EE 525 – Assignment 4

Tanmay Gore – tgore03@iastate.edu

1. **K Means:**

Q.1 a) Loss Function:

$$F(S, \mu) = \sum_{j=1}^{k} \sum_{x_i \in S_j} \|x_i - \mu_j\|^2$$

where, $S = \{s_1, s_2, \dots, s_k\}$ are clusters

$\mu = \{\mu_1, \mu_2, \dots, \mu_k\}$ are centroids

k = no. of clusters.

Since we calculate $\|x_i - \mu_j\|^2$ for only those data points which belong to cluster j, we can write above equation as

$$F(\eta, \mu) = \sum_{i=1}^{n} \sum_{j=1}^{k} \eta_{ij} \|x_i - \mu_j\|^2$$

where $\eta_{ij} = 1$, if x_i belongs to j^{th} cluster else 0.

b) $$F(S, \mu) = \left(\sum_{x_i \in S_j} \|x_i - \mu\|^2\right) + |S_j| \, \|\mu - \mu_j\|^2 \quad (1)$$

where μ – mean of the data points in a cluster.

From Lloyd's Algorithm
After $t-1$ iterations
$F(S^{t+1}, \mu^t) \le F(S^t, \mu^t)$

(As first step in each iteration is to assign each data point to its closest cluster)

Second step, recalculates the cluster centers (centroids)
$\therefore F(S^{t+1}, \mu^{t+1}) \le F(S^{t+1}, \mu^t)$

hence each iteration of Lloyd's algorithm only decreases the value of F

c) η is a $n \times n$ matrix and contains only 0 or 1
$\therefore$ In worst case each point in the matrix is itself a center.

Since F decreases monotonically, we cannot visit the same state of η twice.
$\therefore$ Total number of possible states are 2^{n^2} in worst case, which is finite

Hence algorithm will terminate with upper bound $T = 2^{n^2}$ steps.

2. Nonnegative Matrix Factorization

Q.2

a) To compute $\nabla_H f(H,W)$

Given: $f(H,W) = \frac{1}{2} \|X - WH\|_F^2$

where $W \in \mathbb{R}^{n \times r}$ $\quad H \in \mathbb{R}^{r \times d}$

$$\therefore f(H,W) = \frac{1}{2}\left[(X-WH).(X-WH)^T\right]$$

$$= \frac{1}{2}\left[(X.X^T) - (XH^TW^T) - (WHX^T) + (WHH^TW^T)\right]$$

Taking partial derivative w.r.t. H.

$$\nabla_H f(H,W) = \frac{1}{2}\left[0 - W^TX - W^TX + W^T(H^TW^T)^T + W^T(WH)\right]$$

$$= \frac{1}{2}\left[-2W^TX + 2W^T(WH)\right]$$

$$= W^TWH - W^TX$$

$$= W^T(WH - X)$$

b) Differentiating f w.r.t. W.

$$\nabla_W f(H,W) = \frac{1}{2}\left[0 - XH^T - XH^T + (H^TW^T)^TH^T + W(HH^T)\right]$$

$$= \frac{1}{2}\left[-2XH^T + 2WHH^T\right]$$

$$= H^T(WH - X)$$

c) Given $H_{ij}^{t+1} \leftarrow H_{ij}^t \dfrac{((W^t)^TX)_{ij}}{((W^t)^TW^tH^t)_{ij}}$ — (1)

where $i = \{1,\dots,r\}$ & $j = \{1,\dots,d\}$

dividing $\nabla_H f(H,W)$ by W^TWH

$$\frac{\nabla_H f(H,W)}{W^TWH} = 1 - \frac{W^TX}{W^TWX}$$

$$\therefore \frac{W^TX}{W^TWH} = 1 - \frac{\nabla_H f(H,W)}{W^TWH}$$

Substituting this in (1)

