

# JPEG: A Communication System | JACS

王一恒 15307130257 *Penicillin*

Correspondence: 15307130257@fudan.edu.cn

## SUMMARY

在本项目中，我们依据 JPEG 静态图像压缩标准对原始图像进行压缩；以 BSC 信道噪声为模型进行信道模拟，采取 FEC 的汉明码、ARQ 的循环冗余校验码、FEC 的卷积码等方法进行信道编解码；在信宿对接收的信息进行解码。在本文第一部分，我们主要关注信源与信宿的编解码，探究量化程度、原始图片性质等因素对信源信宿编解码的影响。在本文第二部分，我们主要关注信道噪声以及信道编解码方法对信道传输的影响。

## PART ONE 信源信宿编解码

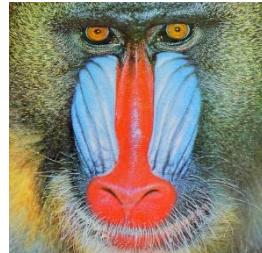
以 MATLAB 为编程语言，我们首先搭建图像的 JPEG 压缩与解压缩程序。我们一致选取.TIF 格式的图片为信源，图片大小均为 512\*512 像素，色彩空间均为 RGB。在本文中，我们共选用了 4 张图片进行分析，如下所示：



Lena.tif



Peppers.tif



Baboon.tif



Brain.tif

以 ISO 发布的 JPEG 静态图像压缩标准<sup>[1,2]</sup>为基础，现将图像压缩与解压缩的基本流程概括梳理如下 [具体算法与代码实现见 Supplementary Materials (下文说明三)]：

1. 读取图片，将 RGB 色彩空间转换为 YCbCr 空间（YUV 空间）
2. 按照从左到右、从上到下的顺序对原始图像进行分块，每个图像块（Block）大小为 8\*8 像素，对于 512\*512 像素大小的原始图像，共 4096 个 Block
3. 对每个 Block 进行正向离散余弦变换（FDCT 变换），将原始数据分解为直流分量（DC）和交流分量（AC），每个 Block 对应 1 个 DC 分量与 63 个 AC 分量
4. 分别对 FDCT 变换后的 Y 分量数据和 CbCr 分量数据进行量化（Quantization）

5. 对 DC 分量进行差分脉冲编码调制(DPCM), 对 AC 分量进行游程长度编码(RLE), 分别得到二者的中间表达式, DC 分量的中间表达式包含 Size 和 Value 两部分, AC 分量的中间表达式包含 Run/Size 和 Value 两部分
6. 分别对上述分量进行 Huffman 编码, 最终得到传输给信道的码字
7. 信宿接收码字, 依次读取字节, 逐个查找 Huffman 表进行解码
8. 对解码后图像依次进行反量化、反向离散余弦变换(IDCT 变换)、色彩空间转换, 最终得到恢复后的图像

**说明一:**通常在步骤 1 之后还可对 YCbCr 图像进行下采样 (Subsampling), 这样可以进一步提高压缩效率而只对图片质量造成很小的影响。下采样分为 4:4:4、4:2:2、4:1:1、4:2:0 等多种类型, 为了减少变化因素, 使研究结果更明确, 我们在本文中一致不进行下采样 (也即 4:4:4 的类型)。

**说明二:**JPEG 压缩标准对 Value 分量的编码采取变长整数编码 (VLI 编码), 由于在代码中输入 VLI 码表太过繁琐, 我们对于 Value 值也同样采取了 Huffman 编码。

**说明三:**我们在本文末附有完整的 MATLAB 代码以及丰富的备注, 其中还包含有较为详细的 JPEG 编码原理讲解 [Supplementary Materials]. 我们同样提供可运行的完整示例代码, 见附件 JACS mlx, 该附件为 MATLAB Live Script 文件, 需用 MATLAB 2016a 及以上版本打开, 其内容和 Supplementary Materials 完全等同。若无相应 MATLAB 版本, 可运行附件 JACS.m, 即普通的 MATLAB 脚本。普通脚本功能和 JACS mlx 完全等同, 只是不包含 JPEG 编码原理讲解, 因此可对应地参考 Supplementary Materials 内容。运行 JACS.m 文件时应保证在同一文件夹下不存在同名的 JACS mlx 文件。

**说明四:**本项目使用的所有代码均由 王一恒 (Penicillin) 编写, 可访问 [\[GitHub\]](#) 进行下载, 使用需遵循 MIT 开源协议 [Copyright 2017-2018 Yiheng Wang Penicillin] [\[The MIT License\]](#).

