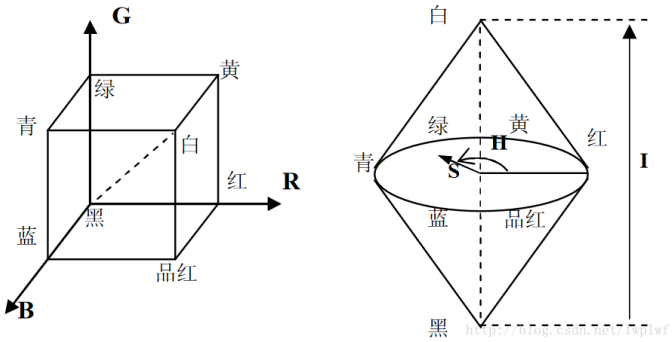
2.1 Preprocessing

HSI 模型

RGB系统与人眼强烈感知红绿蓝三原色的事实能很好地匹配。但RGB模型和CMY/CMYK模型不能很好地适应实际上人解释的颜色。所有引出HIS模型。

HSI (Hue, Saturation, Intensity) 模型是从人的视觉系统出发，用 **色调(Hue)、色饱和度(Saturation)和亮度(Intensity)** 来描述色彩。色调是描述纯色(纯黄色、纯橙色或纯红色)的颜色属性。饱和度是一种纯色被白光稀释的程度的度量。亮度是一个主观描述子，体现无色的强度概念。

RGB向HSI模型的转换是由一个基于笛卡尔直角坐标系的单位立方体向基于圆柱极坐标的双锥体的转换。基本要求是将RGB中的亮度因素分离，通常将色调和饱和度统称为色度，用来表示颜色的类别与深浅程度。在图中圆锥中间的横截面圆就是色度圆，而圆锥向上或向下延伸的便是亮度分量的表示。



从RGB到HSI的彩色转换

从RGB空间到HSI空间的转换有多种方法，这里仅说明最为经典的几何推导法。RGB与HSI之间的转换关系为：

$$\mathrm{R,G,B} \in [0,255]$$

$$R' = \frac{R}{255}$$

$$G' = \frac{G}{255}$$

$$B' = \frac{B}{255}$$

$$H = \begin{cases} \theta, & G' \geq B' \\ 2\pi - \theta, & G' < B' \end{cases}$$

$$S = 1 - \frac{3\min(R', G', B')}{R' + G' + B'}$$

$$I = \frac{1}{3}(R' + G' + B')$$

$$\theta = \cos^{-1}\left[\frac{(R'-G')+(R'-B')}{2\sqrt{(R'-G')^2+(R'-B')(G'-B')}}\right]$$

参考代码 (https://cloud.tencent.com/developer/article/1451369)

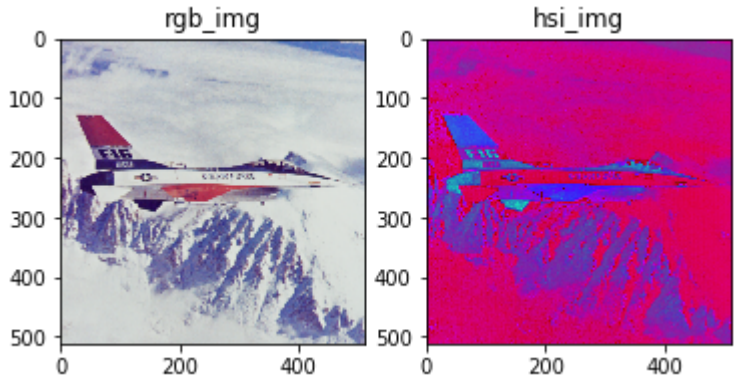
In [47]:

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt
import sys

ColorSize = 256
d = 1
pi = 3.1415926
S = 512
def rgb2hsi(rgb_lwpImg):
    np.seterr(divide='ignore',invalid='ignore')
    #rows,cols,channels = rgb_lwpImg.shape
    rgb_lwpImg = rgb_lwpImg.astype(np.float32)
    # 归一化到[0,1]
    img_bgr = rgb_lwpImg.copy()/255
    b, g, r = cv2.split(img_bgr)

    Tdata = np.where((2*np.sqrt((r-g)**2+(r-b)*(g-b))) != 0,np.arccos((2*r-b-g)/
    Hdata = np.where(g >= b,Tdata,2*pi-Tdata)
    Hdata = Hdata / (2*pi)
    Sdata = np.where((b+g+r) != 0, 1 - 3*np.minimum(b,g,r)/(b+g+r),0)
    Idata = (b+g+r)/3

