JPEG 图像算法

1. JPEG 简介和相关工作

JPEG 是由联合图像专家组（Joint Photographic Experts Group）开发的一种图像压缩方法，文件扩展名为.jpg 或.jpeg。1992 年它被正式采纳为国际标准，到目前为止是使用最广泛的图像压缩方法。JPEG 是一种基于离散余弦变换的有损压缩格式，能够将图像压缩在很小的储存空间，图像中重复或者不重要的资料会被丢失。JPEG 支持多种压缩级别，压缩比率通常在 10：1 到 40：1 之间，压缩比越大，品质就越低；相反地，压缩比越小，品质就越好，我们完全可以在图像质量和文件尺寸之间找到平衡点。JPEG 再后来升级成了 JPEG2000，这是一种基于小波变换的图像压缩标准，文件扩展名通常为.jp2，同时支持有损和无损压缩，压缩率比 JPEG 高约 30%，并且支持图像渐进传输，让图像由朦胧到清晰显示。此外，JPEG2000 还有“感兴趣区域”特性，可以任意指定图像上某个区域的压缩质量，也可以选择制定的部分先解压缩。JPEG2000 可以向下兼容 JPEG，同时优势明显，被认为是未来取代 JPEG 的下一代图像压缩标准。

2. JPEG 算法描述以及代码

本次实验实现传统的 JPEG 算法，压缩流程包含了以下几个步骤：颜色模式、填充、二次采样、分块、离散余弦变换（DCT）、量化、Zigzag 扫描排序、DC 和 AC 编码、熵编码。下面我们逐个步骤说明。

2.1 颜色模式转换

JPEG 使用的不是 RGB 颜色空间，而是 YIQ 或者 YUV 的颜色空间。本次实验采用的是 YUV 颜色空间，RGB 和 YUV 的转换关系如下：

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & 4187 & 0.0813 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.772 & 0 \end{bmatrix} \left(\begin{bmatrix} Y \\ U \\ V \end{bmatrix} - \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \right) \tag{2}$$

2.2 填充

由于 JPEG 算法后续是对 8*8 的子块进行 DCT 运算，而在分块之前还要对 U、V 分量进行一次采样，采样是在每个 2*2 的单元中进行的，因此我们需要将原图的图像的长宽用 0 填充到 16 的倍数，填充的方法很简单，向右和向下填充即可，如下图。

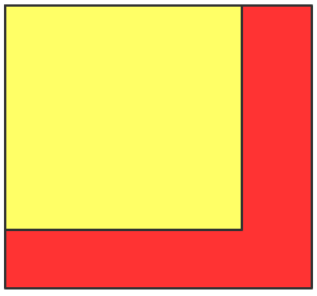


图 1. 填充方法示例。黄色区域是原图，
红色区域是用 0 填充出来的部分。

2.3 二次采样

经过研究发现，人眼对亮度的敏感度比色彩变换的敏感度高，所以可以认为 Y 分量比 U 和 V 更重要，因此对 U 和 V 分量进行采样，作用是在人眼不可察觉的前提下通过损失一定精度来降低数据的存储量。本次实验采用 4：1：1 的采样方式，具体来说，在每个 2*2 的单元中，Y 分量采样 4 次（相当于保留了全部数据），U 和 V 采样一次（取 2*2 区域的左上角那个单元）。

2.4 分块

DCT 变换是对 8*8 的子块进行处理，所以我们将上述步骤的图片按从左到右，从上到下的顺序分成若干 8*8 的子块存储下来即可。

2.5 离散余弦变换（DCT）

这一步就是对前面分好的每一个 8*8 子块，各自单独按公式计算就可以了。应用 DCT 变换的目的是减少图像中的高频分量，即图像中的一些细节信息，这是因为人眼对细节信息不敏感，这样做节省存储空间，但是图像信息量损失很少。离散余弦变换（DCT）公式如下：

$$F(u,v) = C(u,N)C(v,N)\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} f(i,j)cos\frac{\pi u(2i+1)}{2N}cos\frac{\pi v(2j+1)}{2N} \tag{3}$$

