Lab 2: Introduction to Image Processing

Shane Kirkley

February 11, 2019

Table of Contents

1. Introduction

The processing of digital images is important to many fields of study, including photo enhancement software and image detection. In this lab, we will explore several features of digital images, and apply some manipulation techniques to them. First we will examine how to convert a color image to grayscale for ease of computation. We will plot the histogram of an image, and examine how to enhance the contrast of an image. Edge detection will be implemented, and finally the effects of resizing an image will be examined.

2. Background

A digital image can be represented as a two-dimensional array of intensity values. For a 24-bit color JPEG image, the intensity values are 8-bits for red, green, and blue. Thus each intensity value ranges from 0 to 255. A color digital image features red, green and blue (RGB) intensity values for each pixel, while a grayscale image has only a black/white intensity for each pixel. The YUV color space uses one luminance component (Y) and two chrominance components (U and V) to encode color information for an image. We can convert an RGB image to YUV by using the formula

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.14713 & -0.28886 & 0.4366 \\ 0.615 & -0.51499 & -0.10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}.$$

For this lab, we will mainly be concerned with the Y component, which alone makes up a grayscale image. We'll see that information from each color plane is represented in grayscale, and computation for features such as edge detection are faster with a grayscale image.

3. Part I: Converting Color Images to Black and White

We can convert an RGB image to grayscale by using the luminance (Y) part of the RGB to YUV color conversion. Specifically, we find the Y intensity value from RGB intensities as

$$Y = \begin{bmatrix} R \\ G \\ B \end{bmatrix} \cdot \begin{bmatrix} .299 & .587 & .114 \end{bmatrix}$$

.

We note the decimal values in this equation sum to 1, essentially designating a percentage of the luminance value for each color. These values are determined by the natural color intensity perception of human vision. Figure 1 shows the Python function that converts a numpy array of RGB image data to grayscale. The Python script `grayscale.py` is used to convert an RGB image to grayscale, and is shown in the appendix. The results of running this script on images are shown in Figures 2 and 3. For the remainder of the lab, we will use built in Pillow function `convert` to handle RGB to grayscale conversion.

```
def grayscale(img):
    return np.clip(np.dot(img[:,:,:3], [.299, .587, .114]), 0, 255)
```

Figure 1 - Grayscale function converts array of RGB image data to grayscale.

