color with which to plot, i.e. `'r'`, `'g'`, or `'b'` in order to produce color like in

Figure 6. Adjust the horizontal axis limits with `matplotlib.pyplot.axis` to format the histograms in the range of 0 to 250 as in Figure 6 to allow easier comparison of RGB components.

Calculate the average RGB values for the segmented area and use them to create an `elif` structure to classify the color of the selected area. Ensure that all folders are correctly classified for all sample images available on D2L.

Figure 5: Sample image with file folders (left) whose colors are to be automatically identified when segmented manually (right) by the user clicking on the four corners of a folder.

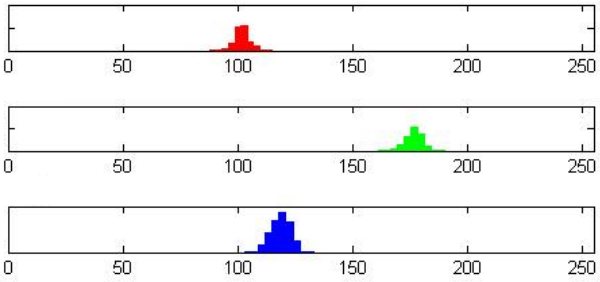

Figure 6: Histogram of RGB values when selecting the green folder in Figure 4.

Laboratory exercise procedure

Rather than attempting to solve the entire problem in one pass, start small and incrementally add complexity until the problem is solved. Before eliminating pixels on all four sides of the segmented folder and concerning on which side the polygon center lies, black out pixels on one side of the line defined by two points identified by the user. Complete the following code fragment to accomplish this task.

```
BGR = cv2.imread(    )
RGB =
R = RGB[   ,   ,   ]
G = RGB[   ,   ,   ]
B = RGB[   ,   ,   ]
plot.figure(1)
plot.clf()
plot.imshow(RGB)
p = plot.ginput(2)
m =
b =
p = np.meshgrid(   ,   )
R[y > m*x+b] = 0
G[          ] =
B[          ] =
imNew[   ,   ,   ] = R
imNew[   ,   ,   ] = G
imNew[   ,   ,   ] = B
plot.figure(2)
plot.clf()
plot.imshow(imNew)
```

Now modify the code to request four clicks from the user, which are assumed to be adjacent corners defining a four-sided polygon. Compute the center of the polygon and use it to determine which side to keep for each of the four lines defined by each side of the polygon. Display the segmented image keeping only the region within the mouse clicks. Finally, complete the histograms and the decision structure to return the color of the selected folder.

Grading rubric

1. 20 points: Black out all pixels on one side of a line defined by clicking twice in an image
2. 40 points: Display the fully segmented image
3. 20 points: Produce the R, G, and B histograms for the segmented image
4. 20 points: Demonstrate correct classification for all folders