其中：

$$C(x,N) = \begin{Bmatrix} \sqrt{\frac{1}{N}} & x=0 \\ \sqrt{\frac{2}{N}} & x=0 \end{Bmatrix} \tag{4}$$

解压时的逆变换公式为：

$$f(i,j) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u,N)C(v,N)F(u,v)cos\frac{\pi u(2i+1)}{2N}cos\frac{\pi v(2j+1)}{2N} \tag{5}$$

2.6 量化

量化是 JPEG 算法中图像精度损失的根源，也是产生压缩效果的根源。在经过 DCT 变换后，由于每个 8*8 块中位于(0,0)的是直流分量，因此，在每一个块中，左上角是图像的低频成分，右下角是高频成分。由于人眼对高频率分量不敏感，我们对图像中每一个成分除以一个常量（即量化），使得高频成分接近为 0，低频成分变成很小的正数或者负数，这样做就能节省存储数字所需位数，达到减小存储空间的目的。量化算法如下式所示：

$$\hat{F}(u,v) = round(\frac{F(u,v)}{Q(u,v)}) \tag{6}$$

其中 F(u,v)是 DCT 变换得到的结果，Q(u,v)是量化矩阵，$\hat{F}(u,v)$是量化后的结果。Q 分为亮度和色度两种矩阵。对于 Y 分量，采用亮度量化矩阵（如表 1），对 U 和 V 采用色度量化矩阵（如表 2）

表 1.亮度量化表

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

表 2.色度量化表

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

2.7 Zigzag 扫描排序

后续的 AC 编码需要先进行 Zigzag 扫描排序，目的是为了保证低频分量先出现，高频分量后出现，以增加连续的“0”的个数。简单点说就是将 8*8 的矩阵中的数据换一个顺序得到一个 64 位向量，扫描方法如下图所示。

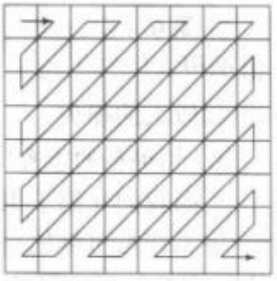


图 2. JPEG 中的 Z 字型扫描

2.8 DC 系数的 DPCM 编码

DC 系数即经过 DCT 变换后的 8*8 块左上角的那一个数字，它有两个特点：系数的数值较大、相邻两个块的 DC 系数值变化不大。因此使用 DPCM（差分脉冲调制编码）对 DC 系数进行编码，方法是除却第一个 DC 系数的编码是其本身外，后续的 DC 系数采用它与前一个块的 DC 系数的差值来进行编码，由于差值很小，因此编码所需位数也会大大减少。

2.9 AC 系数的 RLE 编码

AC 系数在 Zigzag 扫描后进行游长(RLE)编码。在 RLE 步骤中，每个串采用{RUNLENGTH，VALUE}的数对方式表示，其中 RUNLENGTH 表示串里 0 的数目，VALUE 表示下一个非 0 系数。为了节省位数，一个特殊的数字对(0,0)紧跟着最后一个非 0 的 AC 系数，表示达到了“块”的结尾。假设 Zigzag 扫描后得到的向量为：

(32,6,-1,-1,0,-1,0,0,0,-1,0,0,1,0,0,…,0)

那么 RLE 压缩的结果就是：

(0,6)(0,-1)(0,-1)(1,-1)(3,-1)(2,1)(0,0)

2.10 熵编码

为了更加节省空间，对于 DC 和 AC 系数，我们先求出它们的 VLI（变长整数）形式。之后再通过查找 Huffman 编码表进行熵编码。

VLL 形式其实就是（VLI_len，VLI_code）这样的形式，其中 VLI_code 为系数的二进制形式，VLI_len 为该二进制数的位数，负数的二进制数用其正数的反码表示。

Hiffman 编码表分为 DC 和 AC 两种，每种里面又分为亮度表和色度表，DC 的两种表如表 3 和表 4 所示。

表 3. DC 系数亮度编码表

键值(bitcount)	编码值（二进制数）
0	000
1	010
2	011
3	100
4	101
5	110
6	1110
7	11110
8	111110
9	1111110
10	11111110