$$H_{ij}^{t+1} \leftarrow - \frac{H_{ij}^t}{((W^t)^TW^tH^t)_{ij}}(\nabla_H f(H,W)) + H_{ij}^t$$

hence $\alpha_H^t = \dfrac{H_{ij}^t}{((W^t)^TW^tH^t)_{ij}}$

Similarly dividing $\nabla_W f(W,H)$ by WHH^T, we get

$$\frac{XH^T}{WHH^T} = 1 - \frac{\nabla_W f(H,W)}{WHH^T}$$

Substituting this value in

$$W_{ij}^{t+1} \leftarrow \frac{W_{ij}^t (X(H^t)^T)_{ij}}{(W^tH^t(H^t)^T)_{ij}}$$

$i = \{1,\dots,n\}$, $j = \{1,\dots,r\}$

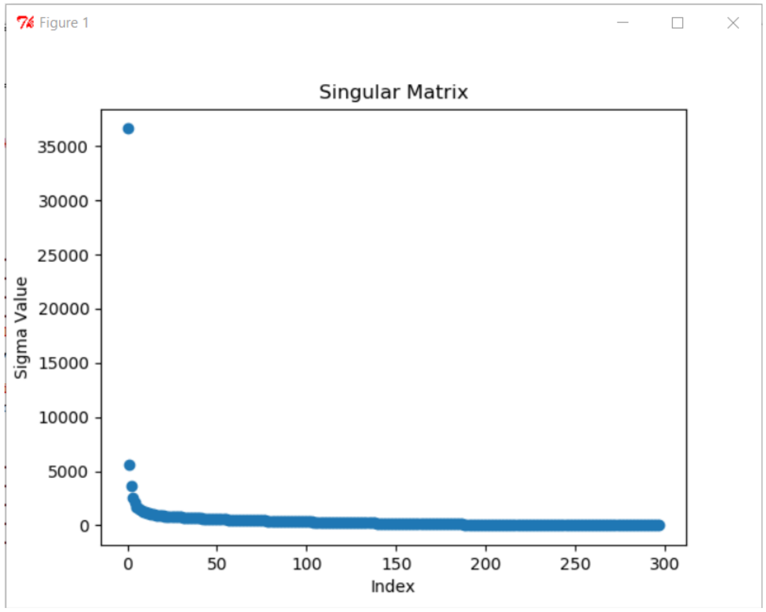
$$W_{ij}^{t+1} \leftarrow W_{ij}^t - \frac{W_{ij}^t \nabla_W f(\cancel{W}H,W)}{(W^tH^t(H^t)^T)_{ij}}$$

