University of Colorado at Boulder ECEN 4532

Instructor: Mike Perkins

An Introduction to Image Processing

Lab 2
Lab report due on February 11, 2019

This is a Python only lab. Each student must turn in their own lab report and programs.

1 Introduction

This lab will introduce the field of image processing. We will write from scratch a number of commonly used basic routines. Although extensive image processing libraries exist, it is good to implement a few functions yourself in order to appreciate what is happening at the pixel level. It is also good python and numpy practice.

1.1 Background

A common uncompressed image format is BMP. Digital images stored in this format begin with a short header that is followed by a 2-D array of sample values, commonly called "pixels" or "pels". The images we will work with have three 8-bit numbers associated with each pixel: an R, G, and a B value. The diagram below illustrates an (R,G,B) pixel-formatted image.

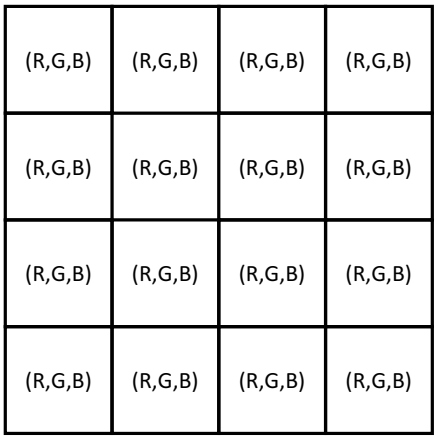

Figure 1: A 4x4 Image with RGB Pixels

The 3 chroma values determine how much of each primary color is present at each pixel location. A wide-range of colors can be represented using 8 bits per chroma channel (2^{24}).

Actually, however, we will do most of our work using monochrome (i.e. grayscale or black/white) images. These images will have 8 bits per pixel with 0 representing black and 255 representing bright white. The monochrome value associated with each pixel can be determined by a dot product with its RGB values as follows:

$$Y = (0.299, 0.587, 0.114) \cdot (R, G, B) \tag{1}$$

The BMP file header encodes the number of rows and columns in the picture along with other information.

There are, of course, many other ways to represent images, including jpeg, ppm, png, and the like. Some of these compress the raw "bmp" pixel data, introducing some distortion in the process. The python library `Pillow` is able to input and output images in a wide variety of formats. Conceptually, it can be thought of as converting images into the BMP format when `open()` is called.

An important concept in imaging is resolution. Assume you have a test image pattern comprising vertical lines of identical width alternating between pure black and pure white. The number of lines in the pattern can be expressed in lines per meter (both the black and white lines are counted). If the imaging system in question is focused on this test pattern at some fixed distance, x, and the vertical lines can be distinguished in the imaging system's output, then, at the distance x, the resolution of the imaging system exceeds the test pattern's lines per meter. Now, without changing any other parameter in the experiment, if we increase the lines per meter in the test pattern to just the point where they can no longer be distinguished in the image, then we have measured the resolution of the system in lines per meter at a distance of x. This can, of course, be rephrased in other ways such as lines per unit angle. The resolution depends on multiple factors including the optical lens, the imaging sensor, the display device, and the supporting electronics.

Importantly, sub-sampling an image reduces its resolution.

2 Converting Color Images to Black and White

A large number of colors can be generated by combing several primary colors, such as Red, Green, and Blue. One common approach is to use one byte for each color component (24-bit color) where each byte assumes a value between 0 and 255 to indicate the intensity of that component. In many image processing tasks, color is

not needed, and a black/white image will suffice. How is a color image converted to black and white? More generally, as many different color spaces are possible depending on the assumptions made about the exact spectral nature of the three primaries, how can one convert from one color space to another?

Normally, it is possible to convert from one space to another using a matrix multiplication. For example, the conversion from RGB to YUV (Y represents luminosity, or black/white intensity) is given by

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} .299 & .587 & .114 \\ -.14713 & -.28886 & .436 \\ .615 & -.51499 & -.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2}$$

The U and V signals are color difference signals that are common in TV applications. Unlike Y, they can be either positive or negative. Y will range from 0 to 255.

For this lab you will be processing black/white "grayscale" images. We will use the python library `pillow for input/output`. The code snippet below shows how to use `pillow` to input and output an image, and how to convert a `pillow` image to a numpy array for processing.

```
#!/usr/local/bin/python3
import numpy as np
from PIL import Image

pil_image_in = Image.open( 'test_img_in.jpg' )
image_in = np.asarray( pil_image_in, np.uint8 )
print(f"shape of image = {image_in.shape}")
rows = image_in.shape[0]
cols = image_in.shape[1]
print(f"rows = {rows}; cols = {cols}")

# modify the image here as appropriate

# Output the image to a file
Image.fromarray(image_in).save('test_img_out.jpg')

# view the image
Image.fromarray(image_in).show()
```

Assignment

1. Using the python `pillow` library for image input and output, write a program that inputs a color image (e.g. in jpg, png, or bmp format) and converts it to a black/white greyscale image. Make sure that you check for values that are too small (less than 0) or too large (greater than 255) after computing Y, and clip values to the range [0, 255]. This program can be written without a `for()` loop, and you should do so if you can. You might find the functions `np.reshape()` and `np.dot()` useful. Convert the two images `test01.jpg` and `test02.jpg` to grayscale.

