CS282: Programming Assignment 1

Richard Guinto <richard.guinto@gmail.com>

February 27, 2018

Abstract

This document is for the fulfillment of the requirements for CS282 Programming Assignment 1 under Prof. Pros Naval, Department of Computer Science, University of the Philippines (Diliman)

1 OpenCV Exercise

Problem Statement

Display a given image using OpenCV. When the user clicks any part of the image, a label is displayed at the mouse pointer showing the *x,y* coordinate as well as the *Blue, Green, and Red* value of the pointed pixel.

Program implementation

The program checks if filename of the image is given using the command-line parameter. If not, then a default image, *RGB.png*, will be loaded by the program.

1.1 Loading an image

The image is loaded using openCV *imread()* function. `IMREAD_COLOR` is set as a flag to ensure that image is converted into 3 channel BGR color format.

The image is displayed in a window using openCV *imshow()* function. The progam then waits for the user to press a key using *waitKey()* function. If the *q* key is pressed, the program closes the window and exit.

1.2 Capturing mouse events

openCV *setMouseCallback()* function can be used to capture mouse events. The window name and callback function are passed as parameters.

When a mouse event occurs, the callback function, `display_pixel()` is called. Two mouse events were monitored by the callback function:

- `EVENT_LBUTTONDOWN` Indicates that the user clicked on the image

  The flag *showPixel* is set to True to start showing the pointer coordinates and pixel BGR value

- `EVENT_MOUSEMOVE` Indicates movement of the mouse pointer on top of the image

  If the *showPixel* flag is set to True, the program will extract the current mouse pointer location as well as the BGR value of the image pixel pointed by the mouse pointer.

  The coordinates and the pixel value are then displayed using the openCV *putText()* function. The text color is chosen such that it is inverse of the current pixel color.

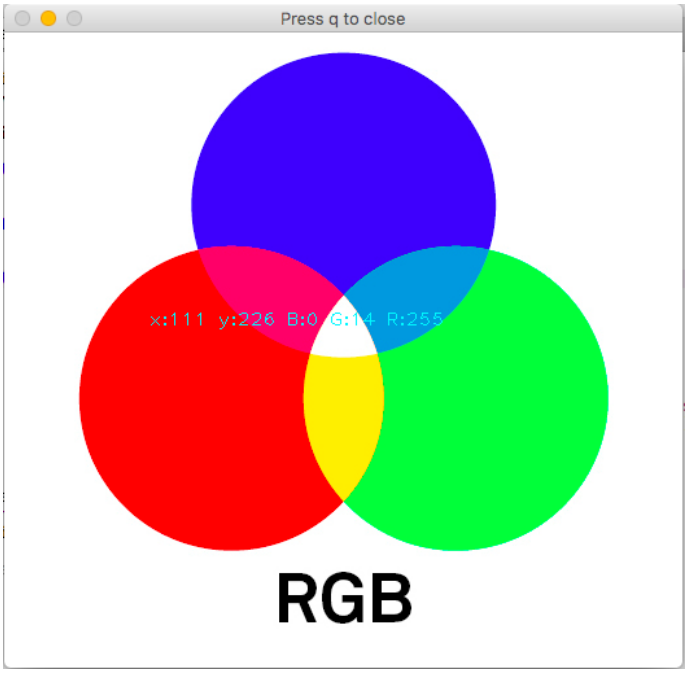

Figure 1: Sample image showing mouse location and pixel color.

2 Canny Edge Detection

Canny Edge Detector is one of the widely used technique for edge detection among images. It is developed by John F. Canny in 1986. There are several steps involved in this edge detection algorithm as described by [2]:

1. Noise Reduction
2. Finding intensity gradient of the image
3. Non-maximum suppression
4. Linking and Hysteresis thresholding

Problem Statement

Perform Canny Edge Detection algorithm into the output of your laptop camera, showing three stages of the video processing:

1. Original image (color)
2. Gray scale image
3. Canny Edge image

2.1 Capturing camera frames

To capture a video, you need to create an openCV *VideoCapture* object [2]. The device index 0 is passed as the parameter to the *VideoCapture* function to use the first camera available. The *cap.get(propId)* function can be used to get the camera property, such as width($propId = 3$) and height($propId = 4$). To scale the frame size, simply call the *cap.set(propId, value)* function passing the property ID and the desired new value. For example, *width = cap.get(3)* and *cap.set(3, width/4)* will reduce the frame width by $1/4$ of the original frame width.

