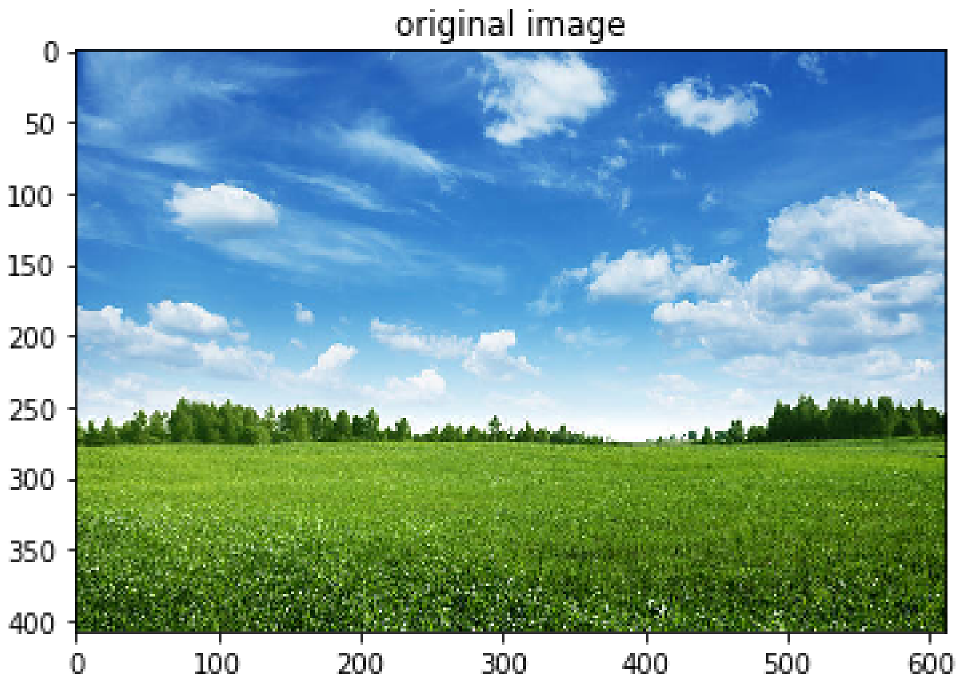
Compress Script

February 16, 2018

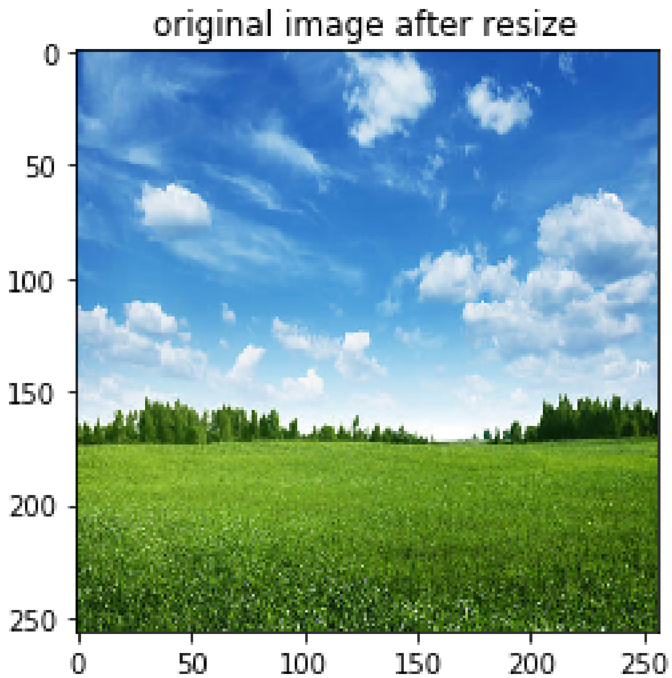
In [1]:
```
# import the packages
import numpy as np
from scipy.misc import imread, imresize, imsave
import matplotlib.pyplot as plt
from numpy import matlib
import math
from scipy import stats
import imageio
from skimage.transform import resize
import skimage
import zlib, sys
import gzip
import matplotlib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

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
```

