**IMAGE PROCESSING**

**Assignment 2 (20 points)**

**Goutham Selvakumar (2092286)**

**#Importing the packages and running it**

import matplotlib.pyplot as plt

import cv2

import os

import PIL

import numpy as np

import math

from PIL import Image

import skimage

**#Running the image with the given path location**

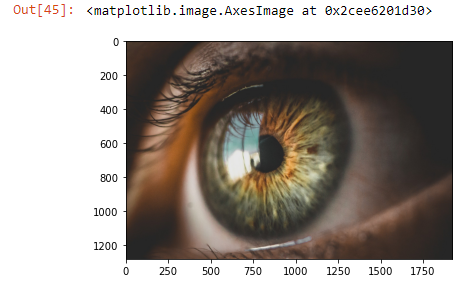
ret\_loc = r'C:\Users\admin\Desktop\Retina.jpg'

ret = cv2.imread(ret\_loc)

**#Converting to BGR to RGB**

ret\_2 = cv2.cvtColor(ret, cv2.COLOR\_BGR2RGB)

plt.imshow(ret\_2)



**Problem 1: *Zooming/Oversampling* and *Shrinking/Subsampling Images* by Pixel Replication (15/15)**

**Write your own function capable of shrinking and zooming an image by pixel replication and decimation. Assume that the desired zoom/shrink factors will be the inputs for your function and will have integer values: a negative input means shrink and a positive input means expand. Do not use MatLab built in functions (or other language libraries) for decimation and replication, although you can use other MatLab functions.**

1. **Use your program to shrink an image by a factor of 4 in each dimension. Show the shrunk image.**
2. **Use your program to zoom the image back to its original size.  Show the zoomed image and explain how and why the original image and the shrunk/zoomed images are different.**

**#Resizing the image**

**#Defining the scale with retina (image) and scale as the parameters**

def scale(ret, scale):

ret\_fx = ret.shape[0]

ret\_fy = ret.shape[1]

ret\_channels = ret.shape[2]

**#Downsizing the image**

if scale < 0:

new\_fx = round(ret\_fx / abs(scale))

new\_fy = round(ret\_fy / abs(scale))

**#Upscaling the image**

if scale > 0:

new\_fx = ret\_fx \* abs(scale)

new\_fy = ret\_fy \* abs(scale)

**#Scaling the image**

scaled\_fx = [int(ret\_fx \* y / new\_fx) for y in range(new\_fx)]

scaled\_fy = [int(ret\_fy \* x / new\_fy) for x in range(new\_fy)]

scaled\_ret = [[ret[int(ret\_fx \* y / new\_fx)] [int(ret\_fy \* x / new\_fy)] for x in range(new\_fy)] for y in range(new\_fx)]

return scaled\_ret

**#Downisizing the image by -4 factor**

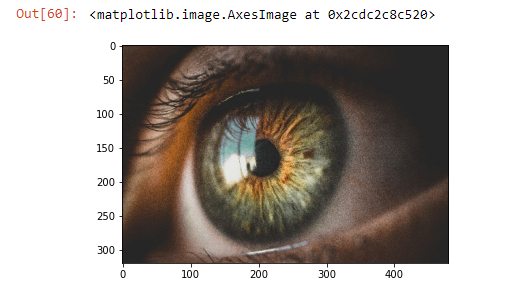
dz\_ret = scale(ret\_2, -4)

**#Saving the downsized image(retina)**

plt.savefig('dz\_ret.jpg')

**#Showing the downsized image(retina)**

plt.imshow(dz\_ret)



**#Zooming the image by 4 factor**

**#Increasing the pixel resolution in the image (retina)**

ret\_path = r'C:\Users\admin\Desktop\Retina.jpg'

ret = cv2.imread(ret\_path)

zoom\_ret = 4

for i in range(2):

image = np.repeat(ret, zoom\_ret, axis=i)

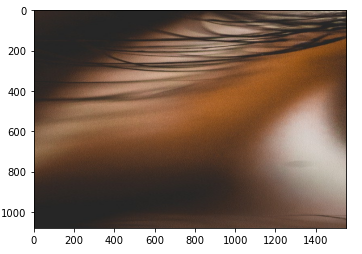
**#Cropping out the particular region to zoom in**

image = image[200:1750, 200:1750]

ret = cv2.cvtColor(ret, cv2.COLOR\_BGR2RGB)

plt.imshow(image)

plt.show()



**#Rescaling the image to it's original size**

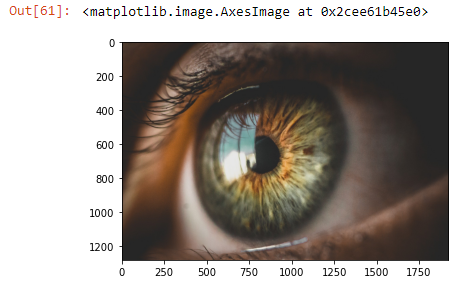
ret\_path = r'C:\Users\admin\Desktop\Retina.jpg'

ret = cv2.imread(ret\_path)

**#Rescaling the image**

rs\_ret = scale(ret\_2, 4)

plt.imshow(ret\_2)



The difference between the original image and the shrunk image is that we can see the shrunk image being distorted (noise) since it’s been divided by the absolute value of function [scale < 0]. Also, the image is again zoomed in the ratio of 200:1750 by selecting a particular region and cropping it out.

