**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：

(1)

(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。

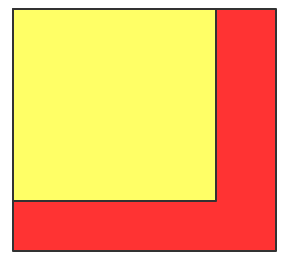


图1.填充方法示例。黄色区域是原图，

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：

(3)

其中：

(4)

解压时的逆变换公式为：

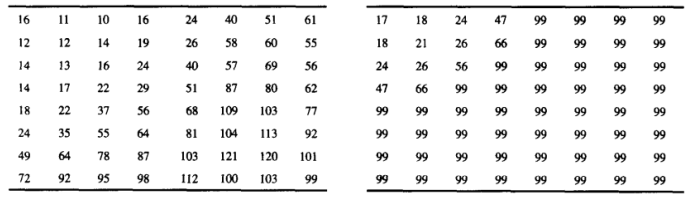
(5)

**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： (6)

其中F(u,v)是DCT变换得到的结果，Q(u,v)是量化矩阵，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 表2.色度量化表



**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。

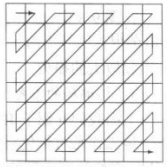


图2.JPEG中的Z字型扫描

**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：

(7)

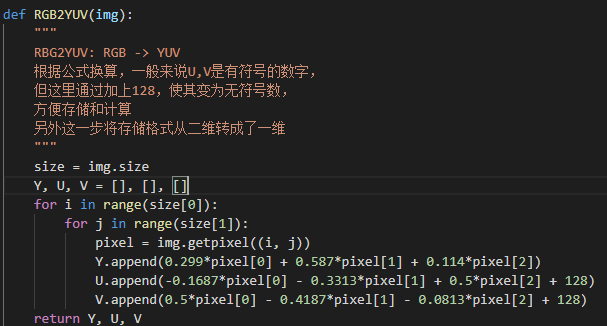
其中是拼接的意思。最后将一个子块的DC 熵编码和所有AC熵编码按顺序拼接在一起就得到该子块的熵编码。

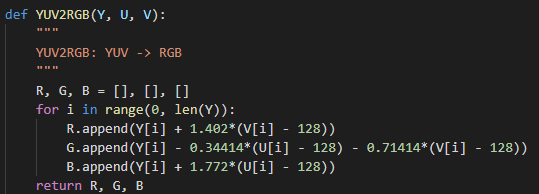
**3.关键代码**

本次实验的代码采用python3.7实现。

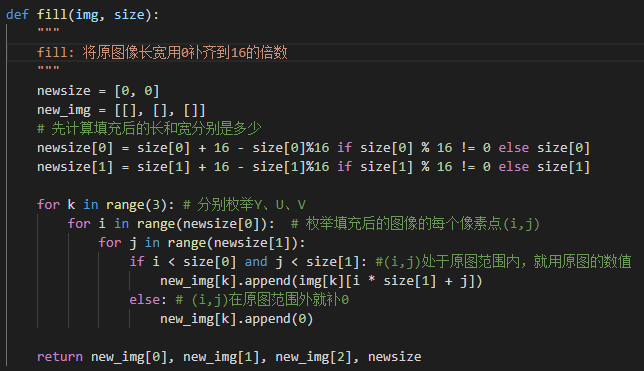
**3.1 颜色模式转换**

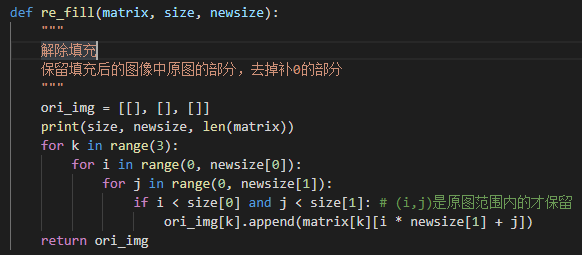
按照2.1中的公式计算，即可，这里同时将原本图片的二维存储按照从左到右，从上到下的顺序改成了一维存储。