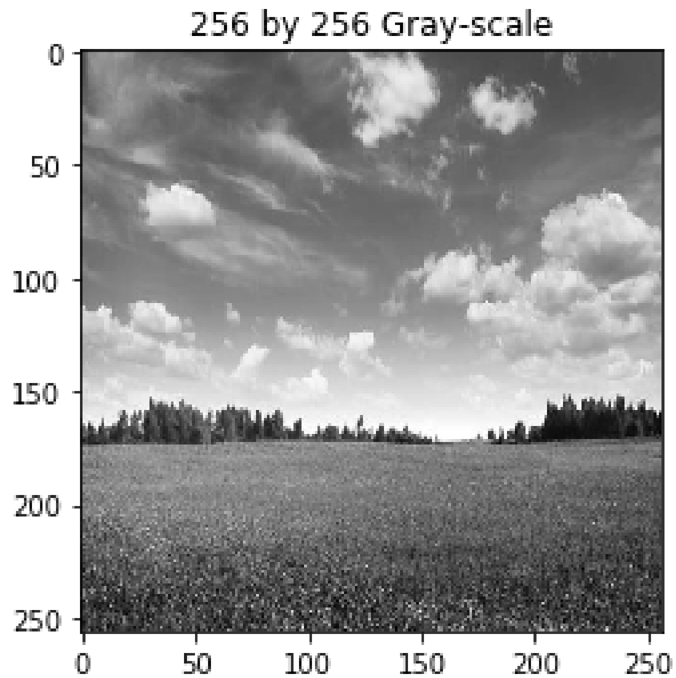
In [3]:
```
# reads in a jpeg image
A = imageio.imread('image.jpg')

# show the original image just read in
plt.imshow(A, cmap = plt.get_cmap('gray'))
plt.title("original image")
plt.show()
```



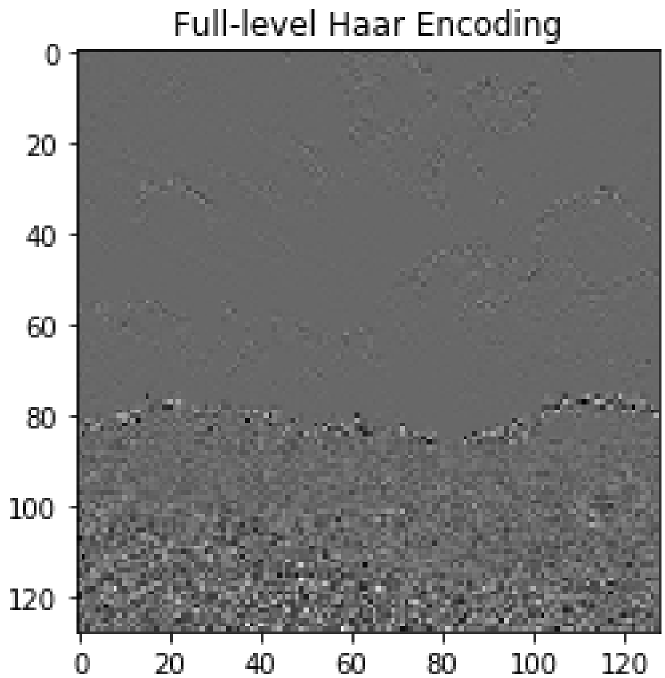
In [4]:
```
# resize the image(before apply gray scale function) as a 256 by 256 matrix
A = skimage.transform.resize(A, [256, 256], mode='constant')

#print(A)

# show the jpeg image in a figure
plt.imshow(A, cmap = plt.get_cmap('gray'))
plt.title("original image after resize")
plt.show()
```




```
In [5]: # Apply the rgb2gray function to the image
        A = rgb2gray(A)

        #print(np.amax(A))

        # show the jpeg image in a figure
        plt.imshow(A, cmap = plt.get_cmap('gray'))
        plt.title("256 by 256 Gray-scale")
        plt.show()
```



In [6]:
```
# make a deep copy of resize&gray-scale image
B = copy.deepcopy(A)

# set size to 256
N = 256

# Doing full-level Encoding (Forward Haar Transform)
for i in range(int(np.log2(N))):
    T,P = GetHaarMatrices(N)
    #print(T.shape)
    B[0:N, 0:N] = P*T*B[0:N, 0:N]*T.T*P.T
    N = int(N/2)

# show the result of full-level encoding
plt.figure()
plt.imshow(B[128:256,128:256], cmap = plt.get_cmap('gray'))
plt.title("Full-level Haar Encoding")
plt.show()

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




```
[[ 1.37032397e+02  8.28231535e-01  2.24150933e+00 ...  7.84313725e-03
  -3.92156863e-03 -3.37009804e-04]
 [ 4.66996372e+00 -4.49479731e+00 -1.76164275e+00 ...  7.84313725e-03
  -3.92156863e-03 -3.37009804e-04]
 [-6.79411587e+00 -1.14338176e+01 -6.27488108e-01 ...  3.92156863e-03
  -3.92156863e-03 -3.37009804e-04]
 ...
 [ 2.56199918e-02 -1.04924725e-01 -2.73541940e-02 ...  1.92135713e-02
   5.18132643e-02 -1.91346419e-02]
 [ 1.64313295e-01 -4.03217201e-02 -5.23184702e-02 ...  1.17415065e-02
   3.46326596e-02 -2.19951143e-02]
 [-6.83361694e-02 -7.27696756e-02 -1.01429493e-01 ... -4.42582886e-02
   1.48929418e-01 -7.86932466e-02]]
```

In [7]:
```
# create an empty numpy array record the sign of array
sign = np.empty([256,256])