**Problem 2: Basic Grey Level Transformations (5/5)**

1. **Read and display an image.**

**#Running the image with the given path location**

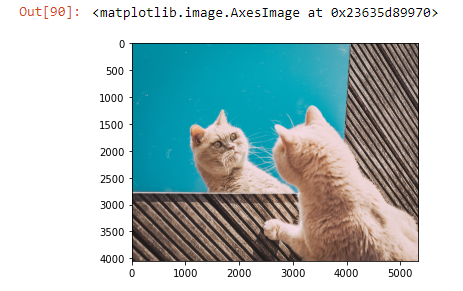
cat\_path = r'C:\Users\admin\Downloads\Cat.jpg'

cat\_2 = cv2.imread(cat\_path)

**#Converting BGR to RGB**

cat\_2 = cv2.cvtColor(cat\_2, cv2.COLOR\_BGR2RGB)

plt.imshow(cat\_2)



1. **Calculate the negative of the image and display it.**

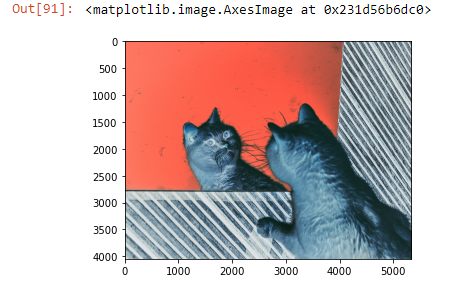
We can use opencv here, but skimage is easier to implement for viewing the image in negative.

**#Calculating the negative image using the scikit**

cat = skimage.io.imread(cat\_path)

neg\_cat = 255 - cat

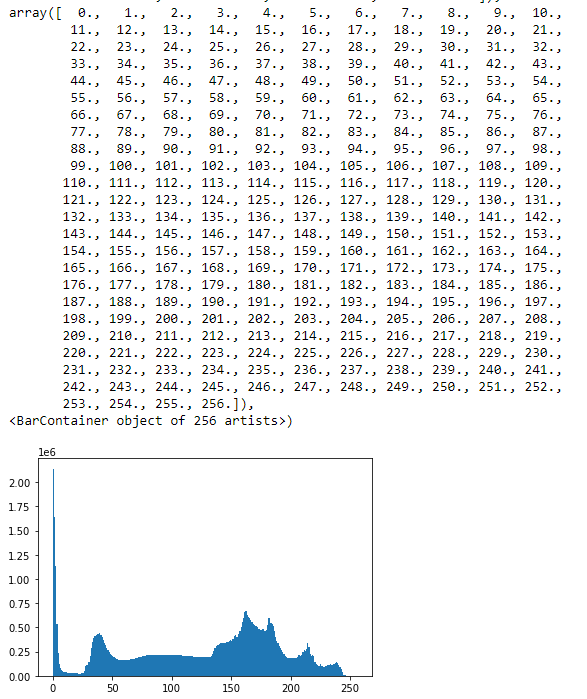
plt.imshow(neg\_cat)



1. **Perform contrast stretching. You can use the imadjust.m function to perform the image transformation.**

**#Showing the histogram of the image**

plt.hist(cat.ravel(), 256, [0, 256])



**#RGB Equalizing**

r, g, b = cv2.split(cat)

#red component is equalized

#blue component is equalized

#green component is equalized

equalize\_r = cv2.equalizeHist(r)

equalize\_g = cv2.equalizeHist(g)