$$\therefore \alpha_W^t = \frac{W_{ij}^t}{(W^tH^t(H^t)^T)_{ij}}$$

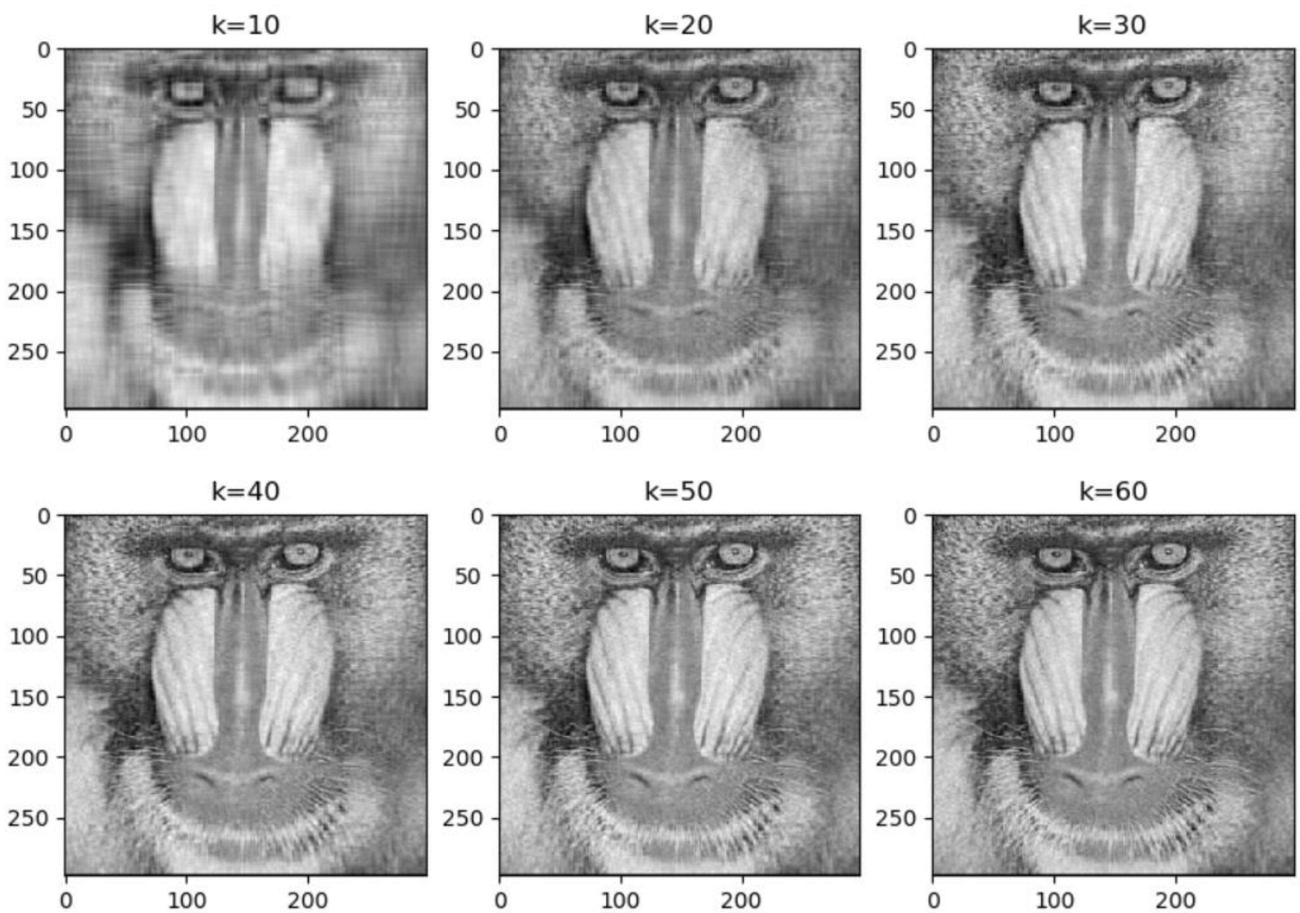
d) Yes, H^{t+1} and W^{t+1} satisfy the non negativity constant.

3. **SVD Image Compression:**

   a. **Plot Singular Matrix:**



   b. **Approximation of X using SVD:**



c. Compression Ratio

k	Compression Ratio
10	0.0335570469799
20	0.0671140939597
30	0.10067114094
40	0.134228187919
50	0.167785234899
60	0.201342281879

As seen above images after k = 40 does provide significant clarity in the image however the storage space required to store them keeps on increasing. Therefore storing images with k >40 does not provide any benefits.

