Uncompress Script

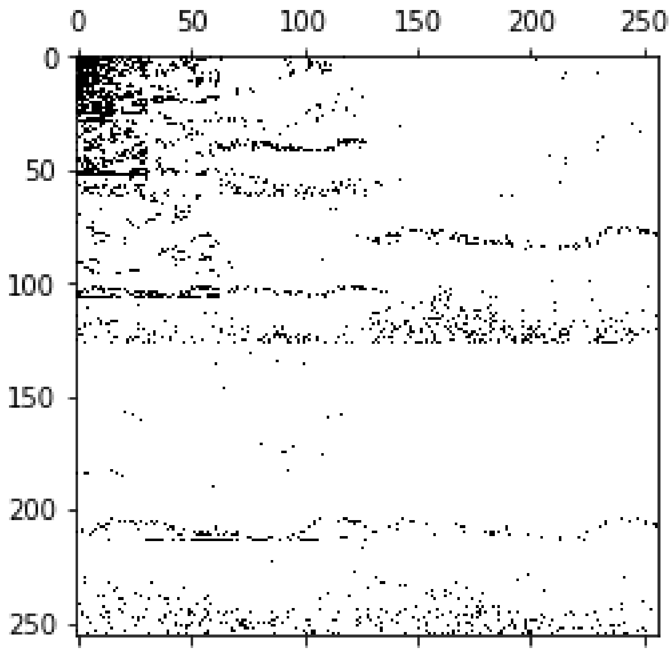
February 16, 2018

In [1]:
```
# import the packages
import numpy as np
from scipy.misc import imread, imresize, imsave
import matplotlib.pyplot as plt
from numpy import matlib
import math
from scipy import stats
import imageio
from skimage.transform import resize
import skimage
import zlib, sys
import gzip
import matplotlib
import scipy
import copy
import random
import io
import sys
```

In [2]:
```
# define a function to covert the image to a gray scale image
def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

# define a function to get the proper Haar matrix and permutation matrix
def GetHaarMatrices(N):
    Q = np.matrix("[1,1;1,-1]")
    M = int(N/2)
    T = np.kron(matlib.eye(M),Q)/np.sqrt(2)
    P = np.vstack((matlib.eye(N)[::2,:],matlib.eye(N)[1::2,:]))
    return T,P
```

In [3]:
```
# use zlib to uncompress the compressed_data
compressed_data = gzip.open('compressed_data.txt.gz', 'rb').read()
decompress_data = zlib.decompress(compressed_data)

# convert the byte-like object to numpy array
decompress_data = np.frombuffer(decompress_data, dtype=int)
```

```
# reshape the data to 2D
indices = np.reshape(decompress_data, (256, 256))

# show the image before reverse log quantization
plt.spy(indices)
plt.show()

print(indices)
```




```
[[127  31  50 ...   0   0   0]
 [ 64  63  46 ...   0   0   0]
 [ 71  81  26 ...   0   0   0]
 ...
 [  0   0   0 ...   0   0   0]
 [  1   0   0 ...   0   0   0]
 [  0   0   0 ...   0   0   0]]
```

In [4]:
```
# make the codebook.txt readable for python
codebook = []

with open('codebook.txt', 'r') as f1:
    codebook = [line.strip() for line in f1]

codebook = [float(i) for i in codebook]
```

```
print(codebook)
```