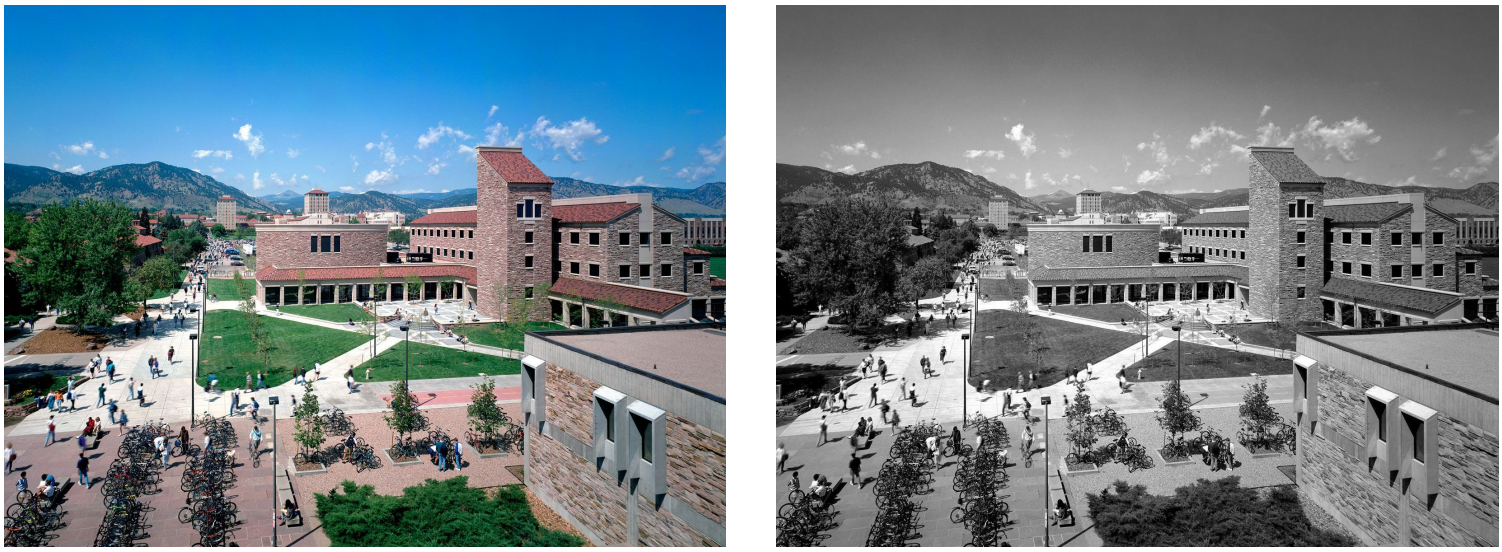


Figure 2 - Grayscale conversion of test01.jpg

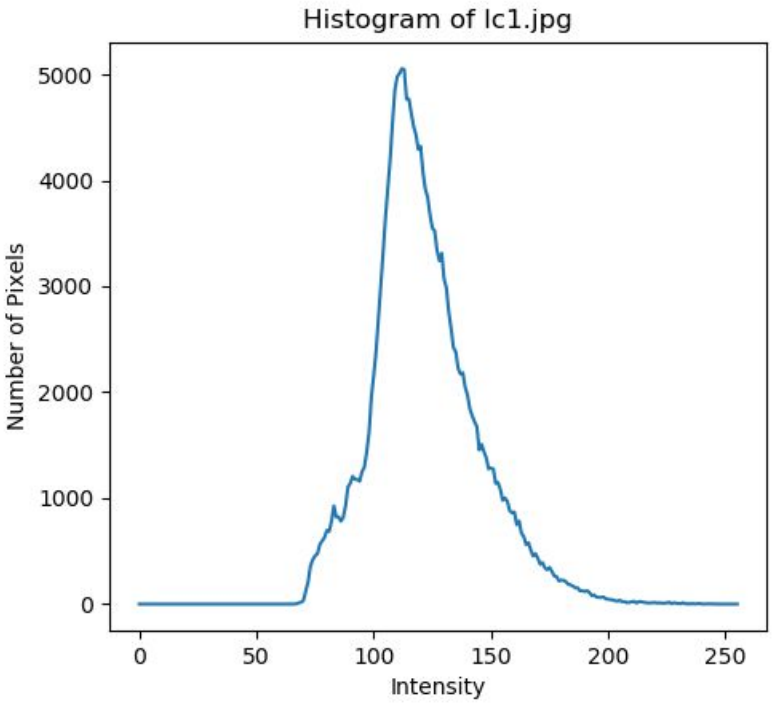
Figure 2 - Grayscale conversion of test02.jpg