    # 输出HSI图像，扩充到255以方便显示，一般H分量在[0,2pi]之间，S和I在[0,1]之间
    hsi_lwpImg = rgb_lwpImg.copy()
    hsi_lwpImg[:,:,0] = Hdata*255
    hsi_lwpImg[:,:,1] = Sdata*255
    hsi_lwpImg[:,:,2] = Idata*255

    return hsi_lwpImg

if __name__ == '__main__':
    inpath = "DataSet/Standard_benchmark_image/Airplane.tiff"
    rgb_lwpImg = cv2.imread(inpath)
    hsi_lwpImg = rgb2hsi(rgb_lwpImg)
    hsi_lwpImg = np.array(hsi_lwpImg)
    hsi_lwpImg = hsi_lwpImg.astype(np.uint8)

    rgb_lwpImg = rgb_lwpImg[:,:,::-1]
    hsi_lwpImg = hsi_lwpImg[:,:,::-1]

    plt.subplot(1,2,1)
    plt.title("rgb_img")
    plt.imshow(rgb_lwpImg)
    plt.subplot(1,2,2)
    plt.title("hsi_img")
    plt.imshow(hsi_lwpImg)


# 提取I分量作为输入图像
inpath = "DataSet/Standard_benchmark_image/Airplane.tiff"
rgb_lwpImg = cv2.imread(inpath)
rgb_lwpImg = cv2.resize(rgb_lwpImg,(512,512))
hsi_lwpImg = rgb2hsi(rgb_lwpImg)
input_img = np.zeros([hsi_lwpImg.shape[0], hsi_lwpImg.shape[1], 1],dtype=np.uint
for i in range(hsi_lwpImg.shape[0]):
    for j in range(hsi_lwpImg.shape[1]):
        input_img[i,j,0] = hsi_lwpImg[i,j,2]
print(input_img.shape)
```

```
60 print(input_img[:2])
61
```

executed in 548ms, finished 18:11:39 2019-12-15

```
(512, 512, 1)
[[[ 32]
  [203]
  [200]
  ...
  [179]
  [176]
  [163]]

 [[ 36]
  [196]
  [192]
  ...
  [189]
  [168]
  [158]]]
```

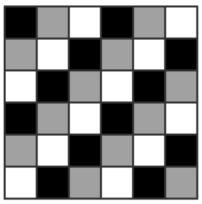


2.2 Feature Extraction

2.2.1 Global Statistical Feature.

GLCM

Here is an example[5] of GLCM. This is an image with only three grayscale level, i.e., 0, 1, 2.



0	1	2	0	1	2
1	2	0	1	2	0
2	0	1	2	0	1
0	1	2	0	1	2
1	2	0	1	2	0
2	0	1	2	0	1

Figure 1: An image with three grayscale level

Its Co-occurrence Matrix, with $(d, \theta) = (1, 0°)$, is described below.

$$P(x,y)_{(d,\theta)} = \begin{bmatrix} 0 & 10 & 10 \\ 10 & 0 & 10 \\ 10 & 10 & 0 \end{bmatrix}, \ where\ (d,\theta) = (1,0°) \qquad (4)$$

参考代码网站 (https://blog.csdn.net/panda1234lee/article/details/71405880)

In [3]:

```python
# 图像矩阵原点在左上角，所以di,dj 会有点绕
# 此处p矩阵的计算做了变种，没有完全按公式来，要不时间复杂度将是0(256*256*0(512*512)) = 15
def get_P_Matrix(ColorSize, d, theta):
    '''GLCM计算

    :param x,y: 为0-255的像素值
    :param d: 距离
    :param theta: 方向

    :return p:归一化的P矩阵
    :rtype ndarray(256*256)
    '''
    di,dj = 0,0

    if theta == 0:
        dj = d
    elif theta == 45:
        di,dj = -d, d
    elif theta == 90:
        di = -d
    elif theta == 135:
        di,dj = -d, -d

    rows,cols, channels = input_img.shape
    assert rows == 512
    assert cols == 512
    assert channels == 1
    p = np.zeros((ColorSize,ColorSize))

    for i in range(rows):
        if i + di < 0 or i + di >= rows:
            continue
        for j in range(cols):
            if j + dj < 0 or j + dj >= cols:
                continue

            pixelx = input_img[i][j][0]
            pixely = input_img[i+di][j+dj][0]
            p[pixelx,pixely] += 1

    p = p/p.sum()
    return p

p = get_P_Matrix(ColorSize, d, 0)
print('p.sum():',p.sum())
print(p.shape,p)

```

executed in 350ms, finished 12:15:32 2019-12-15

```
p.sum(): 1.0
(256, 256) [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]]
```

In [4]:

```
# 测试用的矩阵

# input_img = [
#     [0,1,2,0,1,2],
#     [1,2,0,1,2,0],
#     [2,0,1,2,0,1],
#     [0,1,2,0,1,2],
#     [1,2,0,1,2,0],
#     [2,0,1,2,0,1],
# ]
# ColorSize = 3

# p = np.zeros((ColorSize,ColorSize))

# p = caculate_P_Matrix(p, d, theta)

# print(p)

```

executed in 1ms, finished 12:15:32 2019-12-15

(1)计算 contrast

In [5]:

```
def get_Contrast(p):
    '''
    :param p: P矩阵

    :return contrast:对比度
    :rtype float
    '''
    contrast = 0

    for x in range(len(p)):
        for y in range(len(p[0])):
            contrast += (x-y)**2 * p[x,y]

    return contrast

contrast = get_Contrast(p)
print('contrast:', contrast)
```

executed in 54ms, finished 12:15:38 2019-12-15