## SECTION ONE 量化程度对信源编码的影响

在量化步骤中, 我们采用 ISO 所提供的量化表示例<sup>[1]</sup>作为基准量化表, 如下所示 :

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

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

Table One | 基准量化表

左表对应 Y 空间, 右表对应 CbCr 空间。Y 量化表所含数值一般小于 CbCr 空间, 因为人眼对于 Y 空间的信息(明度)更加敏感, 对 CbCr 空间的信息(色度)相对不敏感, 因此可对 CbCr 空间的信息

进行较大程度的压缩。同时量化表的数值从左上角到右下角一般会逐渐增大, 因为每一个 Block 经过 FDCT 变换之后, 左上角的数据代表低频信息(第一个数据代表直流分量), 右下角的数据代表高频信息, 而人眼对于低频信息的敏感度要高于高频信息。

定性的来说，量化表所含数值越大，压缩效率越高，同时图片质量则会下降。将上表（记为  $Q_0$ ）作为基准，我们定义质量因子（Quality Factor, QF）为  $Q_0$  与某单次进行图像压缩所实际采用的量化表（记为  $Q$ ）之间的比值（对所有 64 个值进行均等缩放），即：

$$QF = Q_0 / Q$$

$QF$  值越大，代表  $Q$  越小，所得图片质量也就越高。改变  $QF$  值的大小将会影响图片质量、压缩效率、乃至压缩程序运行所需时间等。我们分别选取  $QF$  值为  $[1/16, 1/4, 1/2, 1, 2, 4, 8, 16]$ ，对上述 4 张图片进行信源编码，所得结果如下：

## 一、程序运行时间

如 Figure 1 左图所示，我们将压缩时间对质量因子的对数值(以 2 为底)作图，结果呈现类似于指数形式的函数关系，因此进一步对压缩时间也取对数值(以 2 为底)，如右图所示，关系函数仍然具有类似凸函数(Convex)的趋势。

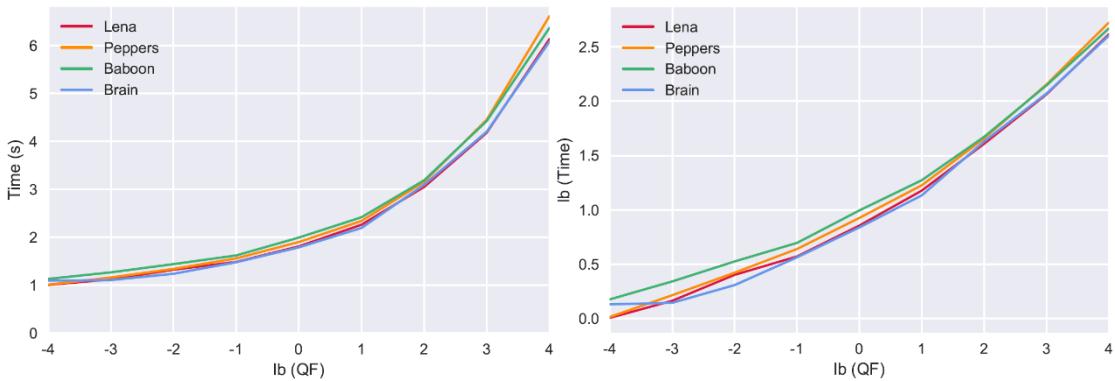
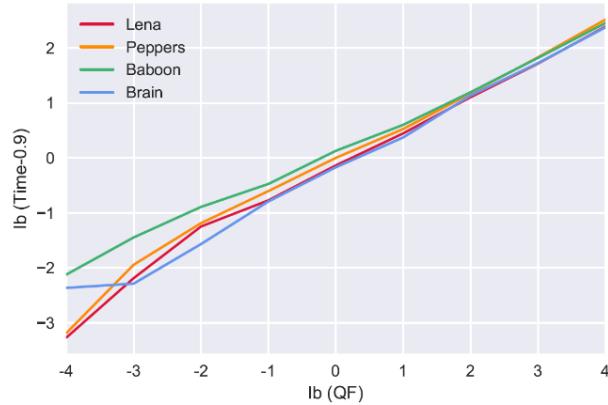


Figure 1 | 质量因子影响图片压缩程序运行时间。左图：压缩时间与质量因子的对数值(以 2 为底)呈现类似于指数形式的函数关系。右图：压缩时间的对数值(以 2 为底)与质量因子的对数值(以 2 为底)之间的关系函数带有轻微的凸函数趋势。四张图片分别用不同的颜色表示。由于信源和压缩算法均确定，程序运行结果仅取决于计算机状态，因此重复实验并无较大意义，图示单次实验的结果。程序运行环境为 MATLAB 2016a，运行系统为 Microsoft Windows 10 (64 位)，处理器为 Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz。注意上述时间为图片压缩所需时间，并不包含解压缩的时间。