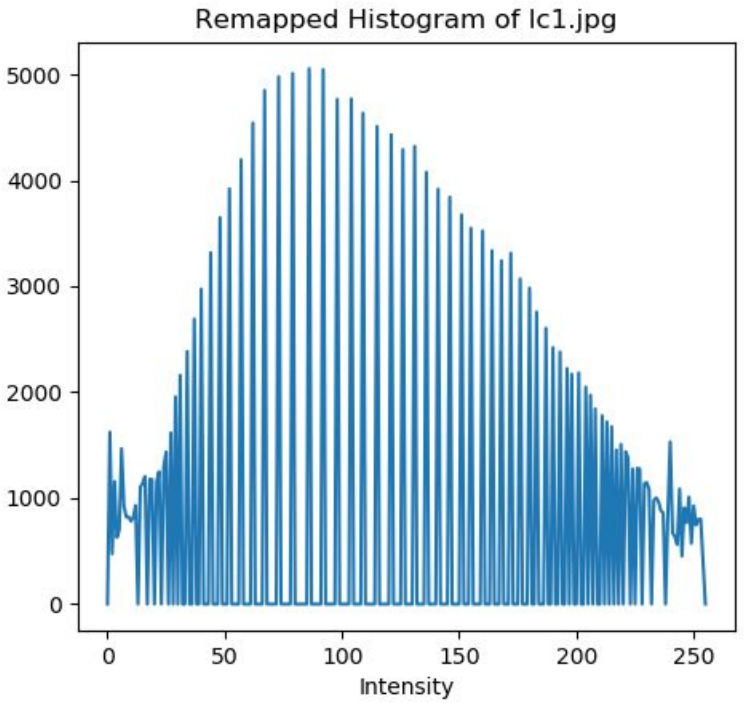
Part II: Contrast Enhancement

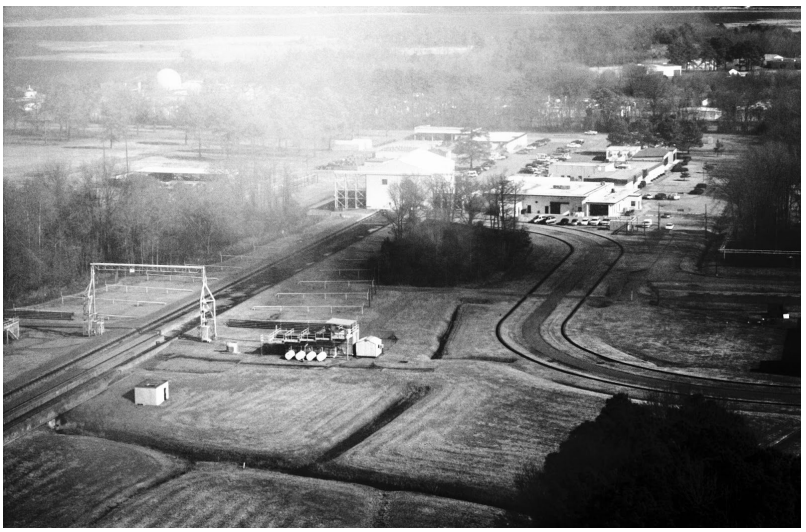
A histogram shows the frequency of occurrence for pixel intensities in an image. We can generate a histogram of a grayscale image by looping over all of the intensities (0 to 255), and counting the number of pixels that match an intensity in the image. A histogram feature of low-contrast images is a narrow band of frequent intensities, with large spans of infrequent intensities. One method of enhancing the contrast of an image is non-linear histogram equalization. The following two pages show original and contrast enhanced images, along with their histograms. These images and histograms were generated with the script `contrast.py` which can be seen in the appendix.