```
contrast: 147.20923663772498
```

(2) 计算correlation

因为 $\boldsymbol{G}$ 取决于 Q，所以选择一个合适的位置算子并分析 $\boldsymbol{G}$ 的元素，可以检测灰度纹理模式的存在。

表 11.3 中列出了一组表征 $\boldsymbol{G}$ 的内容的有用描绘子。(表中第二行的)相关描绘子中使用的量定义如下：

$$m_r = \sum_{i=1}^{K} i \sum_{j=1}^{K} p_{ij}, \qquad m_c = \sum_{j=1}^{K} j \sum_{i=1}^{K} p_{ij}$$

和

$$\sigma_r^2 = \sum_{i=1}^{K} (i - m_r)^2 \sum_{j=1}^{K} p_{ij}, \qquad \sigma_c^2 = \sum_{j=1}^{K} (j - m_c)^2 \sum_{i=1}^{K} p_{ij}$$

如果我们令

$$P(i) = \sum_{j=1}^{K} p_{ij} \quad 和 \quad P(j) = \sum_{i=1}^{K} p_{ij}$$

则前述公式可以写为

$$m_r = \sum_{i=1}^{K} iP(i), \qquad m_c = \sum_{j=1}^{K} jP(j), \qquad \sigma_r^2 = \sum_{i=1}^{K} (i - m_r)^2 P(i), \qquad \sigma_c^2 = \sum_{j=1}^{K} (j - m_c)^2 P(j)$$

参考式(11.3-4)、式(11.3-5)和它们的解释，我们看到 m_r 是沿归一化后的 $\boldsymbol{G}$ 的行计算的均值，m_c 是沿着 $\boldsymbol{G}$ 的列计算的均值。类似地，σ_r 和 σ_c 是分别沿行和列计算的标准差(方差的平方根)。这些量都是标量，与 $\boldsymbol{G}$ 的大小无关。

In [31]:

```
def get_Correlation(p):
    '''
    :param p:P矩阵

    :return correlation: 相关度
    :rtype float
    '''
    correlation = 0

    px = np.sum(p,axis = 1)
    py = np.sum(p,axis = 0)

    x = np.array(range(len(p)))
    ux = np.dot(x,px)

    y = np.array(range(len(p[0])))
    uy = np.dot(y,py)

    sigmax = np.dot((x-ux)**2,px)
    sigmax = np.sqrt(sigmax)

    sigmay = np.dot((y-uy)**2,py)
    sigmay = np.sqrt(sigmay)

    for x in range(len(p)):
        for y in range(len(p[0])):
            if (sigmax * sigmay)== 0:
                continue
            correlation += ( (x-ux)*(y-uy)*p[x,y] )/(sigmax * sigmay)
            #print(correlation)

    return correlation

correlation = get_Correlation(p)
print('\n correlation:', correlation)
```

executed in 89ms, finished 15:50:59 2019-12-15

```
correlation: 0.9627860907607861
```

(3) 计算energy

In [32]:

```
def get_Energy(p):
    '''
    :param p:P矩阵

    :return energy:能量
    :rtype float
    '''
    energy = 0

    energy = np.sum(p.copy()**2)
    return energy

energy = get_Energy(p)
print('energy:',energy)
```

executed in 4ms, finished 15:51:03 2019-12-15

```
energy: 0.001992181874979281
```

(4) 计算homegeneity

In [33]:

```
def get_Homegeneity(p):
    '''
    :param p:P矩阵

    :return homegeneity:同质性
    :rtype float
    '''
    homegeneity = 0
    for x in range(len(p)):
        for y in range(len(p[0])):
            homegeneity += p[x,y]/(1+(x-y)**2)

    return homegeneity

homegeneity = get_Homegeneity(p)
print('homegeneity:',homegeneity)
```

executed in 58ms, finished 15:51:05 2019-12-15

```
homegeneity: 0.30743620915012376
```

获取 textures features T, 长度为16

In [34]:

```
def get_T(ColorSize, d):
    '''
    :param ColorSize:颜色量级
    :param d: 距离d

    :return T: theta分别从0到135，4个上述特征的集合
    :rtype list
    '''
    T = []
    for theta in [0, 45, 90, 135]:
        p = get_P_Matrix(ColorSize, d, theta)
        contrast = get_Contrast(p)
        T.append(contrast)
        correlation = get_Correlation(p)
        T.append(correlation)
        energy = get_Energy(p)
        T.append(energy)
        homegeneity = get_Homegeneity(p)
        T.append(homegeneity)

    return T

d = 1
T = get_T(256, d)
assert len(T) == 16
print('纵轴为 theta = 0, 45, 90, 135')
print('横轴为 contrast, correlation, energy, homegeneity')
for i in range(4):
    print(T[4*i:4*i+4])


```

executed in 1.96s, finished 15:51:08 2019-12-15

