CSc I6716 Fall 2020 -Assignment 1

Computer Science – The City College of New York Computer Vision Assignment 1 (Deadline: 10/04 Sunday) before midnight)

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import itertools

%matplotlib inline
```

In [2]:
```
InputImage = 'IDPicture.png'
```

1. Writing Assignments (10×4 = 40 points)

(1). How does an image change (e.g., objects' sizes in the image, field of view, etc.) when you change the zoom factors of your pinhole camera (i.e., the focal length of a pinhole camera is changed)?

(2). Give an intuitive explanation why a pinhole camera has an infinite depth of field, i.e., the images of objects are always sharp regardless their distances from the camera.

(3). In the thin lens model, 1/o + 1/i = 1/f, there are three variables, the focal length f, the object distance o and the image distance i (please refer to the Slides of the Image Formation lecture). If we define Z = o-f, and z = i-f, please write a few words to describe the physical meanings of Z and z, and then prove that *Zz* = *ff* given 1/o + 1/i = 1/f.

(4). Prove that, in the pinhole camera model, three collinear points in the world (i.e., they lie on a line in 3D space) are imaged into three collinear points on the image plane. You may either use geometric reasoning (with line drawings) or algebra deduction (using equations).

(1) The image plane gets bigger when you move away from the pinhole and smaller when you move closer to the pinhole.

(2) A pinhole camera model has nearly an "infinite depth of field" because there the light rays are nearly singular from the object to the image plane: you have less blurring.

(3) For z = i-f, Z = o-f, and 1/o + 1/i = 1/f:

-> (A) Zz = (i-f)(o-f) = (f-o)(f-i) = f^2 - f(o+i) + (o)(i)

-> (B) 1/o + 1/i = (o + i)/(io)

-> (A & B) Zz = f^2 - (i+o) [(o+i)/(io)]^-1 + (o)(i) = f^2 -(io) +(o)(i) = f^2

The product of the image to the focal plane and the object to the focal plane is the focal plane squared.

(4) In linear algebra, if points are collinear, their determinant is zero.

Take 3 points p1, p2, and p3 with coordinates (x_i, y_i, z_i) for i in 1, 2, 3.

D(p1, p2, p3) = D([x_1, y_1, z_1; x_2, y_2, z_2; x_3, y_3, z_3]) ≡ 0

In 2-D space, take any one (or combination) of the vectors (i = 1, 2, or 3) and set all those elements to zero. Therefore, you'll have a row or column of zeroes ... D(p1, p2, p3) = 0. A corollary is that if you have points that are collinear in high dimensional space, all the lower dimensional space(s) will be collinear.

2. Programming Assignments

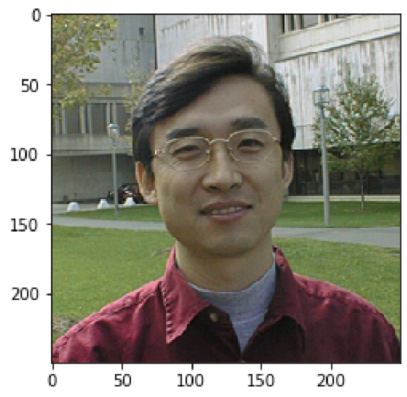
(Matlab preferred – here is a quick matlab tutorial. You may use C++, Java or Python if you like, but you may need to show the running of your program during my office hours when I ask you. If you don't have a Matlab license, CUNY faculty and students will be able to download a standalone Matlab version by creating an account using your CCNY email account at this website.) (15×4 = 60 points)

Image formation. In this small project, you are going to read, manipulate and write image data. The purpose of the project is to make you familiar with the basic digital image formations. Your program should do the following things:

1. Read in a color image C1(x,y) = (R(x,y), G(x,y), B(x,y)) in Windows BMP format, and display it.

In [3]:
```
C1 = plt.imread(InputImage)
plt.imshow(C1)
```

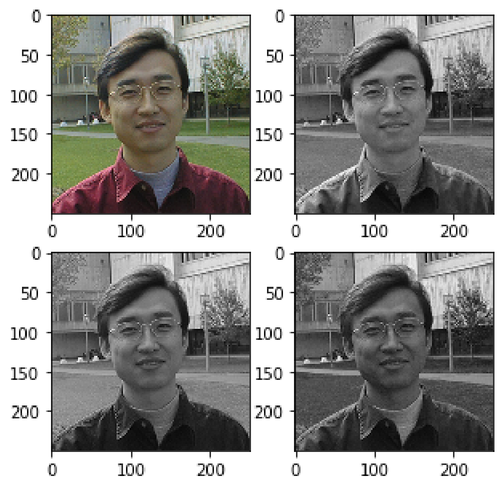
Out[3]:
```
<matplotlib.image.AxesImage at 0x7f80baf81b50>
```



2. Display the images of the three color components, R(x,y), G(x,y) and B(x,y), separately. You should display three black-white-like images.

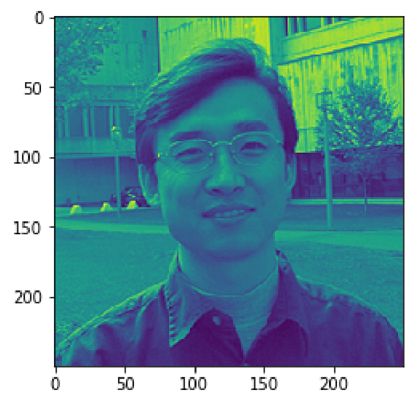
In [4]:
```python
C = [ C1[:,:,i] for i in range(3) ] #C is just the color images wheras C1 includes the Black
print(len(C))
print(len(C[0]))
print(len(C[0][0]))