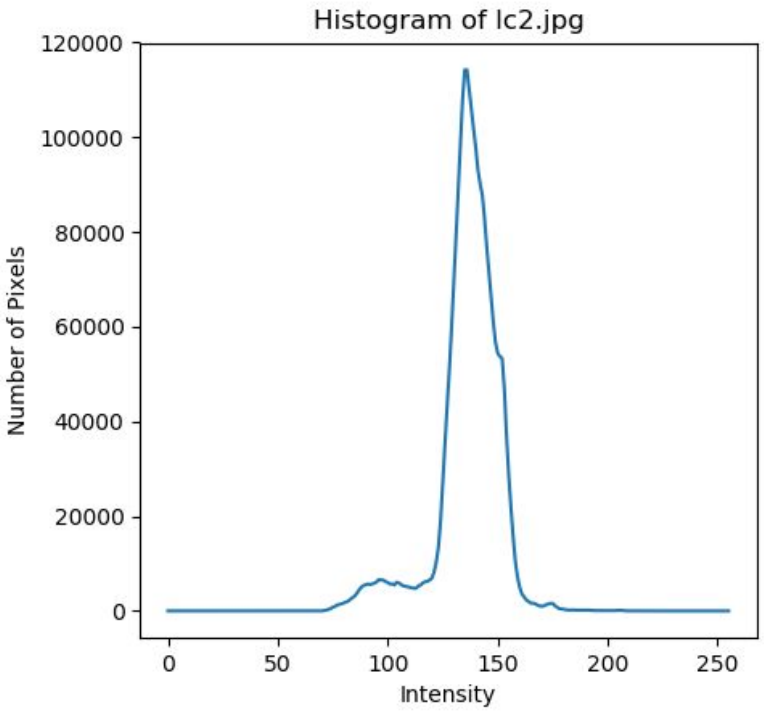
Original

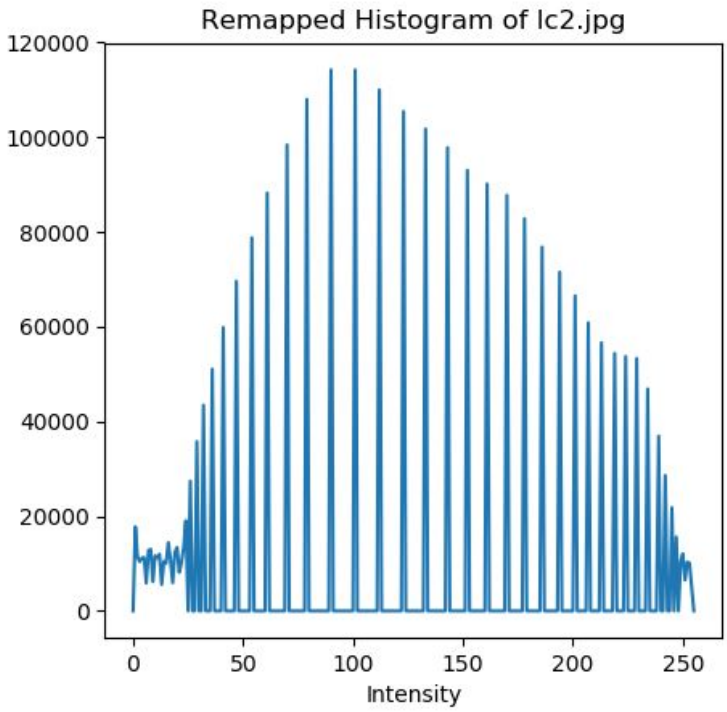
Contrast Enhanced



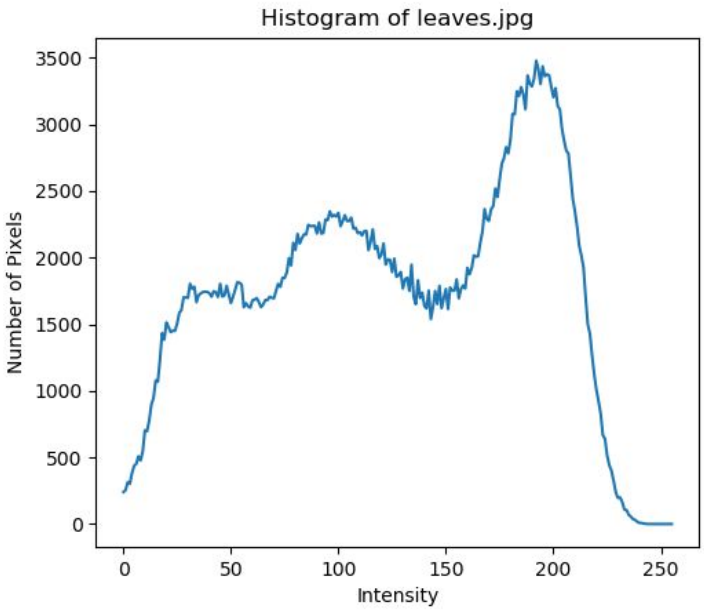


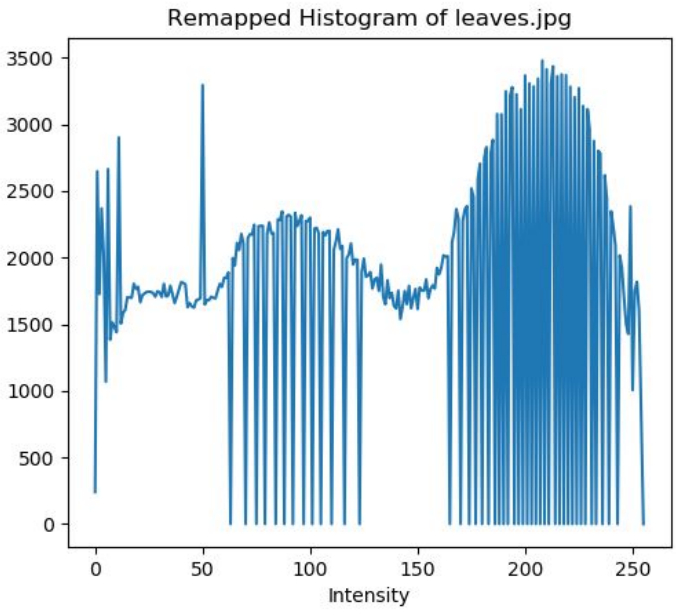



Histogram of lc2.jpg
Number of Pixels
120000
100000
80000
60000
40000
20000
0
0
50
100
150
200
250
Intensity


Remapped Histogram of lc2.jpg
120000
100000
80000
60000
40000
20000
0
0
50
100
150
200
250
Intensity

Performing histogram equalization on an image that doesn't have low contrast can cause saturation. Below is an image which has a decent spread in the original histogram. We see that after equalization, a large number of pixels have been remapped to intensity zero and 255, indicating saturation. Additionally, gaps are introduced between intensities, so the overall number of intensities present in the image have been reduced.