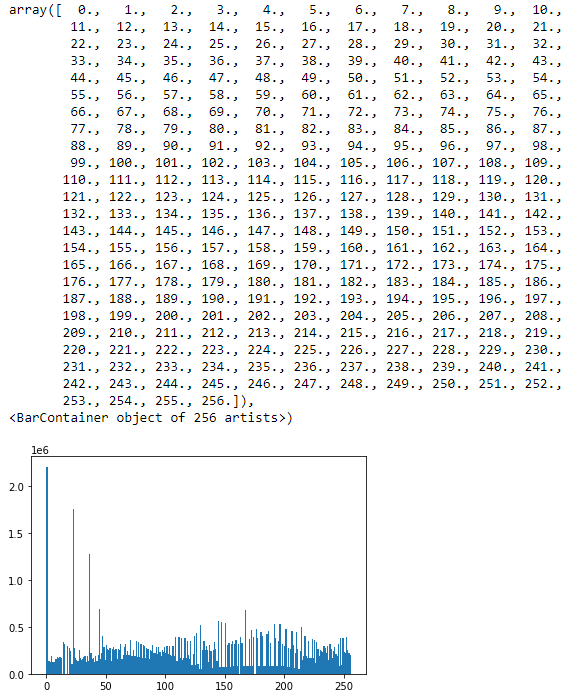
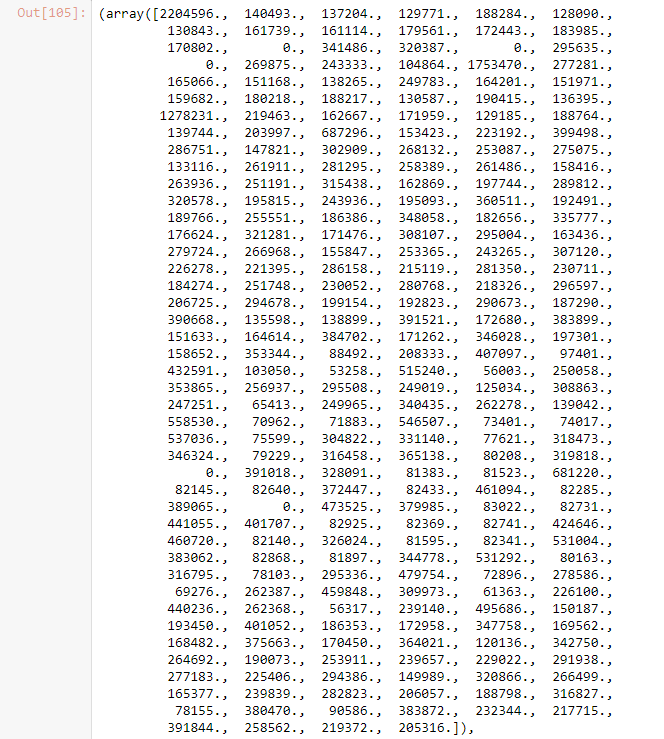
equalize\_b = cv2.equalizeHist(b)

**#Merging the image together with RGB**

eq\_cat = cv2.merge((equalize\_r, equalize\_g, equalize\_b))

**#Plotting the new histogram**

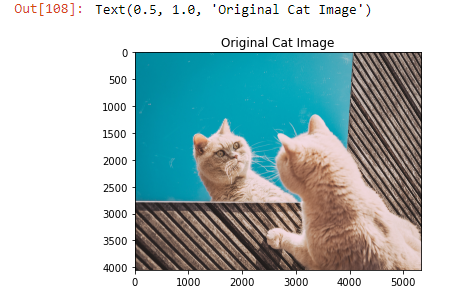
plt.hist(eq\_cat.ravel(), 256, [0, 256])



**#Showing the original image**

plt.imshow(cat)

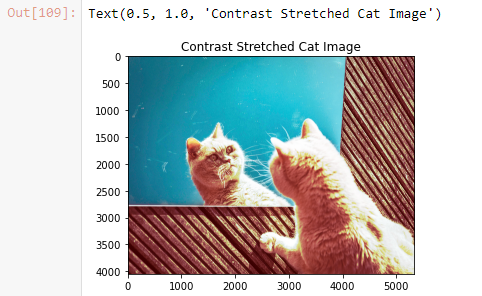
plt.title('Original Cat Image')



**#Contrast Image is displayed**

plt.imshow(eq\_cat)

plt.title('Contrast Stretched Cat Image')



**Work by hand:**

**Some basic relationships between pixels (5 points) Note: You do not need to use Mat lab code to solve this problem.**

**Consider the image segment shown below (the values in blue represent the p and q pixels):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 3 | 1 | 2 | 1 | (q) |
|  | 2 | 2 | 0 | 2 |  |
|  | 1 | 2 | 1 | 1 |  |
| (p) | 1 | 0 | 1 | 2 |  |

1. **Let V= {0, 1} and compute the lengths of the shortest 4-, 8- paths between p and q. If a particular path does not exist between these two points, explain why.**

* There is no path in the 4 for us to get from p to q. So, q has no {0, 1} in its neighborhood. Moreover, there are no more 4 path left for us to travel to.
* For the 8-path there are like 2 paths available from p to q. First one is the shortest 8- path between p and q is 4 that follows the same path. Second, is the path that is quite same as before, where we are able to travel with a step being reduced since we are able to travel diagonally from 0 to 1 where it reaches the q.

1. **Calculate the D4 distance (city-block distance) and the D8 distance (chessboard distance) between pixels p and q. Do these two distances depend on which path you choose between p and q? Explain your answer.**

The D4 distance between the pixels p and q is given using the formula,

Where, x = 0; y = 3; u = 3; v=0

D4 (p, q) = |x-u| + |y-v|

= |0-3| + |3-0|

= 3 + 3

D4 = 6

The D8 distance between the pixels p and q is given using the formula,

D8 (p, q) = max (|x-u|, |y-v|)

= max (|0-3|, |3-0|)

= max (3, 3)

D8 = 3

Therefore, the city-block and chessboard distance is autonomous of the way among p and q since the pixels p and q will have similar distances regardless of which way we pick.