R,G,B= C[0], C[1], C[2]



disimg = [[C1, R],[G,B]]
fig, axs = plt.subplots(2,2, figsize=(5,5))
for i in range(2):
    for j in range(2):
        axs[i,j].imshow(disimg[i][j], cmap='gray',  interpolation='nearest')

plt.show()
```

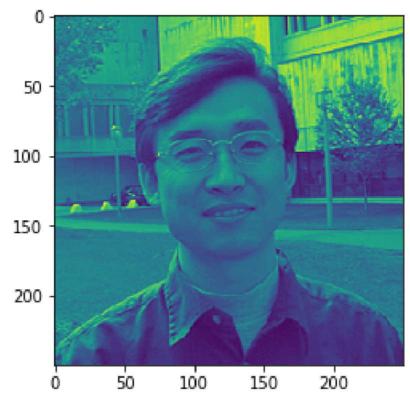
```
3
250
250
```



3. Generate an intensity image I(x,y) and display it. You should use the equation I = 0.299R + 0.587G + 0.114B (the NTSC standard for luminance) and tell us what are the differences between the intensity image thus generated from the one generated using a simple average of the R, G and B components. Please use an algorithm to show the differences instead by just observing the images by your eyes.

In [5]:
```python
def intensity(R, G, B):
    I = 0.299*R + 0.587*G + 0.114*B
    return I

def avgIntensity(R, G, B):
    I1 = ( (R+G+B)/3 )
    return I1
```

In [6]:
```
plt.imshow(intensity(R,G,B))
```
Out[6]:
```
<matplotlib.image.AxesImage at 0x7f80bb7ea190>
```


In [7]:
```
plt.imshow(avgIntensity(R,G,B))
```
Out[7]:
```
<matplotlib.image.AxesImage at 0x7f80bb399990>
```


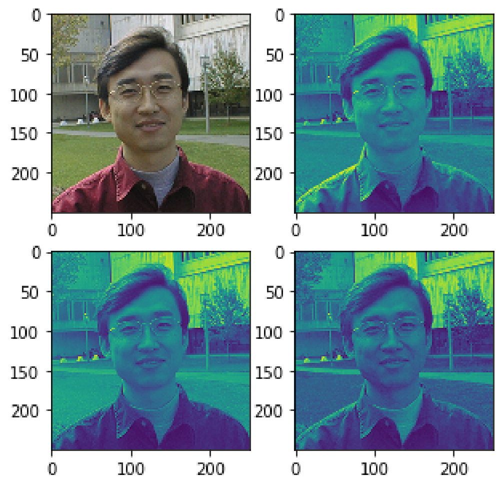
The NTSC standard for luminance looks very similar to the average of the mix of R, G, B. This is related to how our eye sees one color to dominate in an image luminance over the others. The NTSC proportion utilizes this principle. Let's double check to see if we can reproduce the composite-color image by manually using the NTSC proportions and view the monochromatic images meanwhile.

https://www.mathworks.com/matlabcentral/answers/412627-how-can-i-convert-rgb-image-to-ntsc-without-using-rgb2ntsc-command (https://www.mathworks.com/matlabcentral/answers/412627-how-can-i-convert-rgb-image-to-ntsc-without-using-rgb2ntsc-command)

BONUS

In [8]:

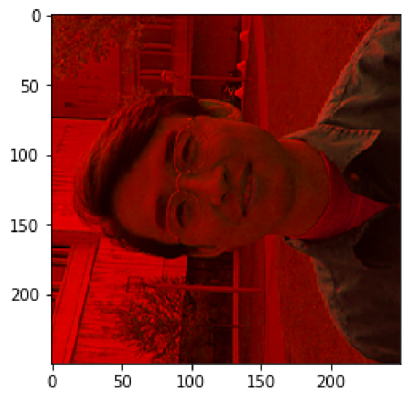
```python
disimg = [[C1, R],[G,B]]
fig, axs = plt.subplots(2,2, figsize=(5,5))
for i in range(2):
    for j in range(2):
        axs[i,j].imshow(disimg[i][j])