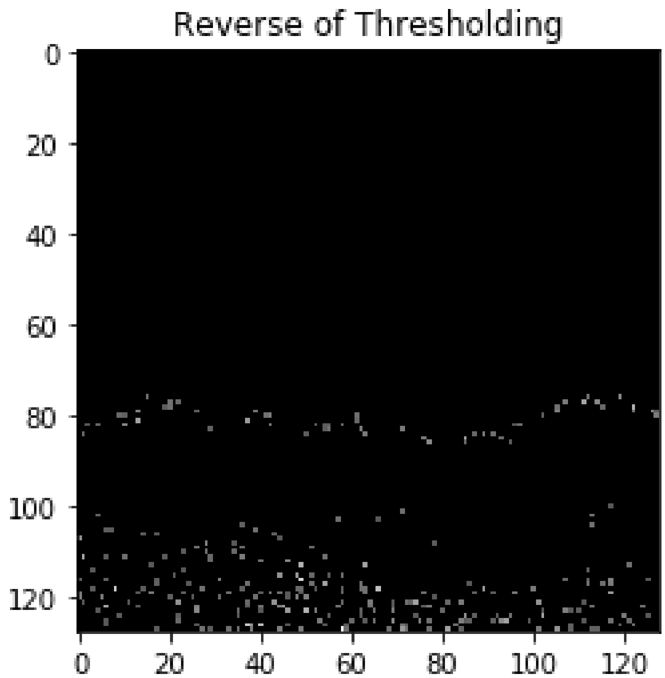
```
[0.0, 0.16562914134646228, 0.17135073430213965, 0.17896790245200203, 0.19287271932004935, 0.20
```

```
In [5]: # using codebook and indices to recover the quanta data
        quanta = np.empty((256, 256))

        for i in range(quanta.shape[0]):
            for j in range(quanta.shape[1]):
                quanta[i][j] = codebook[indices[i][j]]

        print(quanta)
```

```
[[130.59818677   0.79415368   2.19811157 ...   0.           0.
    0.        ]
 [  4.63455098   4.45363012   1.76914495 ...   0.           0.
    0.        ]
 [  6.71842264  11.68084684   0.62617807 ...   0.           0.
    0.        ]
 ...
 [  0.           0.           0.         ...   0.           0.
    0.        ]
 [  0.16562914   0.           0.         ...   0.           0.
    0.        ]
 [  0.           0.           0.         ...   0.           0.
    0.        ]]
```

```
In [6]: # reverse threshold to F
        # make a deep copy of F as G
        G = copy.deepcopy(quanta)

        # read in F row by row, find the min nonzero pixel
        # put the number from data codebook before apply thresholding function
        # in order to put the data back to nonzero
        def reverse_thresholding(source):
            index = 0

            for i in range(source.shape[0]):
                for j in range(source.shape[1]):

                    if source[i][j] == 0:
                        index += 1
                    else:
                        continue
```

```python
# Apply reverse thresholding function to M
reverse_thresholding(G)

# show the image after apply to reverse threshold
plt.imshow(G[128:256,128:256], cmap = plt.get_cmap('gray'))
plt.title("Reverse of Thresholding")
plt.show()

print(G)
```




```
[[130.59818677   0.79415368   2.19811157 ...   0.           0.
    0.        ]
 [  4.63455098   4.45363012   1.76914495 ...   0.           0.
    0.        ]
 [  6.71842264  11.68084684   0.62617807 ...   0.           0.
    0.        ]
 ...
 [  0.           0.           0.         ...   0.           0.
    0.        ]
 [  0.16562914   0.           0.         ...   0.           0.
    0.        ]
 [  0.           0.           0.         ...   0.           0.
    0.        ]]
```

In [7]:
```
# open the sign.txt to put back the sign
sign = open('sign.txt', 'rb').read()

# convert the byte-like object to numpy array
sign = np.frombuffer(sign)

# reshape the sign to 2D numpy array
sign = np.reshape(sign, (256, 256))
print(sign)
```

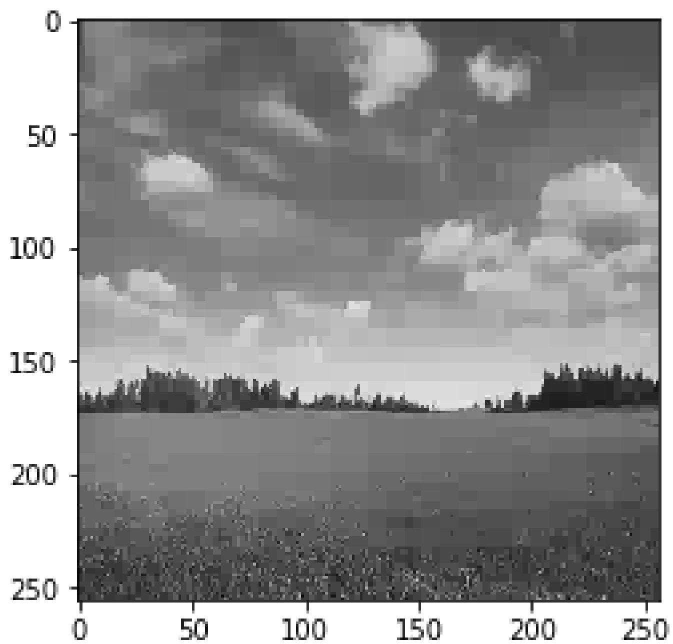
```
[[ 1.  1.  1. ...  1. -1. -1.]
 [ 1. -1. -1. ...  1. -1. -1.]
 [-1. -1. -1. ...  1. -1. -1.]
 ...
 [ 1. -1. -1. ...  1.  1. -1.]
 [ 1. -1. -1. ...  1.  1. -1.]
 [-1. -1. -1. ... -1.  1. -1.]]
```

In [8]:
```
# put the nagative sign back to the correct position
G = G * sign
print(G)
```

```
[[130.59818677   0.79415368   2.19811157 ...   0.          -0.
   -0.        ]
 [  4.63455098  -4.45363012  -1.76914495 ...   0.          -0.
   -0.        ]
 [ -6.71842264 -11.68084684  -0.62617807 ...   0.          -0.
   -0.        ]
 ...
 [  0.          -0.          -0.         ...   0.           0.
   -0.        ]
 [  0.16562914  -0.          -0.         ...   0.           0.
   -0.        ]
 [ -0.          -0.          -0.         ...  -0.           0.
   -0.        ]]
```