2.2 Converting to Grayscale

An image can be converted into gray scale by using the openCV *cvtColor()* function, passing the image and `cv2.COLOR_BGR2GRAY` as the first and second parameter respectively.

Figure 2: Concatenated images from Original video capture, converted grayscale image, and Canny Edge image.

2.3 OpenCV Canny function

Performing Canny Edge detection algorithm in an image can be done by using a single function call to openCV Canny() function. This function accepts 3 required parameters:

- *image*. A 8-bit image
- *min threshold*. The minimum threshold for the hysteresis procedure.
- *max threshold*. The maximum threshold for the hysteresis procedure.

In order to save time in manually tuning the lower and upper thresholds, [3] suggested a simple trick by relying on the median of the image pixels. The *min* and *max* thresholds are then computed using +/- percentage of a *sigma* value. A lower value of *sigma* gives tighter threshold whereas a larger value gives a wider threshold. In practice, a *sigma* value of 0.33 tends to give good results on most of the datasets.

2.4 Concatenating video frames

To be able to concatenate three images horizontally, numpy *hstack()* function can be used. This function accepts a list of images to be concatenated.

In order to concatenate the gray scale image with the 3-channel colored image, the gray scale image can be converted into 3-channel using the openCV *cvtColor()* function with `cv2.COLOR_GRAY2BGR` as the second parameter. The same conversion can be done with the output of the *Canny()* function which is also a 8-bit image.

Lastly, a text label can be embedded into each image by using the openCV *putText()* function. Sample output is shown in figure 2

3 Image Enhancements

Problem Statement

Using spatial domain operators, write openCV codes that perform image enhancement on the following images:

(a) dental.jpg

(b) cells27.jpg

(c) butterfly.jpg

(d) momandkids.jpg

3.1 dental.jpg

3.1.1 Observation

The input image shows some level of darkness/brightness, so we will examine the image carefully by experimenting on different results based on powerlaw transformation

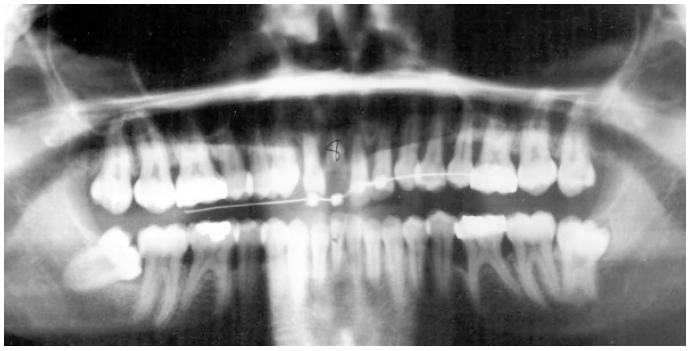

Figure 3: Dental Original Image

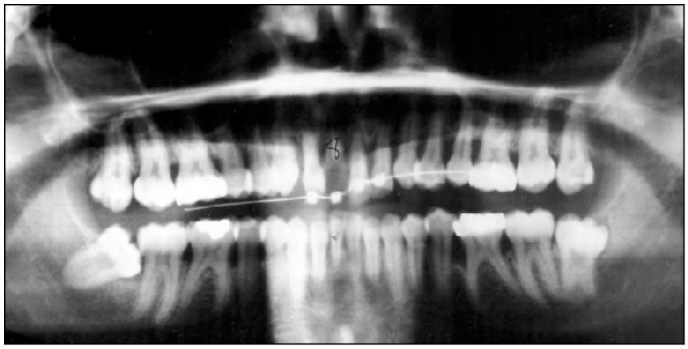

Figure 4: Gamma = 0.6

3.1.2 Power Law Transformation

The power law transformation has the equation of the form: $s = cr^{\gamma}$ where c and γ are positive constants Given the application of different values of Gamma into the sample dental image 3, the application of Gamma = 2.0 in 5 shows the best result as it shows brighter image of the tooth root.

3.2 cells27.jpg

3.2.1 Observation

The image in figure 7 is photographic image of a cell and is predominantly dark.