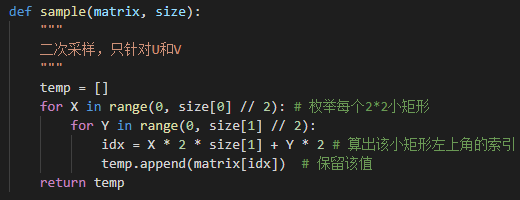
**3.2 压缩时的填充与解码时的解除填充**

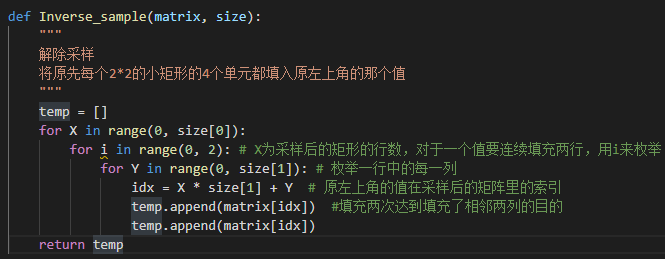




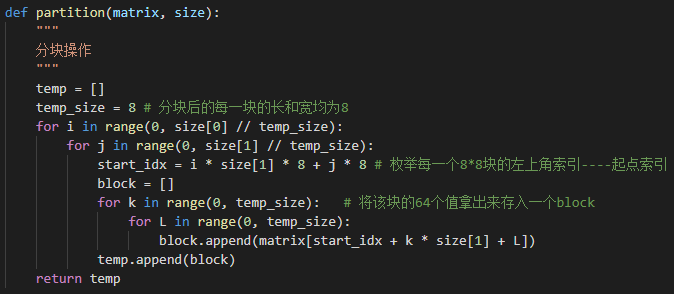
**3.3 压缩时的二次采样与解码时的解除采样**

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。

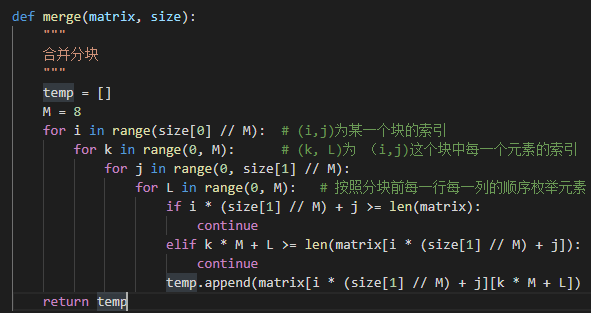




**3.4 压缩时的分块与解码时的合并分块**

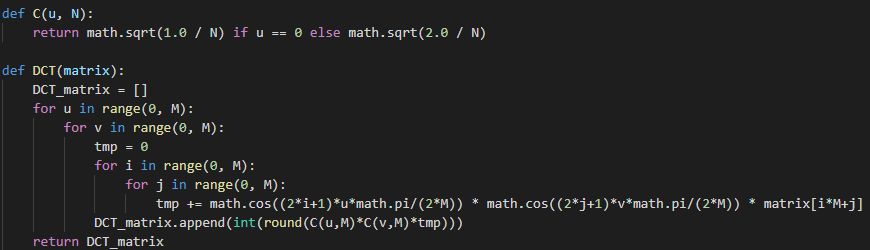


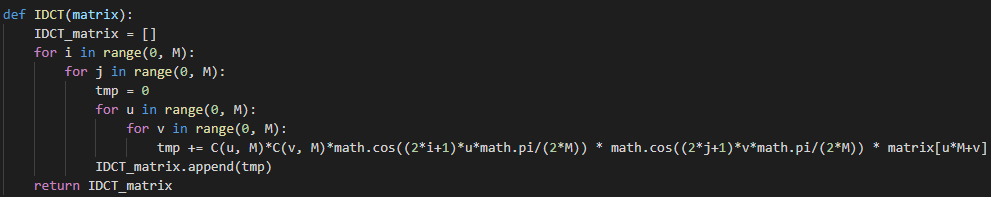
合并分块的时候需要理清思路。假设某个分量重一共有4个分块，2\*2的排列着，我们从左到右从上到下编号为1、2、3、4。1和2在上，3和4在下。那么先枚举1和2的第一行，把数字存下来，再枚举1和2的第二行…以此类推，枚举完1和2再枚举3和4，方法一样。这样才能保证合并后的数值顺序是正确的。



**3.5 DCT和IDCT**

按照2.5的公式计算即可，没有难度。

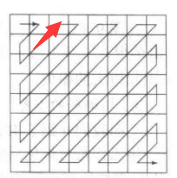
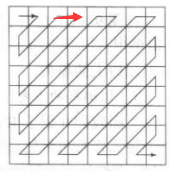
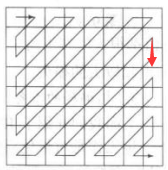