```
纵轴为 theta = 0, 45, 90, 135
横轴为 contrast, correlation, energy, homegeneity
[147.20923663772498, 0.9627860907607861, 0.001992181874979281, 0.30743
620915012376]
[287.3731450170599, 0.9271206404327745, 0.001709018847782496, 0.271285
2793806977]
[159.4591257950134, 0.9597326018324571, 0.0020697479070986694, 0.31201
8091812484]
[272.0873388199341, 0.9308867009738515, 0.001770197206244798, 0.275572
6301366319]
```

2.2.2 Local Invariant Feature

2维DCT 变换

〇、DCT的历史与背景

1807年，法国数学家、物理学家傅里叶（Jean Baptiste Joseph Fourier）提出了傅里叶变换（Fourier Transform, FT）。傅里叶变换的形式有很多种，归一化的二维离散傅里叶变换（Discrete Fourier transform, DFT）可以写成如下形式：

$$F(u,v)=\frac{1}{\sqrt{NM}}\sum_{x=0}^{N-1}\sum_{y=0}^{M-1}f(x,y)e^{-\frac{2\pi i}{N}ux}e^{-\frac{2\pi i}{M}vy}$$

$$f(x,y)=\frac{1}{\sqrt{NM}}\sum_{u=0}^{N-1}\sum_{v=0}^{M-1}F(u,v)e^{\frac{2\pi i}{N}ux}e^{\frac{2\pi i}{M}vy}$$

傅里叶变换包含复数运算，其运算复杂度和存储长度都超过实数运算。为了简化上述过程，同时达到更好的变换效果，余弦变换应运而生。

从傅里叶变换到离散余弦变换，需要一些数学理论的支持。在给定区间内满足狄利赫里条件的连续实对称函数，可以展开成仅含有余弦项的傅里叶级数。

对于定义在正实数域上的函数，可以通过偶延拓或奇延拓，满足上述条件。但如果函数的定义域包含零点，情况则稍有些复杂。

以一个二维离散函数 $f(x,y)(x,y=0,1,\dots,N-1)$ 为例，对其进行偶延拓。

假如序列中不包含零点，自然按照以下方式延拓：

$f(1,0)=f(-1,0)$ 、 $f(0,1)=f(0,-1)$ ，对称中心为 $(0,0)$

由于序列中包括零点，考虑零点后的延拓方式：

$f(0,0)=f(-1,0)$ 、 $f(0,0)=f(0,-1)$ ，对称中心为 $(-\frac{1}{2},-\frac{1}{2})$

因此，按照上述方法延拓后，归一化的二维离散余弦变换可以写成如下形式：

$when\ (x,y)\ or\ (u,v)=(0,0)$

$$F(u,v)=\frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)cos[\frac{\pi}{N}u(x+\frac{1}{2})]cos[\frac{\pi}{N}v(y+\frac{1}{2})]$$