#plt.show()
```



In [9]:
```python
RGBImage = plt.imread(InputImage)
YIQ = np.array([
    0.299*RGBImage[:,:,0] + 0.587*RGBImage[:,:,1] + 0.114*RGBImage[:,:,2],
    0.596*RGBImage[:,:,0] - 0.274*RGBImage[:,:,1] - 0.322*RGBImage[:,:,2],
    0.211*RGBImage[:,:,0] - 0.523*RGBImage[:,:,1] + 0.312*RGBImage[:,:,2],
    #RGBImage[:,:,3]

])
plt.imshow(YIQ.T)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
for integers).
```

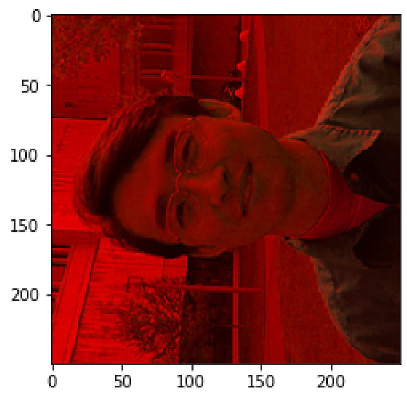
Out[9]:
```
<matplotlib.image.AxesImage at 0x7f80bbe19e10>
```



In [10]:
```python
RGBImage = plt.imread(InputImage)
YIQ = np.array([
    0.299*RGBImage[:,:,0] + 0.587*RGBImage[:,:,1] + 0.114*RGBImage[:,:,2],
    0.596*RGBImage[:,:,0] - 0.274*RGBImage[:,:,1] - 0.322*RGBImage[:,:,2],
    0.211*RGBImage[:,:,0] - 0.523*RGBImage[:,:,1] + 0.312*RGBImage[:,:,2],
    RGBImage[:,:,3]

])
plt.imshow(YIQ.T)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
for integers).
```

Out[10]:
```
<matplotlib.image.AxesImage at 0x7f80bc4c7550>
```



In [11]:
```python
print(len(C1), 'is length of C1')
print(len(C1[0]), 'is length of C1 first element in C1')
print(len(C1[1]), 'is length of C1 second element in C1')
print(len(C1[2]), 'is length of C1 third element in C1')
if len(C1[0][0]) == len(C1[0][1]) == len(C1[0][2]):
    print(f'Each element has {len(C1[0][0])} columns')
C1
```

```
250 is length of C1
250 is length of C1 first element in C1
250 is length of C1 second element in C1
250 is length of C1 third element in C1
Each element has 4 columns
```

Out[11]:
```
array([[[0.6117647 , 0.5529412 , 0.34117648, 1.        ],
        [0.5058824 , 0.4509804 , 0.2627451 , 1.        ],
        [0.54901963, 0.5019608 , 0.31764707, 1.        ],
        ...,
        [0.8666667 , 0.8666667 , 0.8392157 , 1.        ],
        [0.8627451 , 0.8666667 , 0.85490197, 1.        ],
        [0.8509804 , 0.85882354, 0.84705883, 1.        ]],

       [[0.65882355, 0.60784316, 0.42745098, 1.        ],
        [0.5882353 , 0.5372549 , 0.3647059 , 1.        ],
        [0.5254902 , 0.47843137, 0.2784314 , 1.        ],
        ...,
        [0.87058824, 0.87058824, 0.8509804 , 1.        ],
        [0.85882354, 0.8666667 , 0.85490197, 1.        ],
        [0.85882354, 0.8666667 , 0.85490197, 1.        ]],