11	111111110

表 4. DC 系数色度编码表

键值(bitcount)	编码值（二进制数）
0	00
1	01
2	10
3	110
4	1110
5	11110
6	111110
7	1111110
8	11111110
9	111111110
10	1111111110
11	11111111110

AC 的亮度和色度 Huffman 编码表由于过长，不在本报告中展示，具体可见代码中 tools 文件夹下的 Table.py 里的 AC_Luminance_Huffman(AC 亮度 Huffman 编码表)和 AC_Chroma_Huffman(AC 色度 Huffman 编码表)。与 DC 编码表不同之处在于 AC 的表的键值变为(RUNLENGTH, VLI_len)。

设某个系数的 Huffman 编码为 Huffman_code，那么其熵编码就是：

$$Entropy\_code = Huffman\_code \,||\, VLI\_code \tag{7}$$

其中||是拼接的意思。最后将一个子块的 DC 熵编码和所有 AC 熵编码按顺序拼接在一起就得到该子块的熵编码。

3. 关键代码

本次实验的代码采用 python3.7 实现。

3.1 颜色模式转换

按照 2.1 中的公式计算，即可，这里同时将原本图片的二维存储按照从左到右，从上到下的顺序改成了一维存储。

```python
def RGB2YUV(img):
    """
    RBG2YUV: RGB -> YUV
    根据公式换算，一般来说U,V是有符号的数字，
    但这里通过加上128，使其变为无符号数，
    方便存储和计算
    另外这一步将存储格式从二维转成了一维
    """
    size = img.size
    Y, U, V = [], [], []
    for i in range(size[0]):
        for j in range(size[1]):
            pixel = img.getpixel((i, j))
            Y.append(0.299*pixel[0] + 0.587*pixel[1] + 0.114*pixel[2])
            U.append(-0.1687*pixel[0] - 0.3313*pixel[1] + 0.5*pixel[2] + 128)
            V.append(0.5*pixel[0] - 0.4187*pixel[1] - 0.0813*pixel[2] + 128)
    return Y, U, V
```

```python
def YUV2RGB(Y, U, V):
    """
    YUV2RGB: YUV -> RGB
    """
    R, G, B = [], [], []
    for i in range(0, len(Y)):
        R.append(Y[i] + 1.402*(V[i] - 128))
        G.append(Y[i] - 0.34414*(U[i] - 128) - 0.71414*(V[i] - 128))
        B.append(Y[i] + 1.772*(U[i] - 128))
    return R, G, B
```

3.2 压缩时的填充与解码时的解除填充

```python
def fill(img, size):
    """
    fill: 将原图像长宽用0补齐到16的倍数
    """
    newsize = [0, 0]
    new_img = [[], [], []]
    # 先计算填充后的长和宽分别是多少
    newsize[0] = size[0] + 16 - size[0]%16 if size[0] % 16 != 0 else size[0]
    newsize[1] = size[1] + 16 - size[1]%16 if size[1] % 16 != 0 else size[1]

    for k in range(3): # 分别枚举Y、U、V
        for i in range(newsize[0]):  # 枚举填充后的图像的每个像素点(i,j)
            for j in range(newsize[1]):
                if i < size[0] and j < size[1]: #(i,j)处于原图范围内，就用原图的数值
                    new_img[k].append(img[k][i * size[1] + j])
                else: # (i,j)在原图范围外就补0
                    new_img[k].append(0)

    return new_img[0], new_img[1], new_img[2], newsize
```

```python
def re_fill(matrix, size, newsize):
    """
    解除填充
    保留填充后的图像中原图的部分，去掉补0的部分
    """
    ori_img = [[], [], []]
    print(size, newsize, len(matrix))
    for k in range(3):
        for i in range(0, newsize[0]):
            for j in range(0, newsize[1]):
                if i < size[0] and j < size[1]: # (i,j)是原图范围内的才保留
                    ori_img[k].append(matrix[k][i * newsize[1] + j])
    return ori_img
```