经过分析，我们认为程序运行时间可以分解为与图片质量相关的部分  $T$  与不相关的部分  $T_0$ ，后者即为无论图片质量大小均需要的固有运行时间。根据 Figure 1 左图，我们认为 0.9s 是固有运行时间比较合理的大概估计值，因此我们将总运行时间减去  $T_0$ ，从而得到  $T$  的估计值，再将  $lb(T)$  对  $lb(QF)$  作图，所得结果如 Figure 2 所示，可以看出它们呈现较好的线性关系，也即  $T$  和  $QF$  呈幂函数关系：

$$T = \alpha * QF^k$$

其中  $k$  值即线性函数的斜率，这里斜率位于 0.6 左右(小于 1)， $\alpha$  接近于 1(线性函数基本经过原点)。等价的说，即我们的算法对图片的压缩时间和所采用量化表的(平均)数值大小呈指数小于 1 的幂函数关系。不过要注意这里得出的仅仅是粗略的定性关系，并且具有很强的相对性(量化表标准的选取，质量因子的定义等)，确切的函数关系还需要更细致的验证。



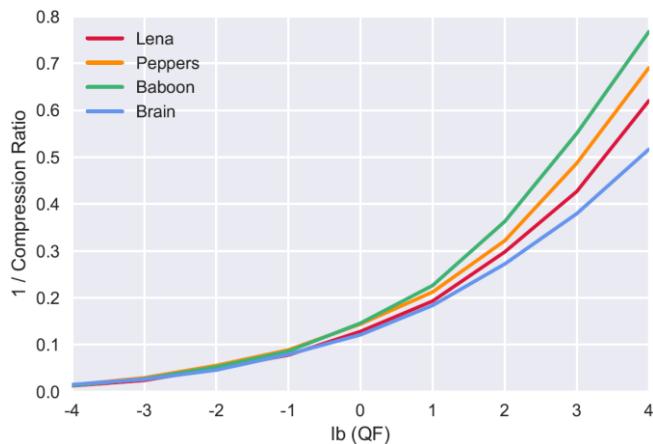
**Figure 2 |** 图片压缩程序运行时间的对数值与质量因子的对数值呈线性关系。将程序运行时间减去固有运行时间的估计值 0.9s，再取对数值(以 2 为底)，并将其对  $\text{lb}(\text{QF})$  作图，二者近似呈线性关系，并且斜率小于 1，纵截距接近于 0.

## 二、图像压缩程度

定义压缩率 (Compression Ratio) 为

$$\text{Compression Ratio} = \frac{\text{BIT}_{\text{Uncompressed}}}{\text{BIT}_{\text{Compressed}}}$$

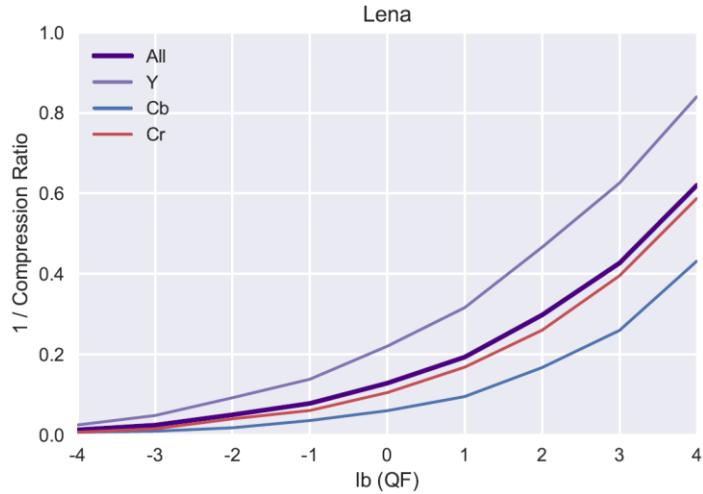
其中  $\text{BIT}_{\text{Uncompressed}}$  为  $512 \times 512 \times 8 \times 3$  bits， $512 \times 512$  代表图像大小，8 代表每个像素用 8bits 表示，3 代表 YCbCr 共 3 个色彩空间， $\text{BIT}_{\text{Compressed}}$  为压缩后所得总码字长度，这里我们首先将 YCbCr 三个色彩空间共同考虑。压缩率越高说明压缩程度越高，压缩率的倒数则反映所得码字的长度大小。为直观起见，我们将压缩率的倒数对  $\text{lb}(\text{QF})$  作图，所得结果如 Figure 3 所示。



**Figure 3 |** 图片压缩率的倒数值与质量因子对数值的关系。随着质量因子的增大，压缩率的倒数值也逐渐增大，即压缩程度逐渐降低，最终压缩率的倒数值会接近于 1，即无压缩的状态。要注意图示关系仅仅对一定范围内的 QF 值成立，因为 QF 值过大会导致量化表数值小于 1，QF 值过小则会导致量化之后的图像数据值全为 0，这两种情况都是不现实的。

对于压缩率倒数的对数值、压缩率、压缩率的对数值，我们都能对  $\text{lb}(\text{QF})$  作图，从结果中可以推测，对于合理范围内的 QF 值，压缩率 (或者其倒数) 的对数值与 QF 值也呈现一定的线性关系 [Supplementary Figure 1].