In [9]:
```
# make a deep copy of G
J = copy.deepcopy(G)

# get number of times of decoding and the starting point
N = len(J)
times = int(np.log2(N))
start = 2

# Doing full-level decoding (Backward Haar Transform)
for i in range(times):
    T,P = GetHaarMatrices(start)
```

```
        J[0:start, 0:start] = T.T*P.T*J[0:start, 0:start]*P*T
        start = 2 * start

    # show the result of full-level decoding
    plt.figure()
    plt.imshow(J, cmap = plt.get_cmap('gray'))
    plt.show()

    # print the info of J
    print(J)
```



```
[[0.31332279 0.31332279 0.43884974 ... 0.31523406 0.31523406 0.31523406]
 [0.31332279 0.31332279 0.43884974 ... 0.31523406 0.31523406 0.31523406]
 [0.31332279 0.31332279 0.43884974 ... 0.31523406 0.31523406 0.31523406]
 ...
 [0.17002039 0.17002039 0.25283496 ... 0.28278207 0.1971067  0.1971067 ]
 [0.25283496 0.25283496 0.25283496 ... 0.28278207 0.1971067  0.1971067 ]
 [0.25283496 0.25283496 0.25283496 ... 0.28278207 0.1971067  0.1971067 ]]
```

In [10]:
```
#####################
# get PSNR#
#####################

# get the information from the original image(before full-level encoding)
# reads in the original image
```

```
A = imageio.imread('image.jpg')

# resize the image(before apply gray scale function) as a 256 by 256 matrix
A = skimage.transform.resize(A, [256, 256], mode='constant')

# Apply the rgb2gray function to the image
A = rgb2gray(A)

print(A)
```

```
[[0.31905515 0.3648937  0.42410076 ... 0.35208627 0.35567083 0.35600784]
 [0.31661375 0.36307557 0.43115786 ... 0.35208627 0.35567083 0.35600784]
 [0.31623509 0.3615686  0.43780567 ... 0.35208627 0.35567083 0.35600784]
 ...
 [0.23155268 0.1562254  0.29268602 ... 0.31176704 0.14906503 0.21840392]
 [0.23837193 0.19653385 0.30720091 ... 0.28660407 0.13619611 0.25433132]
 [0.31247219 0.25910593 0.29068716 ... 0.39854037 0.252657   0.21340572]]
```

In [11]:
```
# get the maximum value of the original image
maxValue = np.amax(A)
print(maxValue)
```

```
0.9999999999999999
```

In [12]:
```
# get the 2D info of origianl image
print(A)
```

```
[[0.31905515 0.3648937  0.42410076 ... 0.35208627 0.35567083 0.35600784]
 [0.31661375 0.36307557 0.43115786 ... 0.35208627 0.35567083 0.35600784]
 [0.31623509 0.3615686  0.43780567 ... 0.35208627 0.35567083 0.35600784]
 ...
 [0.23155268 0.1562254  0.29268602 ... 0.31176704 0.14906503 0.21840392]
 [0.23837193 0.19653385 0.30720091 ... 0.28660407 0.13619611 0.25433132]
 [0.31247219 0.25910593 0.29068716 ... 0.39854037 0.252657   0.21340572]]
```

In [13]:
```
# get the 2D info of the reconstructed image
print(J)
```

```
[[0.31332279 0.31332279 0.43884974 ... 0.31523406 0.31523406 0.31523406]
 [0.31332279 0.31332279 0.43884974 ... 0.31523406 0.31523406 0.31523406]
 [0.31332279 0.31332279 0.43884974 ... 0.31523406 0.31523406 0.31523406]
 ...
 [0.17002039 0.17002039 0.25283496 ... 0.28278207 0.1971067  0.1971067 ]
 [0.25283496 0.25283496 0.25283496 ... 0.28278207 0.1971067  0.1971067 ]
 [0.25283496 0.25283496 0.25283496 ... 0.28278207 0.1971067  0.1971067 ]]
```

```
In [14]: # calculate the MSE
         MSE_arr = np.empty([J.shape[0], J.shape[1]])
         for i in range(J.shape[0]):
             for j in range(J.shape[1]):
                 MSE_arr[i][j] = ((A[i][j] - J[i][j])**2)

         MSE = 0
         for a in range(MSE_arr.shape[0]):
             for b in range(MSE_arr.shape[1]):
                 MSE += MSE_arr[a][b]

         MSE = MSE/(MSE_arr.shape[0]*MSE_arr.shape[1])

         print(MSE)
```

```
0.0024329983400938992
```

```
In [15]: # calculate the PSNR
         PSNR = 20*math.log10(maxValue) - 10*math.log10(MSE)
         print(PSNR)
```

```
26.138581873652385
```