Overall this method of contrast enhancement works well for low-contrast images. A downside to this method is that histograms don't appear uniform - there are gaps between remapped intensities to which no pixels were assigned. This means an overall loss in the number of possible intensities present in an image, which could be noticeable.

Histogram equalization as described in this lab would not work on an RGB image by simply performing equalization one each color channel. By adjusting the intensities of separate color channels, we would be introducing false colors to the image. In order to enhance the contrast of a color image, one could convert the RGB image to YUV color space, perform histogram equalization on the Y intensity, then recombine the image and convert back to RGB.

Part III: Edge Detection

We will implement edge detection for images using the Sobel Operator. This consists of two 3x3 kernels that are convolved with original image data to approximate the horizontal and vertical gradient. Specifically, the kernels K_x and K_y are defined as

$$K_x \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

We convolve K_x and K_y with the source image S to get the horizontal and vertical gradients, G_x and G_y respectively.

$$G_x = K_x * S \qquad G_y = K_y * S$$

Finally, at each point of the image we can find the magnitude of the gradient using

$$|G| = \sqrt{{G_x}^2 + {G_y}^2}.$$

A large magnitude results from high gradient values, indicating the sharp transition from a high intensity to low intensity, or vice versa. This magnitude is then represented by the pixel intensity value of the image, showing edges. Edge detection is implemented using the Sobel operator in script `edge_detection.py`, located in the appendix. Figures 3 and 4 show the results of edge detection with a threshold of 200. A lower threshold results in more edges being shown, while a higher threshold limits the detection of edges with lower magnitudes.