3.3 压缩时的二次采样与解码时的解除采样

注意，采样只针对 U 和 V 两个分量，按照 2.3 所说的方式枚举每个 2*2 的小方形，只保留左上角。在解除采样的时候，就将那个左上角的值填入 2*2 的小方形的 4 个单元格中。这一步会造成一些信息丢失，但是由于人眼对 U 和 V 不敏感，所以看不太出来。

```python
def sample(matrix, size):
    """
    二次采样，只针对U和V
    """
    temp = []
    for X in range(0, size[0] // 2): # 枚举每个2*2小矩形
        for Y in range(0, size[1] // 2):
            idx = X * 2 * size[1] + Y * 2 # 算出该小矩形左上角的索引
            temp.append(matrix[idx])  # 保留该值
    return temp
```

```python
def Inverse_sample(matrix, size):
    """
    解除采样
    将原先每个2*2的小矩形的4个单元都填入原左上角的那个值
    """
    temp = []
    for X in range(0, size[0]):
        for i in range(0, 2): # X为采样后的矩形的行数，对于一个值要连续填充两行，用i来枚举
            for Y in range(0, size[1]): # 枚举一行中的每一列
                idx = X * size[1] + Y  # 原左上角的值在采样后的矩阵里的索引
                temp.append(matrix[idx])  #填充两次达到填充了相邻两列的目的
                temp.append(matrix[idx])
    return temp
```

3.4 压缩时的分块与解码时的合并分块

```python
def partition(matrix, size):
    """
    分块操作
    """
    temp = []
    temp_size = 8 # 分块后的每一块的长和宽均为8
    for i in range(0, size[0] // temp_size):
        for j in range(0, size[1] // temp_size):
            start_idx = i * size[1] * 8 + j * 8 # 枚举每一个8*8块的左上角索引----起点索引
            block = []
            for k in range(0, temp_size):   # 将该块的64个值拿出来存入一个block
                for L in range(0, temp_size):
                    block.append(matrix[start_idx + k * size[1] + L])
            temp.append(block)
    return temp
```

合并分块的时候需要理清思路。假设某个分量重一共有 4 个分块，2*2 的排列着，我们从左到右从上到下编号为 1、2、3、4。1 和 2 在上，3 和 4 在下。那么先枚举 1 和 2 的第一行，把数字存下来，再枚举 1 和 2 的第二行…以此类推，枚举完 1 和 2 再枚举 3 和 4，方法一样。这样才能保证合并后的数值顺序是正确的。

```python
def merge(matrix, size):
    """
    合并分块
    """
    temp = []
    M = 8
    for i in range(size[0] // M):  # (i,j)为某一个块的索引
        for k in range(0, M):      # (k, L)为 (i,j)这个块中每一个元素的索引
            for j in range(0, size[1] // M):
                for L in range(0, M):   # 按照分块前每一行每一列的顺序枚举元素
                    if i * (size[1] // M) + j >= len(matrix):
                        continue
                    elif k * M + L >= len(matrix[i * (size[1] // M) + j]):
                        continue
                    temp.append(matrix[i * (size[1] // M) + j][k * M + L])
    return temp
```

3.5 DCT 和 IDCT

按照 2.5 的公式计算即可，没有难度。

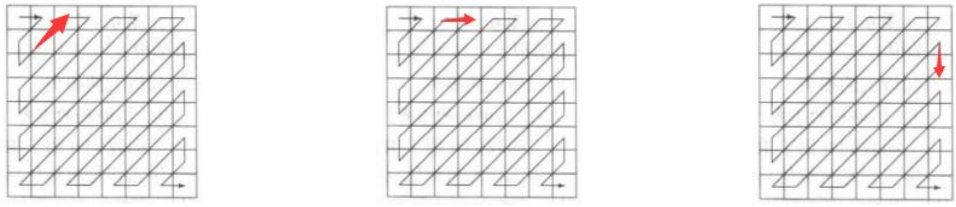
```python
def C(u, N):
    return math.sqrt(1.0 / N) if u == 0 else math.sqrt(2.0 / N)

def DCT(matrix):
    DCT_matrix = []
    for u in range(0, M):
        for v in range(0, M):
            tmp = 0
            for i in range(0, M):
                for j in range(0, M):
                    tmp += math.cos((2*i+1)*u*math.pi/(2*M)) * math.cos((2*j+1)*v*math.pi/(2*M)) * matrix[i*M+j]
            DCT_matrix.append(int(round(C(u,M)*C(v,M)*tmp)))
    return DCT_matrix
```

```python
def IDCT(matrix):
    IDCT_matrix = []
    for i in range(0, M):
        for j in range(0, M):
            tmp = 0
            for u in range(0, M):
                for v in range(0, M):
                    tmp += C(u, M)*C(v, M)*math.cos((2*i+1)*u*math.pi/(2*M)) * math.cos((2*j+1)*v*math.pi/(2*M)) * matrix[u*M+v]
            IDCT_matrix.append(tmp)
    return IDCT_matrix
```