3.2.2 Enhancement

We need to enhance the white/gray details embedded on dark regions of the image by performing a negative transformation as shown in figure 8

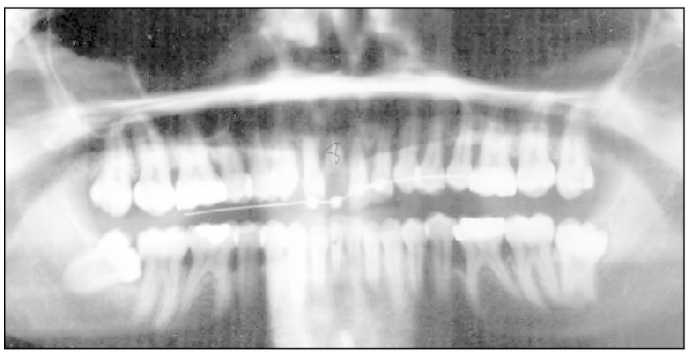

Figure 5: Gamma = 2.0

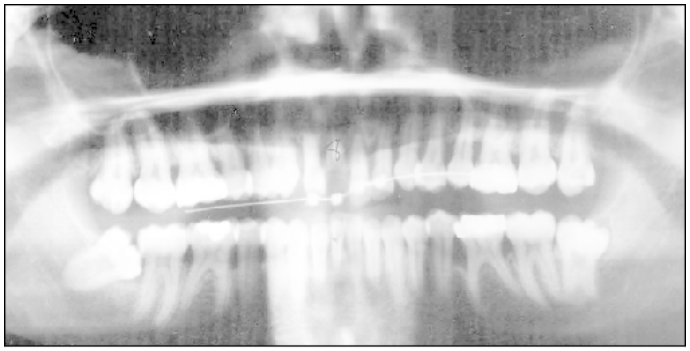

Figure 6: Gamma = 2.5

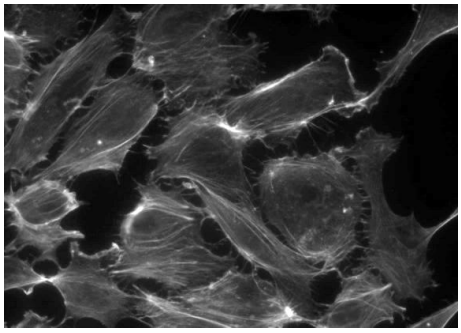

Figure 7: Input Image - cells27.jpg

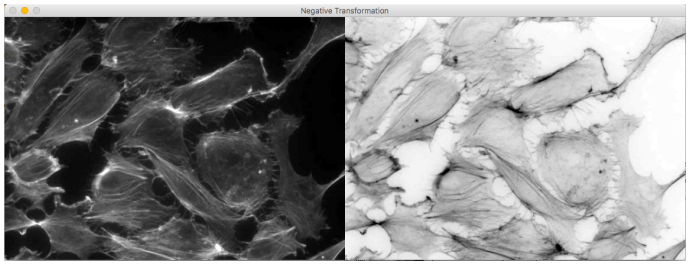

Figure 8: (a) Input Image (b) Negative Transform

Figure 9: Input Image - butterfly.jpg

Figure 10: (a) Original (b) with Median Filter (c) with Histogram Equalization

3.3 butterfly.jpg

3.3.1 Observation

The input image as shown in (a) of figure 9 has two visible problems:

1. salt and pepper noise
2. low contrast

3.3.2 Median Filter

The white and black random pixels can be removed by applying a simple median filter. This can be implemented using openCV *medianBlur()* function. 3x3 kernel shows to be a good value for our sample image as shown in (b) of Figure 10

3.3.3 Histogram Equalization

Upon checking the histogram of the input image, figure 11 shows that majority of the gray level of the image is concentrated at a narrow range. Thus, to be able to enhance the result of the median filter, the gray level can be uniformly distributed using Histogram Equalization. OpenCV supports this feature using the *equalizeHist()* function.

3.4 momandkids.jpg

3.4.1 Observation

The input image contains a lot of noise as shown in figure XXX

3.4.2 Median Filter

The first step to remove the unwanted noise from the input image is to apply a Median Filter. Using a kernel size of 3x3 would result to the removal of majority of the noise pixels, but not all. If kernel size of 5x5 is used instead, the unwanted noise pixels were all removed, but the image suffered with too much blur, as shown in (b) of figure 14