3 Contrast Enhancement

Sometimes a picture displays a low dynamic range and appears low contrast when viewed. A simple contrast enhancement process based on *histogram equalization* will sometimes provide a dramatic improvement. The images below provide an example.

Figure 2: Low Contrast Image

Figure 3: Contrast Enhanced Image

Now that you have implemented your own program to convert RGB to YUV, for the rest of this lab you may use the `Pillow` library to convert images to grayscale format. The following code snippet shows how this is done:

```
pilim = Image.open( sys.argv[1] )
pilim = pilim.convert('L')  # convert to monochrome
imdat = np.asarray( pilim, np.float ) # put PIL image into a numpy array of floats
```

Assignment

2. Write a program to compute and graph the histogram of a grayscale image. This program can be written with a `for()` loop that loops over the intensity values 0 to 255, and not over each pixel.
3. Write a program to perform "histogram equalization" on a grayscale image. Use the technique discussed in class. Again, this program does not require a `for()` loop over each pixel
4. Process two of the three test images `lc01.jpg, lc02.jpg` and `lc03.jpg` (convert to grayscale as required). Show their histograms before and after equalization, as well as the processed images, in your report. Do the histograms appear uniform (why or why not)? Are you pleased with the performance of this technique? What effect does it have on images that aren't low contrast? (feel free to use an image of your choice)
5. Could this technique be used on color images? How?

4 Edge Detection

Edge detection is an important step in many computer vision applications. A commonly used, and intuitively appealing, edge detector is the Sobel edge detector discussed in class. The pictures below show an image before and after edge detection.

Note that there is always some ambiguity about what to do at the edges of images when the algorithm being implemented uses pixels that are "off the map". You may skip the first and last rows and first and last columns in this lab to avoid having to write special code to deal with image edges. (Set them to black or leave them unchanged at your discretion.)

Figure 4: Raw image

Figure 5: After edge detection

Assignment

6. Write a program to implement the Sobel edge detection algorithm on grayscale images and test it on the images `test01.jpg` and `test02.jpg`. Be smart about how you use `for()` loops. It isn't necessary to have a pair of nested loops that process each pixel separately. Experiment with different threshold values, and include results for a threshold value that you like in your report.
7. Are there situations where you would get different results (find different edges) if you processed the R/G/B image planes separately, than you find by processing the grayscale image? Justify your answer (you don't need to write code or show a result, but think it through).

5 Image Resizing

It is often necessary to make images either larger or smaller. There are better and worse ways to do this from a signal processing perspective. In this part of the lab we'll use a simple, but arguably sub-optimal, approach that is often good enough in practice. In particular, for downsampling an image we will use averaging, while for upsampling we will use bi-linear interpolation.

Assume we have an image with R rows and C columns where R and C are both divisible by N, the downsampling factor. When downsampling by N, we create a

new image of size R/N and C/N. To do this, we divide the original image into non-overlapping blocks of size N-by-N, and average the pixel values from each block to obtain a pixel value for the sub-sampled image.

To upsample by N, we use bi-linear interpolation as described in class. The code below shows how bi-linear interpolation can be implemented using `scipy.interpolate`.

```
#!/usr/local/bin/python3
from scipy.interpolate import RectBivariateSpline
import numpy as np

# Example of bi-linear interpolation for
# upsampling by 2.  Below, x and y define the
# 4 points of the grid that are used as
# the basis for the interpolation.
# Geometrically, the 4 points are arranged
# as follows (x gives the "row" and y the "col")
#        [1,5], [1,6]
#        [2,5], [2,6]
x = np.asarray([1,2], np.int)
y = np.asarray([5,6], np.int)

# z holds the image intensities at the 4 points above.
# Geometrically the intensities are arranged as follows
#          50,  100
#          200, 300
z = np.asarray([[50, 100],[200,300]], np.float)

# Generate the interplation object.  Note how
# y and x are reversed in the calling order from
# what you might have expected
interp_spline = RectBivariateSpline(y, x, z, kx=1, ky=1)

# Now use the interpolation object to bi-linearly
# interpolate onto a finer grid (step size 0.5)
x1 = np.linspace(1, 2, 3)
y1 = np.linspace(5, 6, 3)
ivals = interp_spline(y1, x1)
print(f"interpolated values =\n {ivals}")
```

Assignment

8. Write a program to downsample grayscale images by a factor of N. This can be done with a nested pair of `for()` loops that loop over the pixels of the downsampled image. You might fund the function `np.average()` useful
9. Write a program to upsample grayscale images by a factor of N. This, too, can be done using a nested pair of `for()` loops that loop over the pixels of the smaller image. You will want to use the function `RectBivariateSpline()` from `scipy.interpolate` as shown above. (When downsampling an image, and then upsampling its downsampled version, it is acceptable for the upsampled version to have $N-1$ fewer rows and $N-1$ fewer columns. You can leave black pixels at the edges to maintain the overall image size.)
10. Finally, write a program to compute the absolute value of the difference between two grayscale images, and output/display the result (no `for() loop needed`. You should add a factor to multiply the difference (to make it more visible), and be sure to clip values greater than 255 to 255. Now, using your downsample/upsample programs, process the image `test01.jpg` as follows: first, downsample and upsample by a factor of 2. Use your difference program to look at the difference. Next, downsample and upsample by a factor of 4, and use your difference program to look at the difference. What do you observe?