$$f(x,y)=\frac{1}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}F(u,v)cos[\frac{\pi}{N}u(x+\frac{1}{2})]cos[\frac{\pi}{N}v(y+\frac{1}{2})]$$

$when\ (x,y)\ or\ (u,v)\neq(0,0)$

$$F(u,v)=\frac{1}{2N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)cos[\frac{\pi}{N}u(x+\frac{1}{2})]cos[\frac{\pi}{N}v(y+\frac{1}{2})]$$

$$f(x,y)=\frac{1}{2N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}F(u,v)cos[\frac{\pi}{N}u(x+\frac{1}{2})]cos[\frac{\pi}{N}v(y+\frac{1}{2})]$$

在傅里叶变换中，正逆变换的变换核 $e^{-\frac{2\pi i}{N}ux}e^{-\frac{2\pi i}{M}vy}$ 与 $e^{\frac{2\pi i}{N}ux}e^{\frac{2\pi i}{M}vy}$ 相差一个负号；相应的，在余弦变换中，正逆变换的变换核 $cos[\frac{\pi}{N}u(x+\frac{1}{2})]cos[\frac{\pi}{N}v(y+\frac{1}{2})]$ 也应相差一个负号。由于 $cos(x)=cos(-x)$ ，所以余弦变换的正逆变换在形式上具有一致性。

顺带一提，在某些教材上，将上述形式的离散余弦变换写成类似于下面的样子：

$$F(u,v)=\frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)cos[\frac{\pi u(2x+1)}{2N}]cos[\frac{\pi v(2y+1)}{2N}]$$

也就是将 $\frac{1}{2}$ 乘入了整个分式中。虽然这样写只是变了一种形式，不影响结果，但破坏了原本的几何意义（或者说偶延拓过程），谁知道这个 $(2x+1)$ 对应着什么东西。

由 Xnip 截图

也就是说，JPEG算法压缩分下面几个步骤：

1，将图像分块，分成8x8的小块。图像尺寸不够分怎么办？见下图

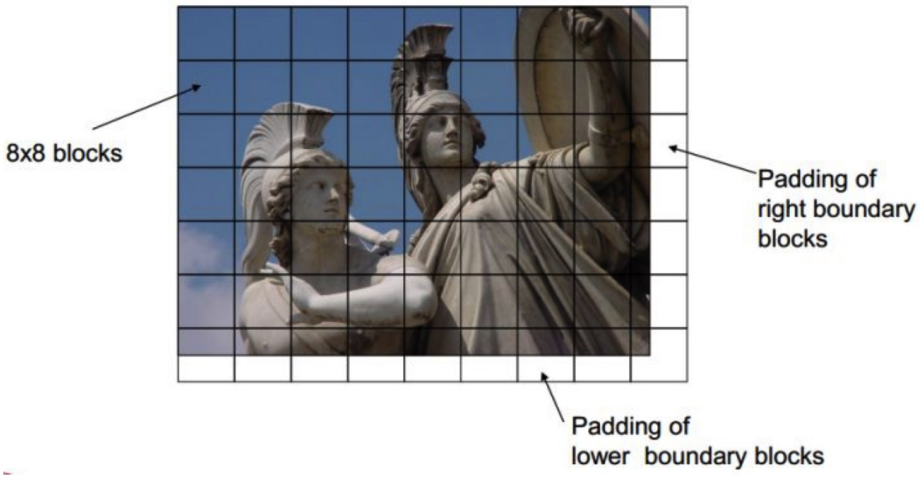


FIG 2

2，对每一个8x8的小块进行DCT变换。什么是DCT变换？就是下图这个玩意。

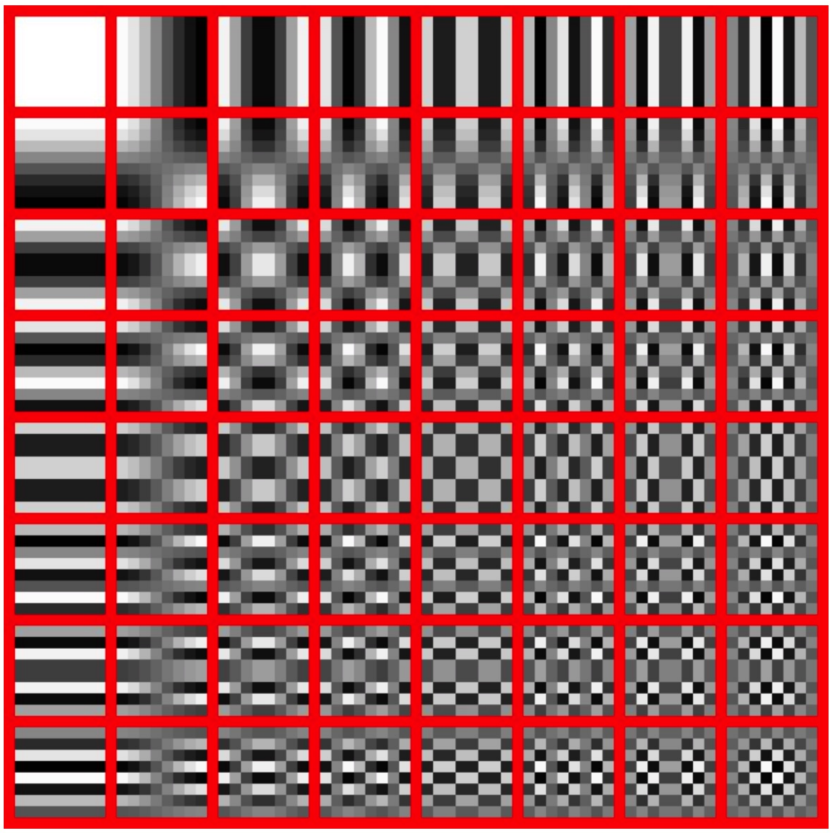


FIG 3 DCT transform

DCT变换到底怎么进行？其实很简单，这个DCT变换不是有8x8=64个小块吗(专业术语叫basis，变换基，或者是DCT filter)，将DCT的每个小块和图像的8x8的小块乘一下。乘一下又是什么意思？乘一下就是将两个8x8的小块对应位置的值相乘，然后再累加起来。这样经过这个DCT变换后，原来的8x8的图像块，就变成8x8的DCT系数块。

参考代码 (https://blog.csdn.net/leviopku/article/details/86512468)

将图像分成s*s的块，s = 64

In [35]:

```python
# 将图像分成s*s的块 , s = 64
def get_B(input_img, s = 64):
    '''
    :param input_img: 输入图像
    :param s: 输入图像分割成好多个块，每个块是s*s的像素

    :return B:分割后并索引好的块矩阵
    :rtype list
    '''
    S = input_img.shape[0]
    row = S//s
    col = row
    N = (S//s)**2
    B = [0]*N
    for i in range(N):
        pos_x = (i//row)*s
        pos_y = (i%row)*s
        B[i] = input_img[pos_x:pos_x+s, pos_y:pos_y+s, 0]
        assert B[i].shape == (64,64)

    return B

B = get_B(input_img, 64)
print(B[:3])

```

executed in 7ms, finished 15:51:14 2019-12-15

```
[array([[ 32, 203, 200, ..., 200, 192, 184],
       [ 36, 196, 192, ..., 199, 199, 194],
       [ 36, 198, 195, ..., 200, 199, 195],
       ...,
       [ 68, 188, 186, ..., 135, 143, 137],
       [ 67, 192, 190, ..., 157, 149, 144],
       [ 68, 191, 187, ..., 172, 167, 154]], dtype=uint8), array([[19
0, 186, 192, ..., 196, 195, 198],
       [203, 204, 202, ..., 201, 201, 198],
       [201, 206, 196, ..., 202, 197, 199],
       ...,
       [142, 146, 143, ..., 204, 206, 205],
       [135, 139, 152, ..., 207, 204, 204],
       [151, 150, 158, ..., 203, 202, 203]], dtype=uint8), array([[19
4, 193, 198, ..., 191, 182, 176],
       [199, 197, 199, ..., 195, 192, 190],
       [198, 199, 199, ..., 194, 191, 188],
       ...,
       [205, 203, 199, ..., 203, 204, 204],
       [204, 203, 205, ..., 199, 200, 204],
       [206, 205, 202, ..., 199, 199, 203]], dtype=uint8)]
```

In [36]:

```python
# DCT 变换
def DCT_transformation(img_src):
    '''
    :param img_src:源图像

    :return img_dct:dct变换后的图像
    :rtype ndarray(64*64)
    '''
    img_src = np.float32(img_src) # 将数值精度调整为32位浮点型
    img_dct = cv2.dct(img_src)  # 使用dct获得img的频域图像
    return img_dct

def get_C_Matrix(N,B):
    '''
    :param N:分割的块数
    :param B:分割后的块矩阵

    :return C:DCT变换后的C矩阵
    :rtype list
    '''
    C = [0]*N
    for i in range(N):
        C[i] = DCT_transformation(B[i])
    return C

C = get_C_Matrix(64,B)
print(len(C),C)
```

executed in 37ms, finished 15:51:15 2019-12-15

```
64 [array([[ 1.1436594e+04, -6.9229439e+01, -4.6804041e+02, ...,
        -1.9536160e+01, -1.6190327e+01, -6.1480370e+00],
       [ 3.2544250e+02, -3.8117340e+02,  2.2608145e+02, ...,
         5.1252937e-01,  3.1172007e-01,  3.5299952e+00],
       [-1.4785325e+01,  1.0341821e+02,  3.9871426e+01, ...,
         1.5720841e+00,  1.4887179e+00, -4.6615763e+00],
       ...,
       [-1.3220803e+00, -1.4210336e+00,  6.9718871e+00, ...,
        -8.2511342e-01,  1.3121440e-02, -7.7966744e-01],
       [-6.3605404e-01, -1.7329650e+00,  1.4384527e+00, ...,
         5.6356770e-01, -1.0224813e+00,  3.1182399e-01],
       [-7.7776256e+00,  3.2479398e+00, -2.5679572e+00, ...,
         8.8504992e-02, -3.7192351e-01, -3.9989060e-01]], dtype=float32), array([[ 1.2079703e+04, -5.5722717e+02, -2.6907330e+02, ...,
        -4.0467267e+00, -1.9013501e+00, -2.0324180e+00],
       [ 3.3670987e+02,  3.3355212e+02,  3.1069885e+02, ...,
         2.4608147e+00,  3.4561257e+00, -2.0584756e-01],
       [ 3.1569907e+02,  8.6808815e+01, -9.3518211e+01, ...,
         3.4306242e+00,  2.1274712e+00,  1.9516854e+00],
```

In [37]:

```python
# 获取Q向量
def get_Q(C,s,N):
    '''
    :param C: DCT变换后的C矩阵
    :param s: s*s 的小块
    :param N: 分割的块数

    :return Q: C[0,1]~C[0,n],C[1,0]~C[n,0] 组合后的向量
    :rtype ndarray(64*64)
    '''
    n = s//2
    Q = [0]*N
    for i in range(N):
        Q[i] = []
        for v in range(n):
            Q[i].append(C[i][0,v])
        for u in range(n):
            Q[i].append(C[i][u,0])

        Q[i] = np.array(Q[i], dtype = 'float32')
    Q = np.array(Q, dtype = 'float32').T
    return Q

Q = get_Q(C,64,64)
print(Q.shape,Q[:1],'\n',Q[1:2])
print(Q)


```

executed in 10ms, finished 15:51:16 2019-12-15