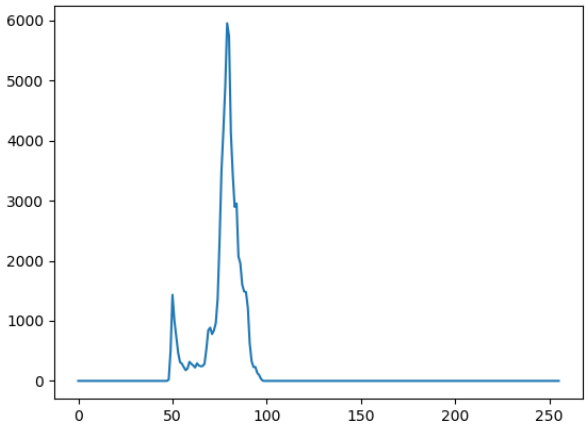

Figure 11: Histogram of the enhanced image in Figure 10 - b

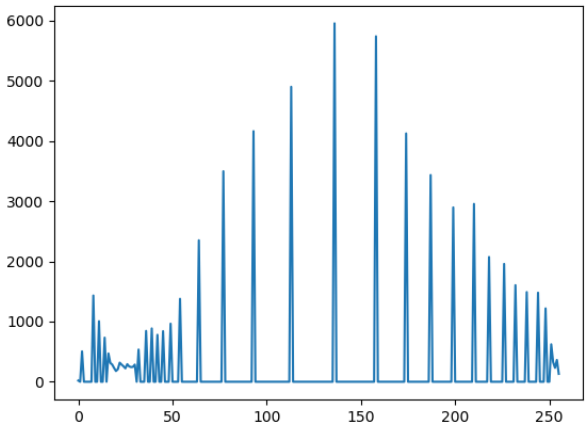

Figure 12: Histogram of the final output image as shown in Figure 10 - c

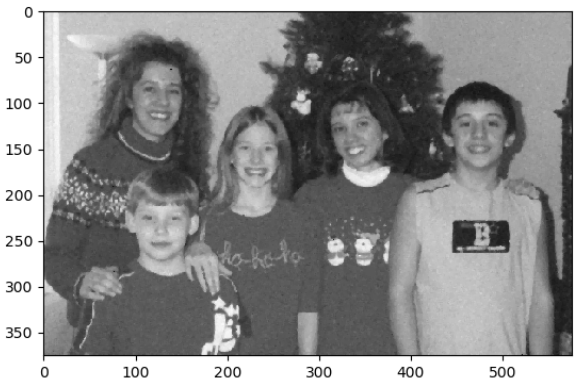

Figure 13: Result of applying Median Filter 3x3

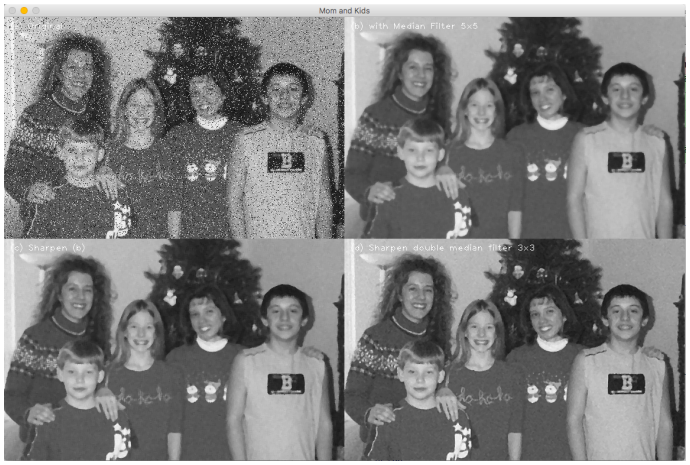

Figure 14: (a) Original Image (b) with Median Filter 5x5
(c) Sharpen (b) using unmasking with Gaussian Blur (d) Sharpen using two-stage median filter and Gaussian Blur unmask

3.4.3 Unmask with Gaussian Blur

I've done two experiments:

1. Apply Median Filter 5x5, compute Gaussian Blur with 5x5 kernel, and then perform image unsharp masking (sharpening) with 1.5 to -0.5 ratio between original and blur mask. See Figure 14b and c

2. Apply Median Filter 3x3 two times, compute Gaussian Blur with 5x5 kernel, and then perform image unsharp masking (sharpening) with 1.5 to -0.5 ratio between original and blur mask. See Figure 14d

4 Unsharp Masking in Spatial Domain

Unsharp masking consists of blurring the original image and subtracting the blurred image from the original. We call the original image as the "mask". The mask (or fraction of it) is then added to the original.

Problem Statement

Write openCV code that will perform unsharp masking on the given image *building.tif* and compare the filtered image with the original.

4.1 The Mask

The mask is computed as the original image minus the blur version of the image. The blur filter can be implemented using openCV *GaussianBlur()* function. In this experiment, I used Gaussian Blur to get the blur version of the image using 5x5 kernel and sigma value (kernel standard deviation) of 3. Figure 15 shows the blur version and the computed mask of the original image.