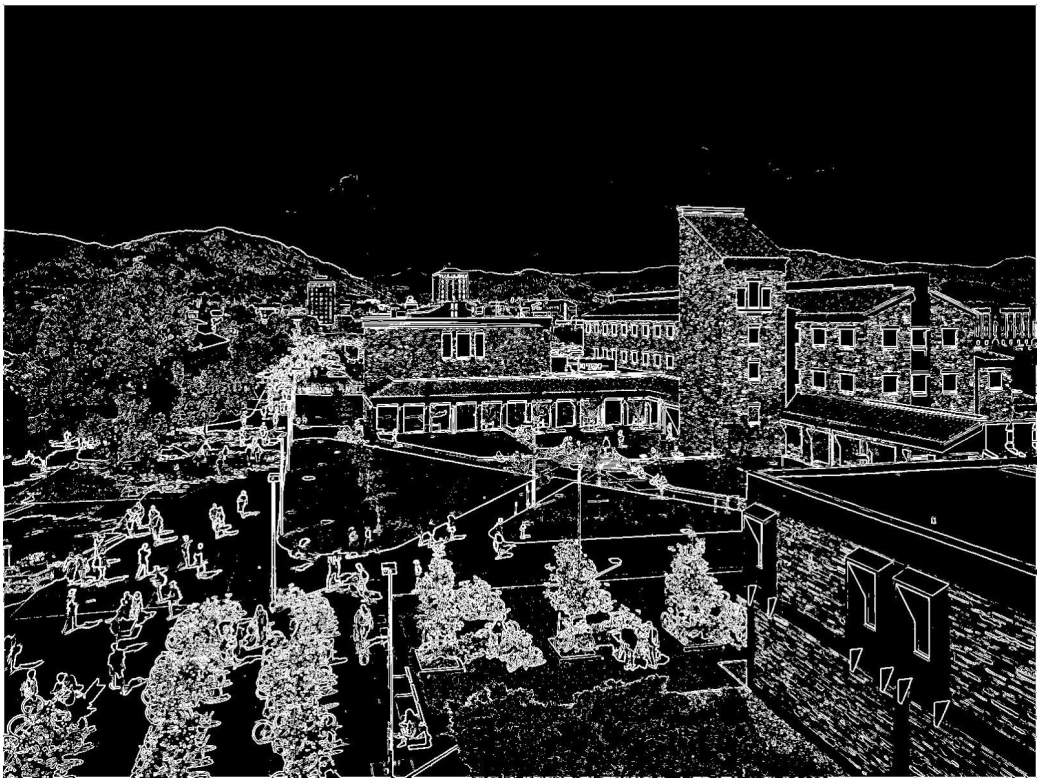


Figure 3 - Edge detection on test01.jpg

Figure 4 - Edge detection on test02.jpg

If edge detection was implemented on an RGB image, there are scenarios in which the edges detected in separate color planes could be different. For example, consider the edge produced by a red square on a pink background. This edge would be displayed with a high magnitude in the edge detection of the red plane. However, since pink is simply a lower intensity of red, this information would be mostly or completely omitted in the green and blue planes. Therefor, the edge detection of the green and blue planes would not show the edge created by the red square. Edge detection in grayscale is convenient because each color plane from an RGB image has some representation due to the conversion shown in Part I.

Part IV: Image Resizing

a. Downsampling

We can downsample an image in order to reduce the size of an image. We implement downsampling with the computation of the average intensity value for some non-overlapping grid of pixels, then assigning that intensity to a single pixel in the downsampled image. A grid of size N x N results in the downsampling of an image by a factor of N. The script `downsample.py`, shown in the appendix, down-samples an image by a factor `N`, provided in the calling arguments. Figure 5 shows the result of downsampling an image by a factor of 4.

Figure 5 - Downsampling by factor of 4 of test01.jpg

b. Upsampling

We can upsample an image to increase the size of the image. We implement upsampling by using bilinear interpolation of the original image data. For upsampling by a factor of N, we use 2 x 2 grids of pixels in the original image, which are bilinearly interpolated to the resulting (N+1) x (N+1) grids of pixels for the upsampled image. The script `upsample.py` up-samples an image by factor `N`, provided in the calling arguments. This script is shown in the appendix. Figure 6 shows the result of upsampling the image from Figure 5 by a factor of 4, restoring the image to its original size.

Figure 6 - Upsampling by factor of 4 of previously downsampled test01.jpg

c. Loss of Information in Downsampling

We note that the downsampled then upsampled image in Figure 6 is the same size as the original image, however it appears noisy and pixelated. This is because we lose information in an

image by downsampling it, then approximate new information with bilinear interpolation in the upsampling process. Figure 7 shows the absolute difference between the original image and the downsampled then downsampled image by a factor of 2. Figure 8 shows the difference in the same image but with upsampling and downsampling by a factor of 4. These images were generated using the script `diff.py`, located in the appendix.

We see that a higher down/up sample factor creates a larger difference from the original image. Areas of the image which are locally similar in intensity remain relatively unaffected by down/upsampling in comparison to areas which have a high diversity in intensity. An edge occurs in a small number of pixels, so by downsampling we lose information on edges, then by upsampling we "stretch" the edges out, resulting in a different edge from the original image. The differences are thus greater near edges, and indeed the difference images resemble the edge detection image from part 3.

Figure 7 - Difference between original and down/up sampled test01.jpg by factor of 2.