# record the sign
for i in range(B.shape[0]):
    for j in range(B.shape[1]):
        if B[i][j]<=0:
            sign[i][j] = -1
        else:
            sign[i][j] = 1
```

```
        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]: # make 2 deep copy of B
        X = abs(copy.deepcopy(B))
        Y = copy.deepcopy(B)

        # convert X(2D numpy array) into 1D numpy array
        Y = Y.ravel()

        print(X)
```

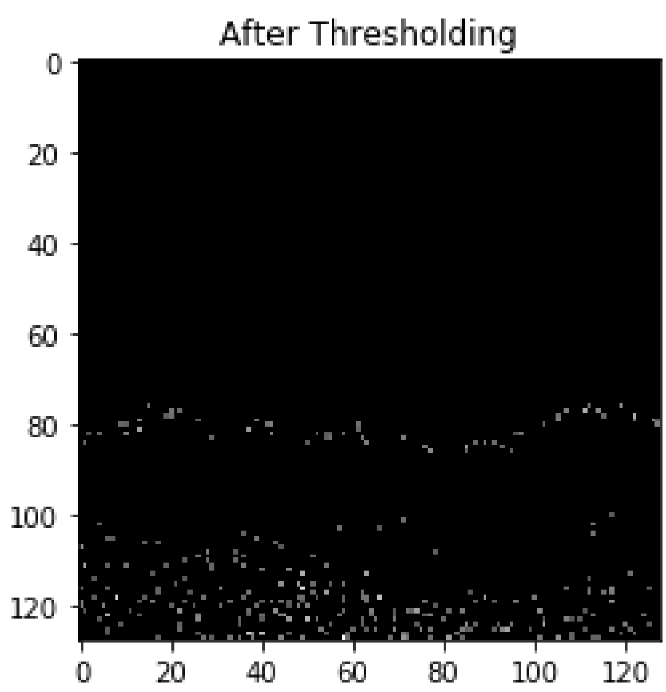
```
[[1.37032397e+02 8.28231535e-01 2.24150933e+00 ... 7.84313725e-03
  3.92156863e-03 3.37009804e-04]
 [4.66996372e+00 4.49479731e+00 1.76164275e+00 ... 7.84313725e-03
  3.92156863e-03 3.37009804e-04]
 [6.79411587e+00 1.14338176e+01 6.27488108e-01 ... 3.92156863e-03
  3.92156863e-03 3.37009804e-04]
 ...
 [2.56199918e-02 1.04924725e-01 2.73541940e-02 ... 1.92135713e-02
  5.18132643e-02 1.91346419e-02]
 [1.64313295e-01 4.03217201e-02 5.23184702e-02 ... 1.17415065e-02
  3.46326596e-02 2.19951143e-02]
 [6.83361694e-02 7.27696756e-02 1.01429493e-01 ... 4.42582886e-02
  1.48929418e-01 7.86932466e-02]]
```

```
In [9]: # make a deep copy to X to get the threshold but not affect X
        Z = copy.deepcopy(Y)

        # sort the numpy array by its absolute value
        Z = np.sort(abs(Z))

        # promopt to ask user what the top percent pixel will retain the same
        cutoff = input('How many percents of smallest elements you want to set to zero?')

        # define thresholding function to find the threshold
        def find_th(source, percentage):
            index = 0
            index = math.floor(len(source) * percentage / 100)
```

```
            threshold = source[index]
            return threshold
```

```
How many percents of smallest elements you want to set to zero?95
```

In [10]:
```
# apply the thresholding function to find the threshold th
th = find_th(Z, int(cutoff))
# print(th)

# implementation of the threshold process to numpy array X
for i in range(X.shape[0]):
    for j in range(X.shape[1]):

        if X[i][j] > th:
            continue
        else:
            X[i][j] = 0



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

# print the matrix out the make sure A apply to the threshold function correctly
print(X)
```



```
[[137.032397     0.82823154   2.24150933 ...   0.           0.
    0.        ]
 [  4.66996372   4.49479731   1.76164275 ...   0.           0.
    0.        ]
 [  6.79411587  11.43381759   0.62748811 ...   0.           0.
    0.        ]
 ...
 [  0.           0.           0.         ...   0.           0.
    0.        ]
 [  0.1643133    0.           0.         ...   0.           0.
    0.        ]
 [  0.           0.           0.         ...   0.           0.
    0.        ]]
```

In [11]:
```
# initialize the value to create proper partition and codebook
MX = np.amax(X)
bits = int(math.log2(X.shape[0]))
NP = 2**(bits-1)-1
c1 = 0
diff = (MX/th)**(1/NP)