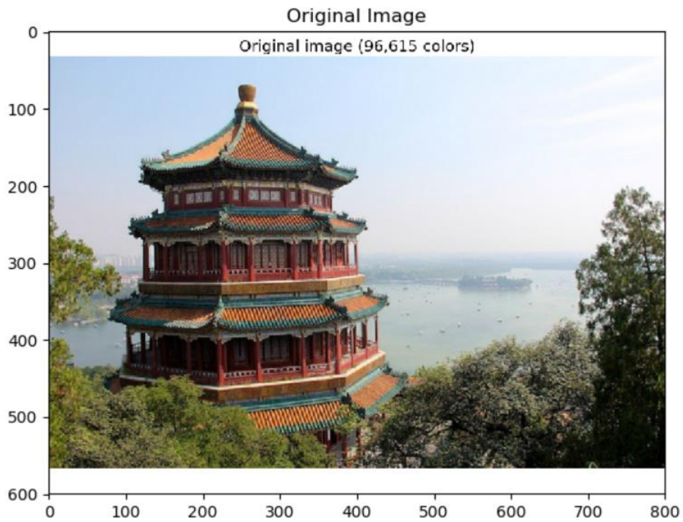
d. Code

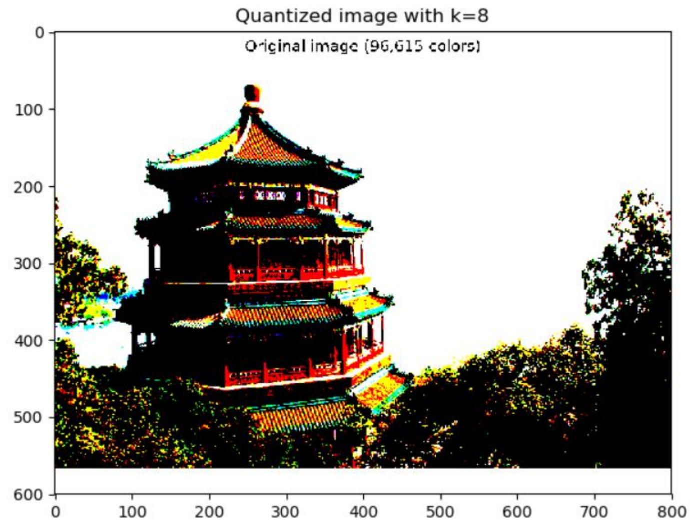
```
import numpy as np
import matplotlib.pyplot as plt
import math


#Read the Image file
from PIL import Image
im = Image.open("mandrill-grayscale.jpg") #Can be many different formats.
pix = im.load()

l,h= im.size
x = np.empty((l,h), dtype=float)

#Take Average of the pixels and save the matrix
for i in range(l):
   for j in range(h):
      val = im.getpixel((i,j))
      x[j][i] = sum(val)/3

plt.figure(1)
plt.imshow(x, cmap='gray')
plt.title("Original Image in gray scale")
plt.show()
#Calculate the SVD of x
u,s,v = np.linalg.svd(x)

#Print singular values
s_index = np.empty(s.size, dtype=int)
for i in range(s.size):
   s_index[i] = i
plt.scatter(s_index, s)
plt.xlabel("Index")
plt.ylabel("Sigma Value")
plt.title("Singular Matrix")
plt.show()


def reconstruct(k):
   uk = u[:,:k]
   sk = np.diag(s[:k])
```

```
    vk = v[:k,:]
    no_stored = uk.size + k + vk.size
    #print "Numbers stored for k =",k,":", no_stored
    compression_ratio = float(no_stored)/(u.size+v.size+s.size)
    #print "Compression Ratio =", compression_ratio*100,"%\n"

    #Reconstruct image
    t = np.dot(uk,sk)
    return np.dot(t,vk), compression_ratio

#Reconstruct x
plt.figure(2)
plt.tight_layout()
print "k     Compression Ratio"

plt.subplot(2, 3, 1)
plt.title("k=10")
nx, ratio = reconstruct(10)
plt.imshow(nx, cmap='gray')
print "10   ", ratio

plt.subplot(2, 3, 2)
plt.title("k=20")
nx, ratio = reconstruct(20)
plt.imshow(nx, cmap='gray')
print "20   ", ratio

plt.subplot(2, 3, 3)
plt.title("k=30")
nx, ratio = reconstruct(30)
plt.imshow(nx, cmap='gray')
print "30   ", ratio

plt.subplot(2, 3, 4)
plt.title("k=40")
nx, ratio = reconstruct(40)
plt.imshow(nx, cmap='gray')
print "40   ", ratio

plt.subplot(2, 3, 5)
plt.title("k=50")
nx, ratio = reconstruct(50)
plt.imshow(nx, cmap='gray')
print "50   ", ratio

plt.subplot(2, 3, 6)
plt.title("k=60")
nx, ratio = reconstruct(60)
plt.imshow(nx, cmap='gray')
print "60   ", ratio
plt.show()
```