       [[0.5647059 , 0.52156866, 0.39607844, 1.        ],
        [0.59607846, 0.54509807, 0.42745098, 1.        ],
        [0.5254902 , 0.45882353, 0.32156864, 1.        ],
        ...,
        [0.8509804 , 0.85490197, 0.8352941 , 1.        ],
        [0.85490197, 0.8627451 , 0.8509804 , 1.        ],
        [0.84313726, 0.85490197, 0.84313726, 1.        ]],

       ...,

       [[0.60784316, 0.29411766, 0.3882353 , 1.        ],
        [0.59607846, 0.2784314 , 0.37254903, 1.        ],
        [0.49411765, 0.21176471, 0.28627452, 1.        ],
        ...,
        [0.17254902, 0.11372549, 0.10980392, 1.        ],
        [0.15686275, 0.11372549, 0.10588235, 1.        ],
        [0.13333334, 0.09411765, 0.08235294, 1.        ]],

       [[0.6666667 , 0.38039216, 0.42352942, 1.        ],
        [0.54509807, 0.21960784, 0.30588236, 1.        ],
        [0.4862745 , 0.20392157, 0.22352941, 1.        ],
        ...,
        [0.18431373, 0.11764706, 0.10980392, 1.        ],
        [0.16078432, 0.11764706, 0.10196079, 1.        ],
        [0.15294118, 0.10980392, 0.09411765, 1.        ]],

       [[0.60784316, 0.30588236, 0.2901961 , 1.        ],
        [0.42352942, 0.15686275, 0.18431373, 1.        ],
        [0.5254902 , 0.24705882, 0.25882354, 1.        ],
        ...,
        [0.16078432, 0.12156863, 0.10196079, 1.        ],
        [0.14509805, 0.10196079, 0.08627451, 1.        ],
        [0.16078432, 0.11764706, 0.09803922, 1.        ]]], dtype=float32)
```

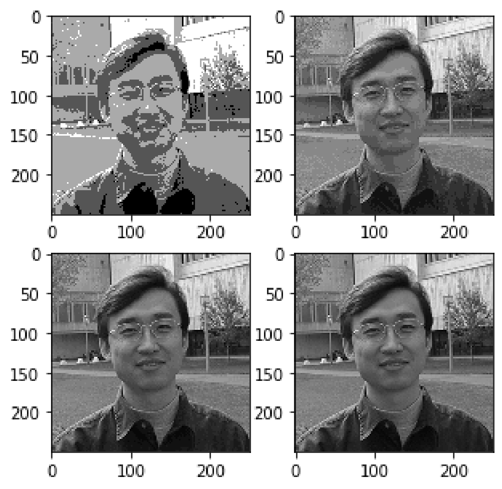
Well it seems there is more to the formula to produce the composite-color image.

4. The original intensity image should have 256 gray levels. Please uniformly quantize this image into K levels (with K=4, 16, 32, 64). As an example, when K=2, pixels whose values are below 128 are turned to 0, otherwise to 255. Display the four

quantized images with four different K levels, and tell us how the images still look like or different from the original ones, and where you cannot see any differences.

In [12]:
```python
def quantize(k,greyImage):
    zeroArray = np.zeros((1,k))
    quantiles = np.copy(zeroArray)
    'initialize a numpy array'
    quantImage = np.copy(greyImage)
    #print('size of quantImage: ',quantImage.shape)

    for i in range(k):
        quantiles[0,i]=(i/k)
    for i,j in itertools.product(range(len(greyImage)), range(len(greyImage[0]))):
        'Get the difference from the quantiles'
        numDiff = abs(quantiles[0] - greyImage[i,j])
        'obtain the minimum difference'
        minDiff = min(numDiff)
#           print('quantiles: ', quantiles)
#           print('numDiff, minDiff :', numDiff, minDiff)
        'use index of minimum difference; the first occurence'
        ind = np.where(numDiff == minDiff)[0] [0]