4.2 Weighted merging of original image and the blur image

By controlling the percentage of the original pixels as compared to the blur pixels, the result will have different sharpness levels. In this experiment, I presented different ratio of the original and blur, with corresponding output as shown in Figure 16. In openCV, the function *addWeighted()* performs this weighted blending.

Figure 15: (a) Original Image (b) Gaussian Blur (c) Mask

Figure 16: Result of applying different weights for original and blur image pixels

4.3 Observation

You will notice that as you increase the sharpness, as in Figure 16c, vertical lines of the rightmost wall of the building becomes more visible. At first, it can be misinterpreted as the "ringing effect" of low pass filtering, but since we used Gaussian Filter, this is not considered as the artifact of the process. In fact, the other wall of the building doesn't exhibit the vertical lines, which proved that the original image indeed has unnoticeable vertical lines only at the rightmost wall.

5 Translation and Rotation Properties of 2D Fourier Transform

Fourier Transform is used to analyze the frequency characteristics of various filters. For images, 2D Discrete Fourier Transform (DFT) is used to find the frequency domain.

5.1 OpenCV and Numpy Functions

To be able to perform DFT into our input image, we can use openCV *dft()* function. This function accepts two parameters:

- image. The input image
- flags. We will use `cv2.DFT_COMPLEX_OUTPUT` as flag to get Real and Imaginary part.

The result of the dft() function has the zero frequency (DC component) at the top left corner. To bring it into the center, you need to shift the result into N/2 in both directions. This is simply done by the numpy function *np.fft.fftshift()*

The real and imaginary components can be converted into magnitude and angle using the openCV *cartToPolar()* function, passing the 2D real numbers and 2D imaginary numbers as parameters.

The magnitude spectrum is then derived from the magnitude component using the numpy *log()* function.

More details can be found from [1]

5.2 Experiment Results

Using the procedures described in the section above, the magnitude spectrum remains the same for cameraman 1 to 3 but differs in phase. Note that cameraman 1 to 3 are just translation operation. Cameraman 4 is a 90 degree rotation of cameraman 1, and you will see from the resulting magnitude

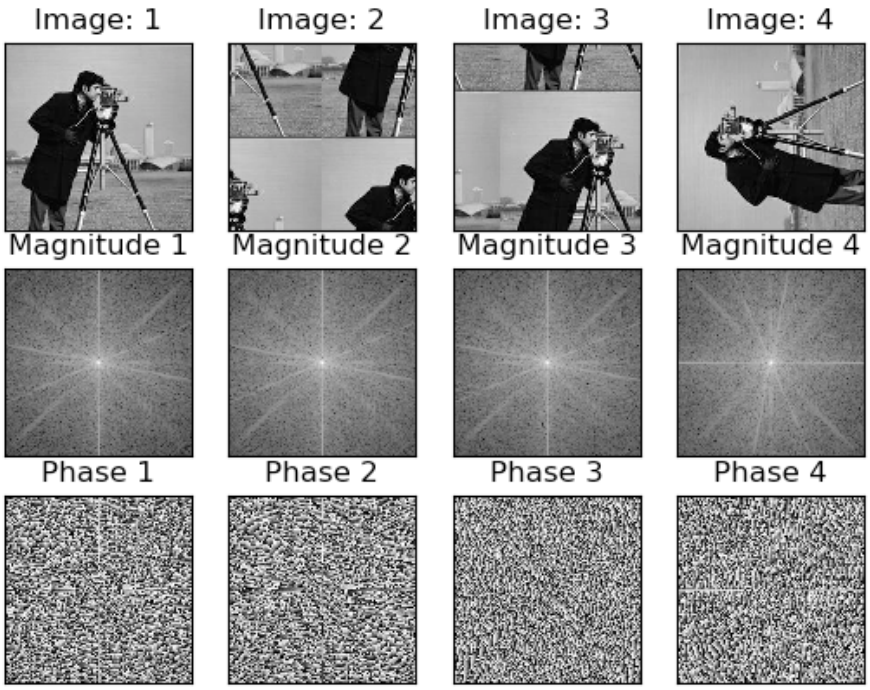

Figure 17: Result of applying Fourier Transformation of different images

spectrum that the magnitude also rotated by the same degree. See figure 17 for the input image, output magnitude and phase.

References

[1] *OpenCV - Fourier Transform.*

[2] *OpenCV - Getting started with Videos.*

[3] Adrian Rosebrock. Zero-parameter, automatic canny edge detection with python and opencv. *https://www.pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/*, 2015.