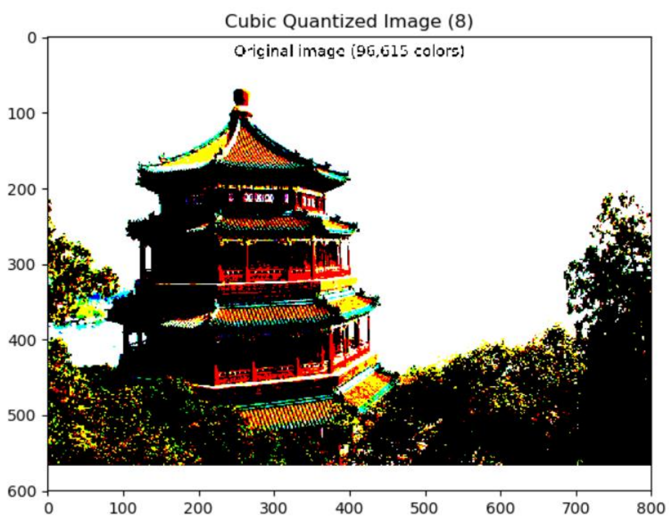
4. **Cubic vs KMeans Quantization:**

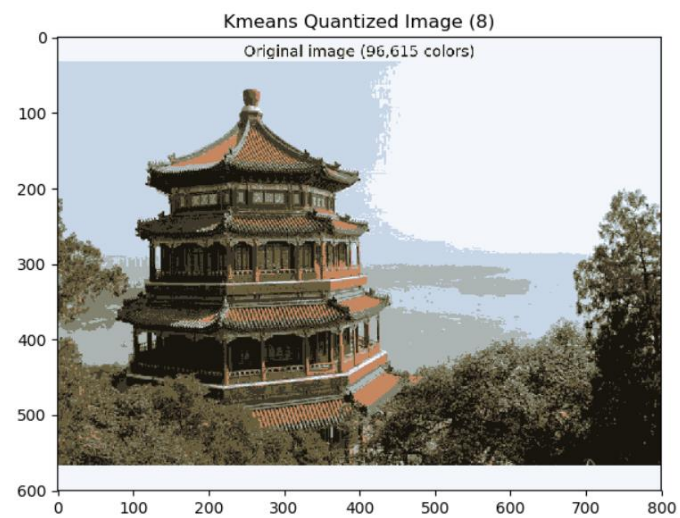
a. **Cubic Quantization for k = 8**



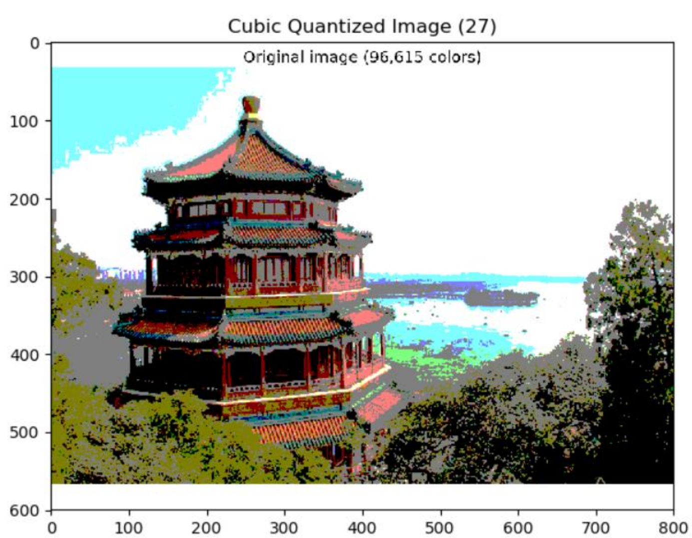


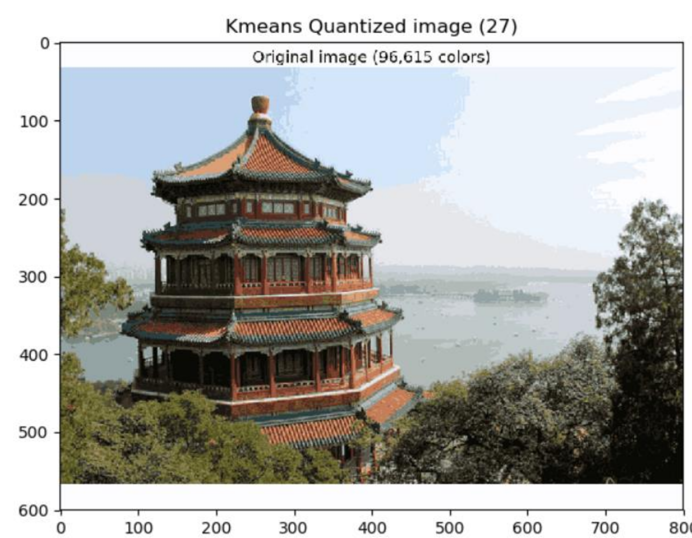
b. **Kmeans Quantization for k = 8**

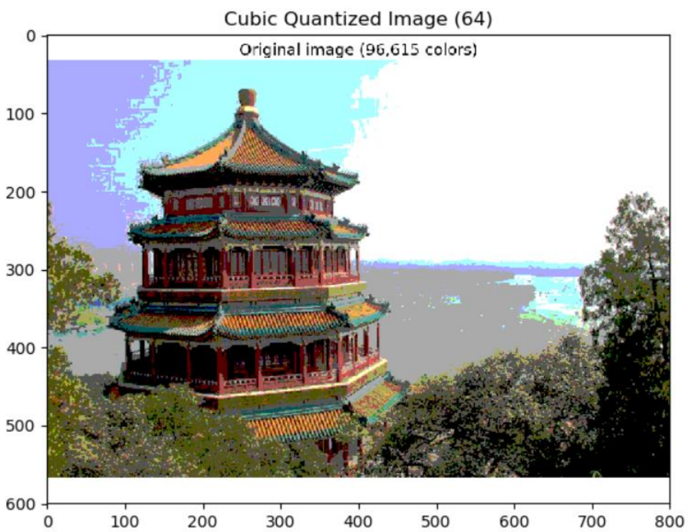


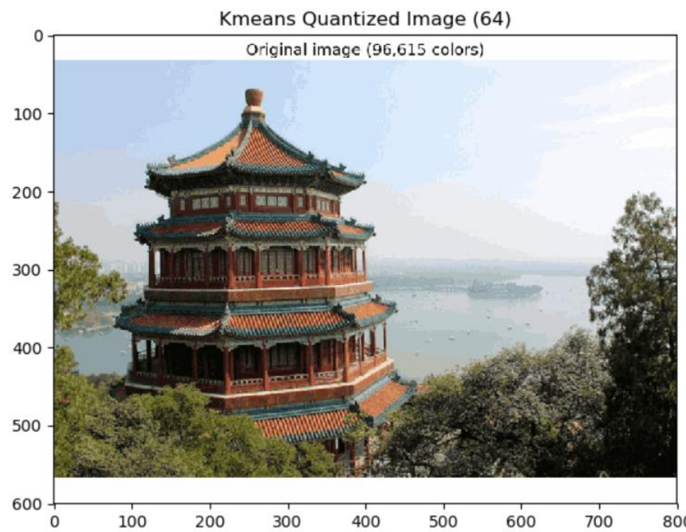


c. **For k = 27 & 64**









Code:

```
import numpy as np
import matplotlib.pyplot as plt
import itertools
from scipy.spatial import distance
from time import time
from sklearn.cluster import KMeans
from PIL import Image

def quantization(color_set, i, j):
    min_dst = np.inf
    min_lst = []
    for lst in color_set:
        dst = distance.euclidean(np.array(lst),x[i,j])
        if dst < min_dst:
            min_dst = dst
            min_lst = lst
    return np.array(min_lst)

def cubic_quantization(cset):
    #generate color representative set
    color_set =  list(itertools.product(cset, repeat=3))

    x_quantized = np.empty_like(x)
    x_shape = x.shape
    for i in range(x_shape[0]):
        for j in range(x_shape[1]):
            x_quantized[i,j] = quantization(color_set, i, j)