```
(64, 64) [[11436.594  12079.703  12706.031  12867.094  12715.703  1251
2.031
  11785.094  11628.656  12797.594  13010.3125 12746.141  12478.609
  12554.219  12838.9375 13139.375  13502.016   9737.656   8581.875
  12860.359  12610.25   12955.031  13469.5625 13587.172  13634.109
  10535.219   8068.578  10283.531  11115.6875 10299.453   9847.297
  11378.547  12560.75    8125.1562  9885.5625  8830.766   9744.953
   9448.266  11925.5    13734.422  14133.344  11329.781  10162.641
   9430.5625  8981.      8870.891   9534.781  12663.172  13966.906
  13274.8125 13303.344  11300.406   9747.734   9968.672  11076.016
  10286.125  13809.609  13117.734  13301.953  13069.453  12442.797
  12391.469  11903.6875  9139.828  10537.641 ]]
 [[  -69.22944   -557.2272    -111.61838     63.81199    -40.55819
    155.96301    160.45898     37.481407  -206.31413     49.015724
     69.42781    109.45979    -66.59932   -170.3013     -37.977623
    -94.461044  3242.0269   -2792.6118      24.214376   110.608086
   -151.35892   -195.99428     34.96428    -89.21649   1306.0708
  -1921.4017    -813.7635     201.0301     175.49927    -37.52925
  -1048.3586     541.34534    272.37335   -319.61334   -851.4734
    380.15616   -547.16797   -671.72534   -234.82874    -37.41033
   1110.687     -928.95215    435.27094   1314.864     -819.55566
    -84.230194 -1119.8943      58.995285  -239.83493     75.96721
   1098.6875     179.49614   -734.9083    -144.71384  -2118.8384
   -222.51707   -280.46454    -22.813892   112.11476    187.34172
   -194.93637    756.3769    -582.0397    -900.1459   ]]
[[ 1.14365938e+04  1.20797031e+04  1.27060312e+04 ...  1.19036875e+04
   9.13982812e+03  1.05376406e+04]
 [-6.92294388e+01 -5.57227173e+02 -1.11618378e+02 ...  7.56376892e+02
  -5.82039673e+02 -9.00145874e+02]
```

```
[-4.68040405e+02 -2.69073303e+02  4.68123474e+01 ... -7.29051880e+02
  8.19798828e+02 -1.03562820e+02]
...
[-2.20563722e+00 -2.46528077e+00 -4.59560108e+00 ...  1.49269135e+02
  1.31245743e+02  1.00891609e+02]
[ 3.55919576e+00 -4.76156998e+00  4.87274826e-01 ... -1.40216507e+02
 -1.09810196e+02 -8.45682526e+01]
[-5.68729734e+00 -4.77082300e+00 -5.06000400e-01 ...  1.45449341e+02
  1.21259781e+02  8.61435242e+01]]
```

In [38]:

```python
#获取U向量
def get_U(Q,N):
    '''
    :param Q: C[0,1]~C[0,n],C[1,0]~C[n,0] 组合后的向量
    :param N: 分割后的块数

    :return U: 归一化Q
    :rtype ndarray(64*64)
    '''
    # 归一化Q
    U = np.zeros(Q.shape)
    for i in range(N):
        amax = Q[i,:].max()
        amin = Q[i,:].min()
        U[i] = (Q[i] - amin) / (amax-amin)

    return U

U = get_U(Q,64)
print(U.shape,U)
```

executed in 8ms, finished 15:51:19 2019-12-15

```
(64, 64) [[0.55534142 0.66138172 0.76465493 ... 0.63235903 0.17663503
0.40711591]
 [0.45129171 0.37042561 0.44426742 ... 0.58810294 0.36631393 0.3136005
4]
 [0.34661782 0.43507963 0.57552403 ... 0.23057075 0.91919786 0.5086664
6]
 ...
 [0.42678356 0.42584631 0.4181563  ... 0.97357512 0.90851456 0.7989426
3]
 [0.65495867 0.62133718 0.64254606 ... 0.07400721 0.19686936 0.2988639
5]
 [0.43432391 0.43755582 0.45259565 ... 0.96730471 0.88200063 0.7581637
5]]
```

U0是每一行的均值，U的其他不变 用U的每一列 - 均值

In [39]:

```python
# 计算U向量内部的欧式距离

def caculate_Euclidean_Distance(U):
    '''
    :param U: 归一化的Q

    :return d: U向量内部的欧式距离
    :rtype list
    '''
    # U0是每一行的均值，U的其他不变，U0.shape = (64,1)
    U0 = np.mean(U, axis = 1).reshape(U.shape[0],1)
    assert U0.shape == (64,1)
    d = [0]*len(U)
    for i in range(len(U)):
        d[i] = np.sqrt( np.sum( (U[:,i] - U0)**2 ) )

    return d

D = caculate_Euclidean_Distance(U)
print('len(D):',len(D))
for i in range(0,len(D),5):
    print(D[i:i+5])

```

executed in 9ms, finished 15:51:21 2019-12-15