DCT 后是量化，Y 进行亮度量化，U 和 V 是色度量化，由于量化操作就是查表之后除以一个常量，比较简单，代码不在这里展示。解码就是把除了的常量乘回来。

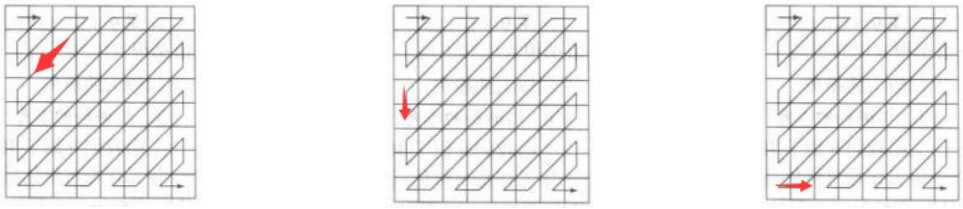
3.6 Zigzag 扫描排序

按照先往右上再往左下的顺序扫描储存。对于两种情况由各自分为 3 种情况。图 3 说明了右上的三种情况，图 4 说明了左下的三种情况。



(a) (b) (c)

图 3. 右上扫描的 3 种情况。(a) (i, j)->(i-1, j+1) (b) (i, j)->(i, j+1) (a) (i, j)->(i+1, j)



(a) (b) (c)

图 4. 左下扫描的 3 种情况。(a) (i, j)->(i+1, j-1) (b) (i, j)->(i+1, j) (a) (i, j)->(i, j+1)

在实现的时候使用 Zigzag_order 存储真实索引。Zigzag_order[i]表示扫描后的第 i 个数在扫描前的索引。后续恢复的时候就使用 Zigzag_order 来帮助找到真实位置。

```
def zigzag(matrix, Zigzag_order):
    """
    zigzag扫描
    """
    temp = []
    i, j =0, 0
    direct = 1 # 0 for 左下， 1 for 右上
    temp.append(matrix[0])
    for k in range(0, M * M - 1):
        if direct == 0:
            if check(i+1, j-1): # check是检查是否超出边界
                i, j = i+1, j-1
            elif check(i+1, j):
                i, direct = i+1, 1
            elif check(i, j+1):
                j, direct = j+1, 1
        else:
            if check(i-1, j+1):
                i, j = i-1, j+1
            elif check(i, j+1):
                j, direct = j+1, 0
            elif check(i+1, j):
                i, direct = i+1, 0
        if len(Zigzag_order) != M*M: # Zigzag_order[i]表示扫描后的第i个在扫描前的索引。
            Zigzag_order.append(i * M + j)
        temp.append(matrix[i * M + j])
    return temp
```

```
def re_zigzag(matrix, Zigzag_order):
    """
    使用Zigzag_order恢复扫描前的顺序
    """temp: list
    temp = list(range(64))
    for i in range(64):
        temp[Zigzag_order[i]] = matrix[i]
    return temp
```