    return x_quantized

#K-Means
def kmeans_quantization(k):
    #Train K-Means and predict clusters
    kmeans=KMeans(n_clusters=k)
    clusters = kmeans.fit_predict(nx)

    #Reconstruct image
    x_centroid = np.empty_like(x)
    idx = 0
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x_centroid[i,j] = kmeans.cluster_centers_[clusters[idx]]
```

```
        idx+=1
    return x_centroid

def plot_images(k, x_cubic, x_kmeans):
    #Plot Kmeans vs cubic quantized image
    plt.figure(k)
    plt.subplot(1, 2, 1)
    plt.imshow(x_cubic)
    plt.title("Cubic Quantized Image ("+str(k)+")")
    plt.subplot(1,2,2)
    plt.title("Kmeans Quantized Image ("+str(k)+")")
    plt.imshow(x_kmeans)
    plt.show()

#Read image
im = Image.open("palace.png")
im.load()
x = np.array(im)

#Reshape x
nx = np.reshape(x, (x.shape[0]*x.shape[1], x.shape[2]))

#Perform cubic quantization on k=8
t0 = time()
x_cubic = cubic_quantization([0,255])
print "Cubic Quantizatoin: time required =", time() - t0,"secs\n"

#Plot quantized vs original image
plt.figure(1)
plt.subplot(1, 2, 1)
plt.imshow(x)
plt.title("Original Image")
plt.subplot(1,2,2)