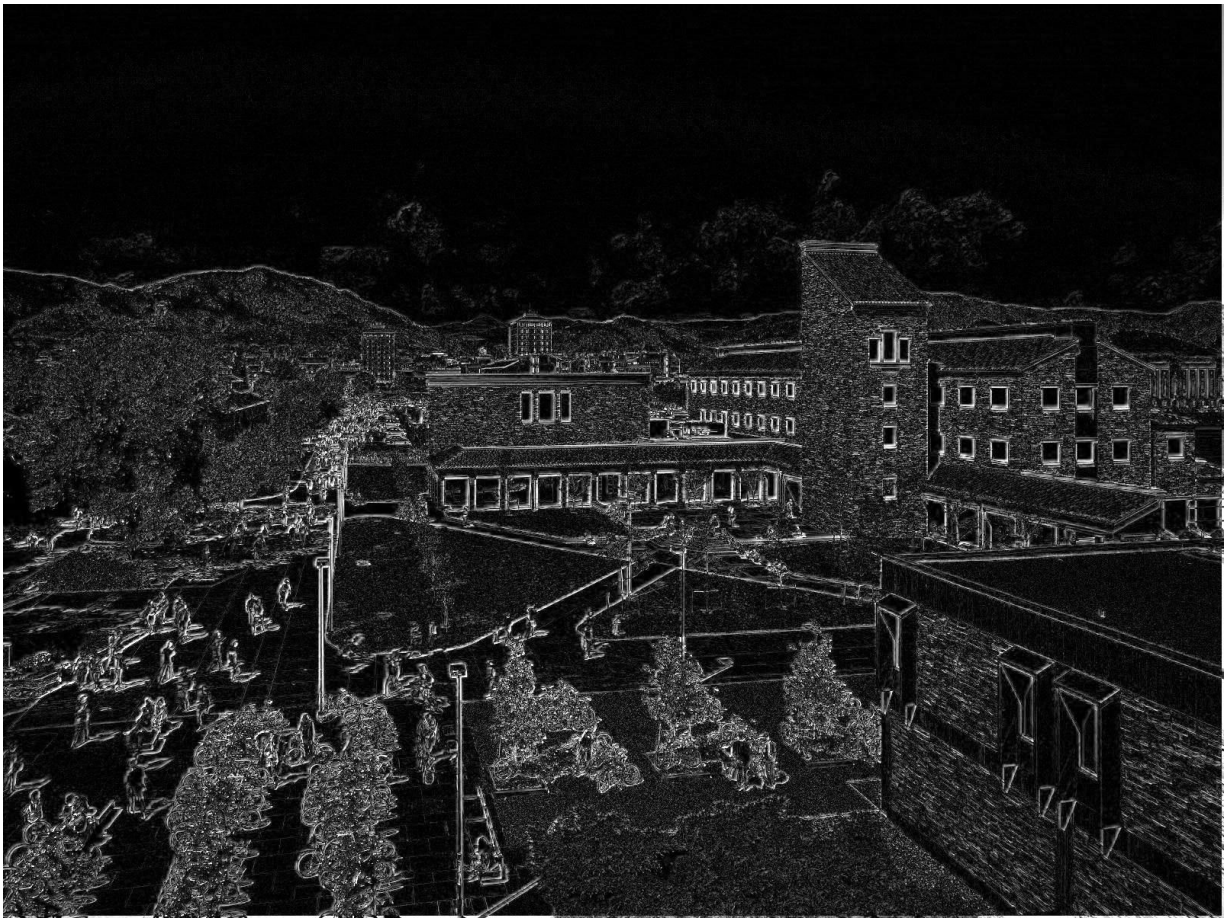


Figure 8 - Difference between original and down/up sampled test01.jpg by factor of 4.

Conclusion

In this lab we examined how digital images are represented in JPEG format and how we can process them. We converted RGB color images to grayscale and introduced a non-linear contrast enhancement method. Edge detection using the Sobel operator was examined, and the effects of resizing images using average downsampling and bilinear interpolation upsampling were investigated. By implementing these image processing techniques in Python, we learned how to easily and effectively manipulate and process image data, which finds application in many fields of study, including computer vision.

Appendix

grayscale.py

```
import numpy as np
from PIL import Image
import sys

Y = np.array([.299, .587, .114])

# converts array of rgb image data to grayscale
def grayscale(img):
   return np.clip(np.dot(img[:,:,:3], Y), 0, 255)

pil_image_in = Image.open(sys.argv[1])
image_in = np.asarray(pil_image_in, np.uint8)
filename_out = sys.argv[2]

image_out = grayscale(image_in)

# Output the image to a file
Image.fromarray(image_out.astype(np.uint8)).save(filename_out)

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

contrast.py

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import sys

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

filename_out = sys.argv[2]

histogram = np.zeros(256)

# create histogram
for intensity in range(0, 255):
```

```
   histogram[intensity] = (imdat == intensity).sum()

S = histogram[1:255].sum()
P = S / 254
remap = np.zeros(256)
remap[255] = 255
T = P
curr_sum = 0
out_val = 1
last_idx = 1

# this loop creates remap array
for intensity in range(1, 255):
   curr_sum = curr_sum + histogram[intensity]
   remap[intensity] = out_val
   if curr_sum > T:
       out_val = round(curr_sum / P)
       T = out_val * P

# change intensity values in image to remapped values.
# this could be done in the remap array loop instead of separately.
new_imdat = np.zeros(imdat.shape)
for intensity in range(0, 256):
   new_imdat[imdat == intensity] = remap[intensity]

# create new histogram
new_histogram = np.zeros(256)
for intensity in range(0, 256):
   new_histogram[intensity] = (new_imdat == intensity).sum()

# plot histogram comparison
plt.subplot(1, 2, 1)
plt.plot(histogram)
plt.xlabel('Intensity')
plt.ylabel('Number of Pixels')
plt.title('Histogram of ' + sys.argv[1])
plt.subplot(1, 2, 2)
plt.plot(new_histogram)
plt.xlabel('Intensity')
plt.title('Remapped Histogram of ' + sys.argv[1])
plt.show()

Image.fromarray(new_imdat).convert('L').save(filename_out)
Image.fromarray(new_imdat).show()
```