3.7 DC 和 AC 编解码

这里为了方便编程，将 DC 的 DPCM 编码也写成（RUNLENGTH， VALUE）的数对形式，只不过这里 RUNLENGTH = 0。在 AC 的游长编码中，由于 JPEG 使用一个字节的高 4 位表示连续的 0 的个数，所以我们一旦遇到连续 15 个 0，加入第 16 个还是 0，这里就要增加一个(15,0)的数对，然后将计数器清 0。解码时，对于每个数对，先加入 RUNLENGTH 个 0，再加入恢复后的 DC 分量或者 AC 分量。

```python
def RLE(temp, src): # AC游长编码
    zero_cnt = 0
    for i in range(1, M*M):
        if src[i] != 0:
            temp.append((zero_cnt, src[i]))
            zero_cnt = 0
        elif i == M*M-1:
            temp.append((zero_cnt, src[i]))
        else:
            zero_cnt += 1
        if zero_cnt > 15: #jpeg使用一个字节的高4位表示连续0的个数，即最多15个
            temp.append((15, 0))
            zero_cnt = 0

def DC_and_AC_encode(matrix, length):
    result = []
    for i in range(length):
        temp = []
        #DC-DPCM encode
        if i == 0:
            temp.append((0, matrix[i][0]))
        else:
            temp.append((0, matrix[i][0] - matrix[i - 1][0]))
        #AC-RLE encode
        RLE(temp, matrix[i])
        result.append(temp)
    return result
```

```python
def DC_and_AC_decode(matrix, length):
    result = []
    for i in range(length):
        temp = []
        for j in range(len(matrix[i])):# 枚举每一个数对
            for k in range(matrix[i][j][0]):  #先加入RUNLENGTH个0
                temp.append(0)
            if i > 0 and j == 0: # 每个块的第一个数对是DC分量，恢复DC分量
                temp.append(matrix[i][j][1] + result[i-1][0])
            else:# 恢复AC分量
                temp.append(matrix[i][j][1])
        result.append(temp)
    return result
```

3.8 熵编码和熵解码

编码部分比较简单，按照 2.10 中所述先求出 VLI 形式的数对(VLI_len，VLI_code)，再查表得 Huffman 编码，将 Huffman 编码和 VLI_code 拼接得到熵编码。

解码的时候也需要通过查表。比如 Y 分量某一个块最后编码是一个很长的二进制数，假设编码是 01010111，开头是 DC 分量，那么我们在 DC 的亮度 Huffman 表中查找有无值为 0 的键值。没有的话就查找值为 01 的键值，也没有就查找值为 010 的键值，这时发现键值为 1 的时候，值为 010。这说明该 DC 分量的 VLI_len 是 1。然后紧接着 VLI_len 个数就是它的 VLI_code，在这里是 1。所以解码后的 DC 的 VLI 形式就是(1,1)，非 VLI 形式就是 1。接着是 AC 分量的解码。用相同的方法在 AC 的亮度 Huffman 表中查表，这里我们找到值为 01 的时候，存在键值(0,2)，说明该 AC 分量的 RUNLENGTH=0, VLI_len=2,紧接着 VLI_len 个数是它的 VLI_code，在这里是 11，恢复成十进制就是 3，所以该 AC 分量的游长编码数对为(0,3)

值得注意的是，由于我没有限制 VLI_code 用几位二进制数表示，这里就采用了它们的实际位数来表示，那么就会出现 0 和-1 的 VLI 形式都是(1,0)，为了区分两者，我将 0 的 VLI 形式改成了（0，0)。另外还有一点，假如某一个块的最后一串数字是 0，但是没有达到连续 16 个 0 所以不能表示为(15,0)，我们假设它是(13，0)，那么 VLI_len=0, 但是 AC 的 Huffman 表中没有键值为(13,0)的，只有(0,0)和(15,0)，所以我们统一将末尾是一串 0 的用（15，0）表示，但是解码的时候需要判断最后这个数对是否是真实的(15,0),方法也很简单，用一个 tot_len 记录已编码（解码）的数字(一个块中一共有 64 个数字)，每解码一个数对，比如 AC 的(13,2)就是解码了 13+1=14 个数字，这样在最后遇到（15，0）时判