plt.title("Quantized image with k=8")
plt.imshow(x_cubic)
plt.show()

#Perform Kmeans quantization on k=8
t0 = time()
x_kmeans = kmeans_quantization(8)
print "Kmeans Quantization: time required =", time() - t0,"secs\n"

#Plot Kmeans vs cubic quantized image
plot_images(8, x_cubic, x_kmeans)

#Perform cubic quantization on k=27
t0 = time()
x_cubic = cubic_quantization([0,127,255])
print "Cubic Quantizatoin: time required =", time() - t0,"secs\n"

#Perform Kmeans quantization on k=27
t0 = time()
x_kmeans = kmeans_quantization(27)
print "Kmeans Quantization: time required =", time() - t0,"secs\n"

#Plot Kmeans vs cubic quantized image
plot_images(27, x_cubic, x_kmeans)


#Perform cubic quantization on k=64
```

```
t0 = time()
x_cubic = cubic_quantization([0,85,170,255])
print "Cubic Quantizatoin: time required =", time() - t0,"secs\n"

#Perform Kmeans quantization on k=64
t0 = time()
x_kmeans = kmeans_quantization(64)
print "Kmeans Quantization: time required =", time() - t0,"secs\n"

#Plot Kmeans vs cubic quantized image
plot_images(64, x_cubic, x_kmeans)
```

References:

1. Discussed with Nitesh Gupta however all solutions were compiled and written by me.