DCT后是量化，Y进行亮度量化，U和V是色度量化，由于量化操作就是查表之后除以一个常量，比较简单，代码不在这里展示。解码就是把除了的常量乘回来。

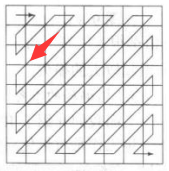
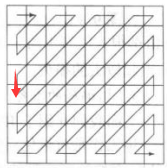
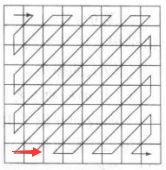
**3.6 Zigzag扫描排序**

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)

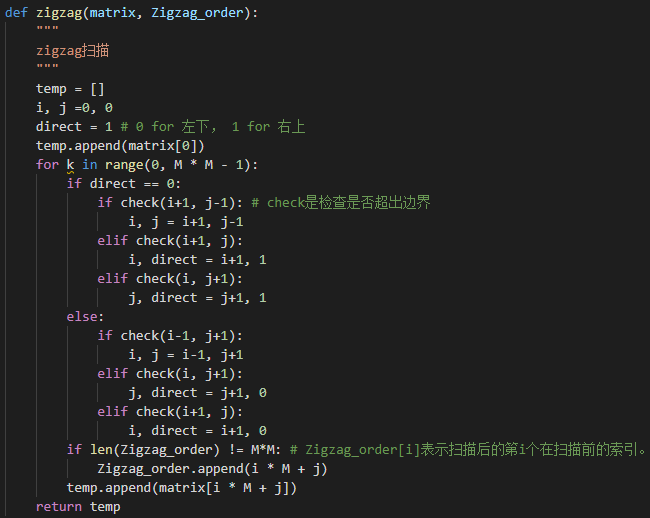
图3. 右上扫描的3种情况。(a) (i,j)->(i-1,j+1) (b) (i,j)->(i,j+1) (a) (i,j)->(i+1,j)

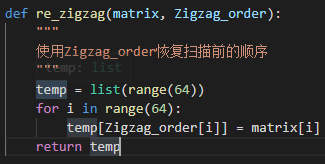
  

(a) (b) (c)

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)

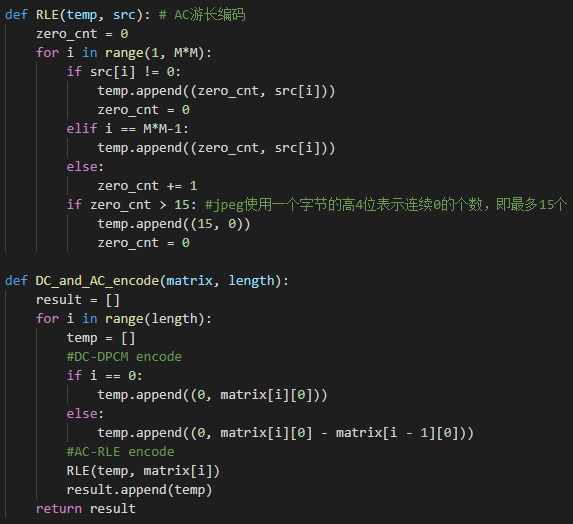
在实现的时候使用Zigzag\_order存储真实索引。Zigzag\_order[i]表示扫描后的第i个数在扫描前的索引。后续恢复的时候就使用Zigzag\_order来帮助找到真实位置。