断一下 tot_len+16 是否大于 64，若大于，就不是真实的(15,0)，减掉多出来的差值即可。

```python
def Entropy_encode(matrix, length, DC_AC):
    result = []
    for k in range(length):
        tot_len = 0  # 用来统计已经编码掉的数，总共64个数，比如AC的游长编码得到一个数对(3,2)这里是3+1=4个数。
        (DC_VLI_len, DC_VLI_code) = VLI(matrix[k][0][1]) # 求出DC的VLI数对
        DC_Huffman_code = ''
        if DC_AC == 0: # 查表得DC 的Huffman编码
            DC_Huffman_code = DC_Luminance_Huffman[DC_VLI_len]
        else:
            DC_Huffman_code = DC_Chroma_Huffman[DC_VLI_len]
        entropy = DC_Huffman_code + DC_VLI_code  # 拼接得到熵编码
        tot_len += 1
        for i in range(1, len(matrix[k])):
            (AC_VLI_len, AC_VLI_code) = VLI(matrix[k][i][1]) # 求出AC的VLI数对
            AC_Huffman_code = ''
            if tot_len + matrix[k][i][0] + 1 == 64 and matrix[k][i][1] == 0:  # 判断是否是最后一个数对
                if DC_AC == 0:
                    AC_Huffman_code = AC_Luminance_Huffman[(15, 0)]
                else:
                    AC_Huffman_code = AC_Chroma_Huffman[(15, 0)]
                tot_len += 16
            else: #不是最后一个数对
                if DC_AC == 0:
                    AC_Huffman_code = AC_Luminance_Huffman[(matrix[k][i][0], AC_VLI_len)]
                else:
                    AC_Huffman_code = AC_Chroma_Huffman[(matrix[k][i][0], AC_VLI_len)]
                tot_len += matrix[k][i][0] + 1
            entropy += AC_Huffman_code + AC_VLI_code
        result.append(entropy)
    return result
```

```python
def Entropy_decode(matrix, length, DC_AC):
    result = []
    for i in range(length):
        j, tot_len, isFirst, str_temp, temp = 0, 0, 0, '', []
        while j < len(matrix[i]):
            str_temp += matrix[i][j]
            j += 1
            if DC_AC == 0: # 0 为亮度 1 为色度
                if isFirst == 0: # 第一个用DC Huffman表
                    items = DC_Luminance_Huffman.items()
                elif isFirst == 1: # 否则用AC Huffman表
                    items = AC_Luminance_Huffman.items()
            elif DC_AC == 1:
                if isFirst == 0: # 第一个用DC Huffman表
                    items = DC_Chroma_Huffman.items()
                elif isFirst == 1: # 否则用AC Huffman表
                    items = AC_Chroma_Huffman.items()
            for (k, v) in items:
                if v == str_temp:
                    str_temp = ''
                    if isFirst == 0: # DC的VLI变回整数
                        VLI_str = ''
                        for z in range(k):
                            VLI_str += matrix[i][j]
                            j += 1
                        if VLI_str == '':
                            temp.append((0, 0))
                        else:
                            temp.append((0, De_VLI(VLI_str)))
                        tot_len += 1
```

```python
                    elif isFirst == 1: # AC的VLI变回整数,此时k为(RUNLENGTH,AC_VLI_len)
                        VLI_str = ''
                        for z in range(k[1]):
                            VLI_str += matrix[i][j]
                            j += 1
                        if VLI_str == '':
                            if tot_len + k[0] + 1 > 64:
                                temp.append((64 - tot_len - 1, 0))
                            else:
                                temp.append((k[0], 0))
                        else:
                            temp.append((k[0], De_VLI(VLI_str)))
                        tot_len += k[0] + 1
                    isFirst = 1
                    break
        result.append(temp)
    return result
```

4. 实验结果

我们以 Y 分量为例来观察压缩 lena.jpg 的每一步的结果，直接从经过填充后的 Y 分量开始，由于数字太多，我们只放出前 20 个数字。

```
[163.254, 163.254, 163.254, 163.02599999999998, 161.738, 161.09699999999998, 159.09699999999998, 157.
88, 156.35299999999998, 156.12499999999997, 155.837, 156.19600000000003, 156.19600000000003, 155.9680
0000000002, 155.96800000000002, 155.97899999999998, 157.19599999999997, 156.19600000000003, 155.196,
 154.196]
```

经过分块后，Y 的第一个块：

```
The first block of Y:
163.25  163.25  163.25  163.03  161.74  161.1  159.1  157.88
161.25  162.25  162.25  162.03  160.74  159.1  157.87  156.88
160.25  160.25  161.03  160.03  159.74  158.1  155.87  154.88
159.25  159.03  159.03  159.03  158.1  157.1  154.87  154.35
159.03  159.03  159.03  159.03  158.1  157.1  154.87  154.35
160.03  160.03  161.03  159.8  160.1  157.87  155.87  155.35
161.03  162.03  161.8  161.8  161.1  158.87  157.75  157.35
162.8  162.8  162.8  162.8  161.87  160.87  158.75  158.12
```