# create empty list of partition and codebook
partition = []
codebook = [c1]
```

In [12]:
```
# create partition list
for n in range(NP):
    partition.append(th*(diff**n))

# print the length of partition list
print(len(partition))
```

```
127
```

In [13]:
```
# create codebook list
for n in range(NP-1):
    codebook.append(random.uniform(partition[n], partition[n+1]))
codebook.append(random.uniform(partition[len(partition)-1], partition[len(partition)-

# print the length of the codebook list
print(len(codebook))
```

```
128
```

In [14]:
```
# convert M(2D numpy array) into 1D list as signal
signal = []

for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        signal.append(X[i][j])
```

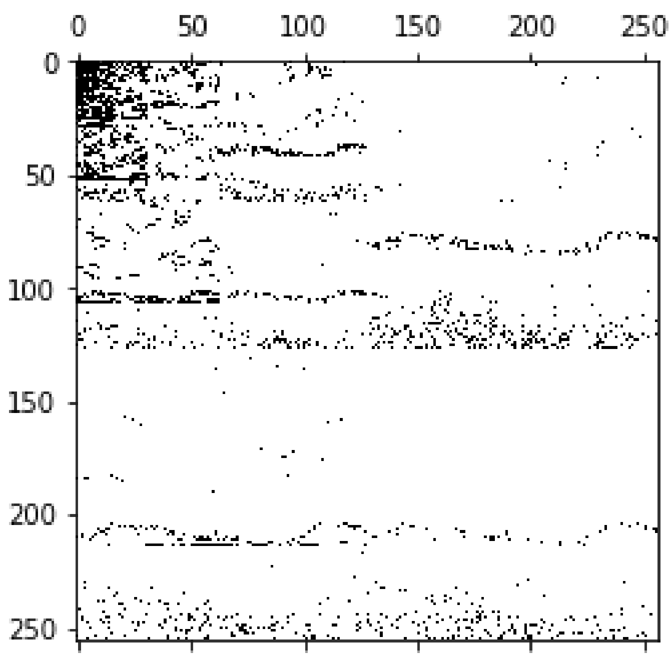
In [15]:
```
# define a function to do quantization
def quantiz(signal, partition, codebook):
    indices = []
    quanta = []

    for data in signal:
        index = 0
        while index<len(partition) and data>partition[index]:
            index += 1
        indices.append(index)
        quanta.append(codebook[index])

    return indices, quanta

# call the quantiz function to get indices and quantized signal list
indices, quanta = quantiz(signal, partition, codebook)

# reshape quantized signal into 2D array
quanta = np.reshape(quanta, (256,256))

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 [16]:
```
# reshape the indices into 2D array
indices = np.reshape(indices, (256,256))
print(indices)
print(type(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]]
<class 'numpy.ndarray'>
```

In [17]:
```
# make a deep copy of image after threshholding
M = copy.deepcopy(quanta)

def log_quantiz(inp):
    for i in range(inp.shape[0]):
        for j in range(inp.shape[1]):

            if inp[i][j] == 0:
                continue
            else:
                inp[i][j] = math.log10(inp[i][j])

log_quantiz(M)

# show the image after apply to log quantization
plt.spy(M)
plt.show()

print(M)
```



```
[[ 2.11593715 -0.10009545  0.34204973 ...  0.          0.
   0.        ]
 [ 0.66600766  0.64871415  0.24776342 ...  0.          0.
   0.        ]
 [ 0.82726732  1.06747433 -0.20330214 ...  0.          0.
   0.        ]
 ...
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [-0.78086325  0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]]
```

```
In [18]: # make a copy of image after thresholding and log_quantiz as N
         N = copy.deepcopy(indices)

         # start of the lossless compression by using zlib
         compressed_data = zlib.compress(N, 9)
         #print(compressed_data)
         compress_ratio = float(sys.getsizeof(compressed_data))/sys.getsizeof(N)

         # print out the percent of lossless compression
         print("compress_ratio:", compress_ratio * 100, "%")
```

```
compress_ratio: 1.492372234935164 %
```

```
In [19]: # create a file to save the compressed data
         f = gzip.open('compressed_data.txt.gz', 'wb')
         f.write(compressed_data)
         f.close()
```

```
In [20]: # create a file to save the sign np array
         f2 = open('sign.txt', 'wb+')
         f2.write(sign)
         f2.close()
```

```
In [21]: # create a file to save the codebook list
         f1 = open('codebook.txt', 'w')
         for item in codebook:
             f1.write("%s\n" % item)
         f1.close()
```