#           print('ind, i, j :',ind,i, j)
        'quantize rational'
        quantImage[i,j] = quantiles[0][ind]
    return quantImage
```

In [13]:
```python
fig, axs = plt.subplots(2,2, figsize=(5,5))
ind = list( itertools.product(range(2),range(2)) )
bit = [4, 16, 32, 64]
RGB = 0.299*R + 0.587*G + 0.114*B
images = []
for k in bit:
    img = quantize(k,RGB)
    images.append(img)# prevents code from duplicating last image in all subplots
    axs[ ind[bit.index(k)] ].imshow(images[bit.index(k)], cmap='gray',  interpolation='neares
    #plt.imshow(img)
```



In [14]:
```python
RGB = 0.299*R + 0.587*G + 0.114*B
```

In [15]:
```python
RGB_64 = np.copy(quantize(64,RGB))
RGB_32 = np.copy(quantize(32,RGB))
RGB_16 = np.copy(quantize(16,RGB))
RGB_4 = np.copy(quantize(4,RGB))
```

In [16]:

```
RGB_4 == RGB_64
```

Out[16]:

```
array([[False, False,  True, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

In [17]:

```
RGB_32 == RGB_64
```

Out[17]:

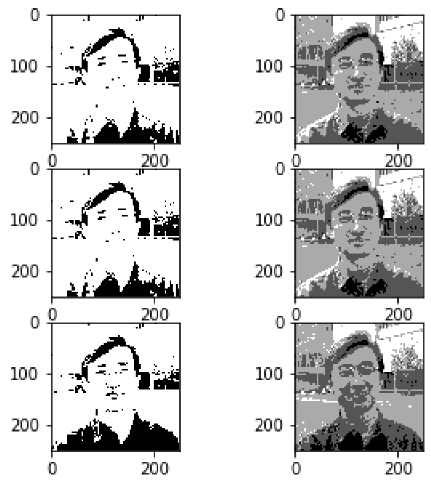
```
array([[False, False,  True, ..., False, False, False],
       [False,  True,  True, ...,  True, False, False],
       [False, False,  True, ...,  True, False,  True],
       ...,
       [ True, False, False, ...,  True,  True, False],
       [ True, False, False, ..., False,  True,  True],
       [False, False, False, ...,  True, False,  True]])
```

The image granular at low bit. By 32bit the images smoothens out and looks very similar to 64bit.

5. Quantize the original three-band color image C1(x,y) into K level color images CK(x,y)= (R'(x,y), G'(x,y), B'(x,y)) (with uniform intervals) , and display them. You may choose K=2 and 4 (for each band). Do they have any advantages in viewing and/or in computer processing (e.g. transmission or segmentation)?

In [42]:

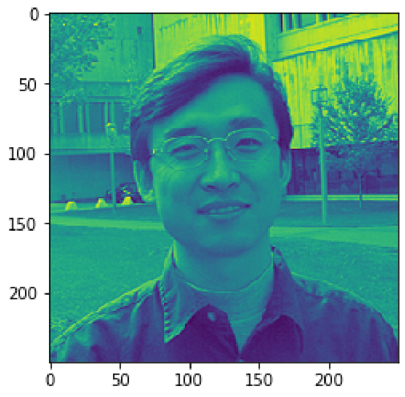
```
fig, axs = plt.subplots(3,2, figsize=(5,5))
arr = [[R,R],[G,G],[B,B]]
images = []
iters = list(itertools.product(range(3),range(2)))

for i, j in iters:
    for k in [2,4]:
        img = np.copy(quantize(k, arr[i][j]))
        images.append(img)
        axs[i][j].imshow(images[iters.index((i,j))], cmap='gray',  interpolation='nearest')
        #plt.imshow(img)

```



6. Quantize the original three-band color image C1(x,y) into a color image CL(x,y)= (R'(x,y), G'(x,y), B'(x,y)) (with a logarithmic function) , and display it. You may choose a function I' =C ln (I+1) (for each band), where I is the original value (0-255) , I' is the quantized value, and C is a constant to scale I' into (0-255), and ln is the natural logarithmic function. Please describe how you find the best C value so for an input in the range of 0-255, the output range is still 0 – 255. Note that when I = 0, I' = 0 too.

In [43]:
```python
RGBlog = 0.299*np.log(R+1) + 0.587*np.log(G+1) + 0.114*np.log(B+1)
RGB256log = quantize(256,RGBlog)

plt.imshow(RGB256log)

```

Out[43]:
```
<matplotlib.image.AxesImage at 0x7f80bbeddf90>
```



The constants for each band are chosen to be the factor recommended by NSTC for each respective band. More or less we see that quantization varies exponentially. This is why the log image looks similar to the non log image.

https://matplotlib.org/tutorials/introductory/images.html (https://matplotlib.org/tutorials/introductory/images.html)