经过 DCT 后 Y 的第一个块：

```
The first block of Y after DCT:
1276  14  -7  0  0  -1  0  0
0  1  0  0  0  0  0  0
12  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  1  1  0  0  0  -1
0  0  0  0  0  0  0  0
0  0  0  1  0  0  1  1
0  0  0  0  0  0  0  0
```

经过量化后 Y 的第一个块，可以看到右下方的高频分量已经全部变为 0，而有上方的低频分量的绝对值也变小了。

```
The first block of Y after quantization:
80  1  -1  0  0  0  0  0
0  0  0  0  0  0  0  0
1  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  0  0  0
0  0  0  0  0  0  0  0
```

Y 的第一个块经过 Zigzag 扫描后得到的 64 位向量：

```
The first block of Y after Zigzag:
[80, 1, 0, 1, 0, -1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Y 的第一个块经过 DC 和 AC 编码后得到的数对们：

```
The first block of Y after DC and AC encode:
[(0, 80), (0, 1), (1, 1), (1, -1), (15, 0), (15, 0), (15, 0), (9, 0)]
```

Y 的第一个块经过熵编码后的结果：

```
The first block of Y after entropy encoding:
1111010100000011100111000111111110011111111110011111111110011111111001
```

压缩结果，其中原图是 408*408 的 24 位图，所以原图总位数为 408*408*24=3995136，这里我们可以看到压缩后位数减少了很多：

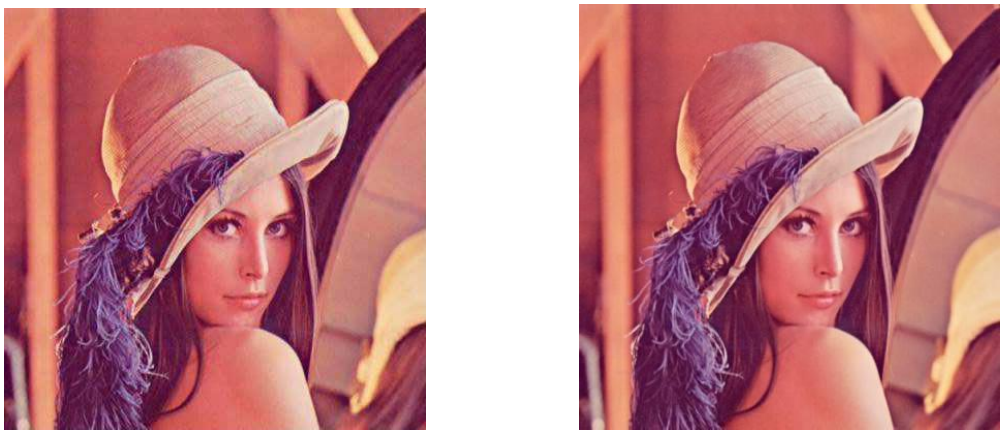
```
原图总位数：3995136
压缩后总位数：293051
压缩比：13.632903487788814
```

解压缩至解除填充之前，Y 的前 20 个数：

```
[162.56046126975616, 163.22681901667755, 164.04331088149868, 164.29905054114317, 163.5403272481151, 161.88264974593417, 159.99316111
00243, 158.74610169512027, 153.94221769576615, 153.94221769576615, 153.94221769576615, 153.94221769576615, 153.94221769576615, 153.9
4221769576615, 153.94221769576615, 153.94221769576615, 154.09282021268208, 154.3831710466734, 154.91966943221777, 155.620638353486]
```

我们可以看到跟压缩前相比，这 20 个数是有一定误差的，这是经过量化后造成了精度丢失的结果。

最后我们对比解压缩得到的图（左图，lena. jpeg）与原来的图片（右图,lena. jpg），可以看到解压缩得到的图片有一点点失真但是不太明显，效果还是不错的。



（a）　　(b)

图 5. Jpeg 解压缩图与原图对比。(a)lena. jpeg　　（b）lena. jpg