```
len(D): 64
[18.94645629596479, 13.054981009855247, 12.602056363637757, 12.812774658322352, 12.737493653540533]
[12.873516284448936, 12.918163045320187, 13.377520936106256, 19.57052570464037, 12.822181874564215]
[12.756306720151747, 12.620398723521584, 13.195669525801835, 12.812259531672224, 12.842043992762266]
[13.180828006218215, 21.697565498469494, 18.94184791932066, 12.762556504046 86, 12.655713730268323]
[12.894570986862021, 13.130698079583867, 12.99959498569281, 13.241572632384 61, 18.080325038533783]
[20.621630671621784, 17.674240422085152, 20.4223810373219, 18.428017698422355, 18.370191442532953]
[16.922047634261624, 13.212071788136434, 19.786531680286732, 16.015749198354367, 17.980453309360442]
[16.815246357166608, 16.068461432549647, 15.242293664965832, 13.49613904083496, 13.482330902558596]
[20.98223782280425, 17.398455513529996, 16.42786962022209, 14.554667068104749, 16.737263006852096]
[15.16842421930363, 15.735525955404666, 13.450133964723387, 20.015909770189833, 13.07667032890979]
[13.621739999613657, 13.511207044159454, 14.898078732985748, 16.0459808750653, 15.369370389987422]
[13.121097818908407, 24.718052924057567, 19.446768325218056, 18.165346685441186, 17.32389263569356]
[19.472465032059585, 19.575050528545635, 18.867434456684546, 17.515829750776074]
```

2.3 Quantization

In [40]:

```python
# 将T和D 特征向量 组合成H1
def get_H1(T,D):
    '''
    :param T: 16个Global Statistical Feature 特征
    :param D: 64个Local Invariant Feature 特征

    :return H1: [T D]
    :rtype list
    '''
    H1 = [] + T + D

    return H1


H1 = get_H1(T,D)
print('len(H1):',len(H1))
for i in range(0,len(H1),5):
    print(H1[i:i+5])
```

executed in 5ms, finished 15:51:24 2019-12-15

```
<class 'list'> <class 'list'>
len(H1): 80
[147.20923663772498, 0.9627860907607861, 0.001992181874979281, 0.30743
620915012376, 287.3731450170599]
[0.9271206404327745, 0.001709018847782496, 0.2712852793806977, 159.459
1257950134, 0.9597326018324571]
[0.0020697479070986694, 0.312018091812484, 272.0873388199341, 0.930886
7009738515, 0.0017701972062447 98]
[0.2755726301366319, 18.94645629596479, 13.054981009855247, 12.6020563
63637757, 12.812774658322352]
[12.737493653540533, 12.873516284448936, 12.918163045320187, 13.377520
936106256, 19.57052570464037]
[12.822181874564215, 12.756306720151747, 12.620398723521584, 13.195669
525801835, 12.812259531672224]
[12.842043992762266, 13.180828006218215, 21.697565498469494, 18.941847
91932066, 12.76255650404686]
[12.655713730268323, 12.894570986862021, 13.130698079583867, 12.999594
98569281, 13.24157263238461]
[18.080325038533783, 20.621630671621784, 17.674240422085152, 20.422381
0373219, 18.428017698422355]
[18.370191442532953, 16.922047634261624, 13.212071788136434, 19.786531
680286732, 16.015749198354367]
[17.980453309360442, 16.815246357166608, 16.068461432549647, 15.242293
664965832, 13.49613904083496]
[13.482330902558596, 20.98223782280425, 17.398455513529996, 16.4278696
2022209, 14.554667068104749]
[16.737263006852096, 15.16842421930363, 15.735525955404666, 13.4501339
64723387, 20.015909770189833]
[13.07667032890979, 13.621739999613657, 13.511207044159454, 14.8980787
32985748, 16.0459808750653]
[15.369370389987422, 13.121097818908407, 24.718052924057567, 19.446768
325218056, 18.165346685441186]
[17.32389263569356, 19.472465032059585, 19.575050528545635, 18.8674344
56684546, 17.515829750776074]
```

In [41]:

```python
# quantize H1 to integer h(q) , hash h = [h(q)] (1<=q<=80)
def quantize(H1):
    '''
    :param H1: [T D]

    :return h: 量化后的H1
    :rtype list
    '''
    H1 = np.array(H1)
    h = np.around(H1*10 + 0.5)
    return list(h)

h = quantize(H1)
print('len(h):',len(h))
for i in range(0,len(h),4):
    print(h[i:i+4])
```

executed in 6ms, finished 15:51:25 2019-12-15

```
len(h): 80
[1473.0, 10.0, 1.0, 4.0]
[2874.0, 10.0, 1.0, 3.0]
[1595.0, 10.0, 1.0, 4.0]
[2721.0, 10.0, 1.0, 3.0]
[190.0, 131.0, 127.0, 129.0]
[128.0, 129.0, 130.0, 134.0]
[196.0, 129.0, 128.0, 127.0]
[132.0, 129.0, 129.0, 132.0]
[217.0, 190.0, 128.0, 127.0]
[129.0, 132.0, 130.0, 133.0]
[181.0, 207.0, 177.0, 205.0]
[185.0, 184.0, 170.0, 133.0]
[198.0, 161.0, 180.0, 169.0]
[161.0, 153.0, 135.0, 135.0]
[210.0, 174.0, 165.0, 146.0]
[168.0, 152.0, 158.0, 135.0]
[201.0, 131.0, 137.0, 136.0]
[149.0, 161.0, 154.0, 132.0]
[248.0, 195.0, 182.0, 174.0]
[195.0, 196.0, 189.0, 176.0]
```

In [ ]:

```python
1
```