edge_detection.py

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import sys
from scipy import signal

kernel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
kernel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

pilim = Image.open(sys.argv[1])

if sys.argv[2]:
   out_filename = sys.argv[2]
else:
   out_filename = 'test_img_edge.jpg'

pilim = pilim.convert('L')  # convert to monochrome
imdat = np.asarray(pilim, np.float) # put PIL image into numpy array of floats

grad_x = signal.convolve2d(imdat, kernel_x)
grad_y = signal.convolve2d(imdat, kernel_y)

edge = np.sqrt(grad_x**2 + grad_y**2)

threshold = 200
edge[edge < threshold] = 0

Image.fromarray(edge).convert('L').save(out_filename)
Image.fromarray(edge).show()
```

downsample.py

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import sys

# usage: downsample.py image_in.jpg image_out.jpg N

# Assume we have an image with R rows and C columns, where R and C are both
divisible by N.
# N is Downsampling Factor
```

```
# create a new image of size R/N by C/N.
# Divide original image into non-overlapping blocks of size NxN, average pixel
# values from each block to obtain a pixel value for subsampled image.

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

out_filename = sys.argv[2]
N = int(sys.argv[3])

R, C = imdat.shape
print(f"Original Image size: Rows={R}, Cols={C}")

img_down = np.zeros((int(R/N), int(C/N)))
for i in range(int(R/N)):
   for j in range(int(C/N)):
       img_down[i, j] = np.average(imdat[i*N:(i*N)+N, j*N:(j*N)+N])

dR, dC = img_down.shape
print(f"Downsampled size: Rows={dR}, Cols={dC}")

Image.fromarray(img_down).convert('L').save(out_filename)
Image.fromarray(img_down).show()
```

upsample.py

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import sys
from scipy.interpolate import RectBivariateSpline

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

out_filename = sys.argv[2]
N = int(sys.argv[3])

R, C = imdat.shape
print(f"Original Image size: Rows={R}, Cols={C}")

img_up = np.zeros((int(R*N), int(C*N)))

# nested for loops over the smaller img
#TODO: this is very slow, make it better?
```

```
for i in range(R - 1):
   for j in range(C - 1):
       x = np.array([i, i+1])
       y = np.array([j, j+1])
       z = imdat[i:i+2, j:j+2]

       interp_spline = RectBivariateSpline(y, x, z, kx=1, ky=1)
       x1 = np.linspace(i, i+1, N+1)
       y1 = np.linspace(j, j+1, N+1)
       ivals = interp_spline(y1, x1)
       img_up[i*N:i*N+(N+1), j*N:j*N+(N+1)] = ivals

dR, dC = img_up.shape
print(f"Upsampled size: Rows={dR}, Cols={dC}")

Image.fromarray(img_up).convert('L').save(out_filename)
Image.fromarray(img_up).show()
```

diff.py

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import sys

UPFACTOR = 2    # increase visibility of difference

pilim1 = Image.open(sys.argv[1]).convert('L')
pilim2 = Image.open(sys.argv[2]).convert('L')
filename_out = sys.argv[3]

imdat1 = np.asarray(pilim1, np.float) # put PIL image into numpy array of
floats
imdat2 = np.asarray(pilim2, np.float)

if imdat1.shape != imdat2.shape:
   print("Image dimensions not equivalent")

diff = np.clip(np.abs(imdat1 - imdat2) * UPFACTOR, 0, 255)

Image.fromarray(diff).convert('L').save(filename_out)
Image.fromarray(diff).show()
```