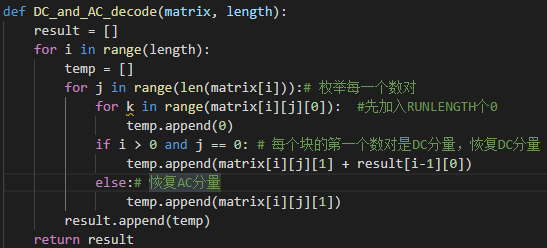




**3.7 DC和AC编解码**

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。



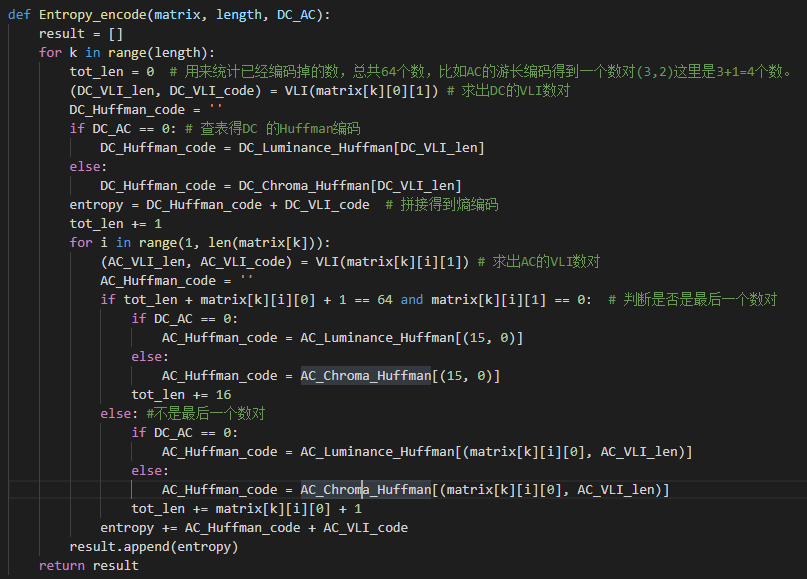


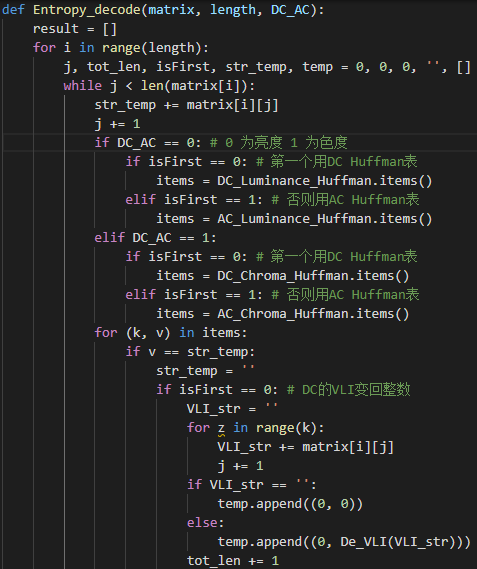
**3.8 熵编码和熵解码**

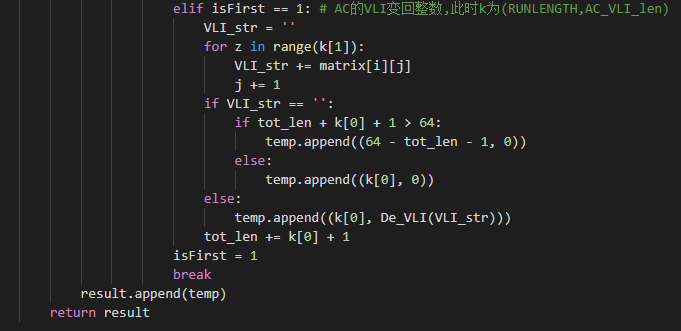
编码部分比较简单，按照2.10中所述先求出VLI形式的数对(VLI\_len，VLI\_code)，再查表得Huffman编码，将Huffman编码和VLI\_code拼接得到熵编码。

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。

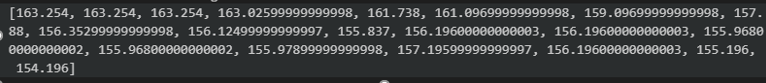




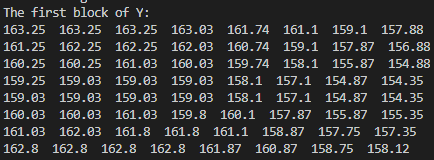


**4.实验结果**

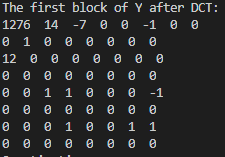
我们以Y分量为例来观察压缩lena.jpg的每一步的结果，直接从经过填充后的Y分量开始，由于数字太多，我们只放出前20个数字。



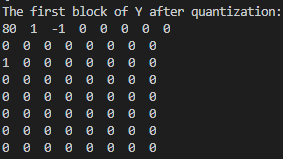
经过分块后，Y的第一个块：



经过DCT后Y的第一个块：



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。



Y的第一个块经过Zigzag扫描后得到的64位向量：



Y的第一个块经过DC和AC编码后得到的数对们：



Y的第一个块经过熵编码后的结果：



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：



解压缩至解除填充之前，Y的前20个数：



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