对于 YCbCr 三个色彩空间，压缩率也是不同的，以 Lena.tif 为例，将不同色彩空间所对应的压缩率的倒数对  $\text{lb}(\text{QF})$  作图，如 Figure 4 所示，四张图片分别对应的结果见 [Supplementary Figure 2]. 需要注意，将 RGB 空间转换至 YCbCr 空间的算法是多样的，对应的结果自然也会不同。由于历史原因，我们的程序所使用的算法与 JPEG 标准算法有细微的差异 [Supplementary Materials].



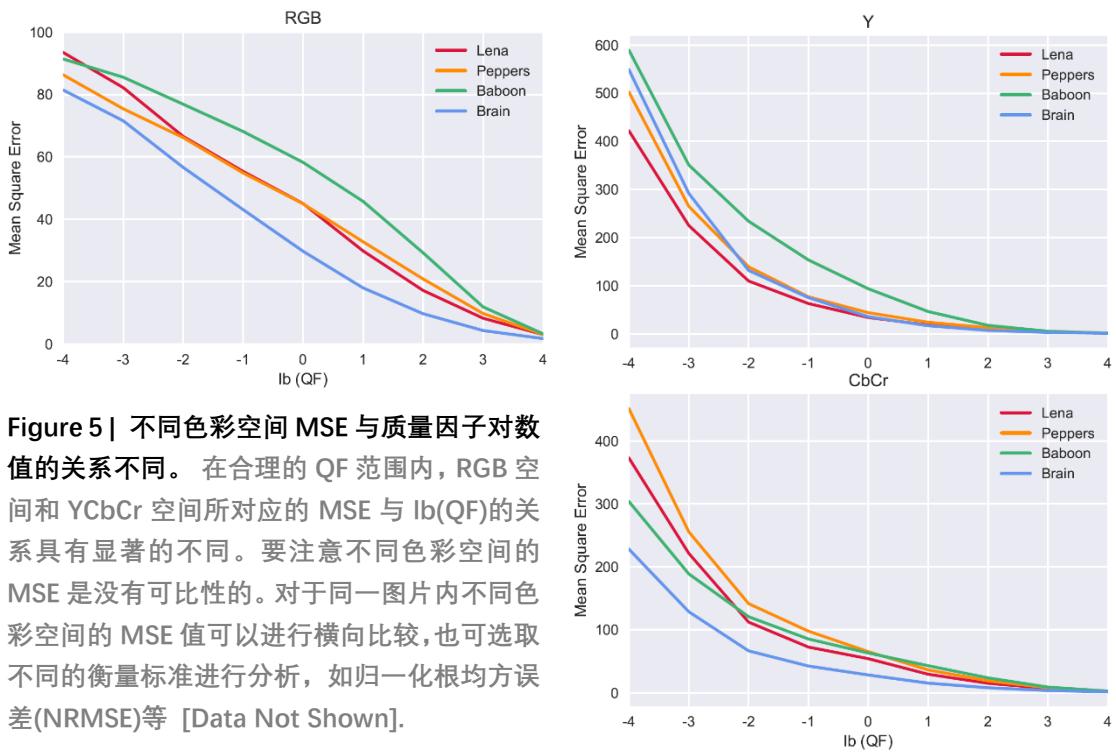
**Figure 4 |** 不同色彩空间压缩率的倒数值与质量因子对数值的关系。色彩空间不同，压缩率也不同。通常来说 Y 空间的压缩程度相对更小，因为其使用的量化表数值较小。CbCr 空间的压缩程度相对更大，而 Cb 和 Cr 二者之间的差异则取决于图像的性质。

### 三、图像质量

我们以均方误差 (MSE) 为作为衡量图像质量的标准，MSE 定义为

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (\hat{x}_i - x_i)^2$$

对于本文所研究的例子，我们对  $X$  的数值首先取绝对值（经过我们的的算法，YCbCr 空间的值可能为负数）。对于 RGB 色彩空间，我们对三个空间不作区分，而对于 YCbCr 空间，我们将 Y 空间和 CbCr 空间区分开。将 MSE 对  $lb(QF)$  作图，所得结果如 Figure 5 所示。可见 QF 值越大，即图片质量越高，对应的 MSE 越小。而 RGB 空间和 YCbCr 空间所对应的 MSE 与  $lb(QF)$  的关系具有显著的不同。在合理的 QF 范围内，RGB 空间对应的 MSE 关于  $lb(QF)$  近似于线性地下降，而 YCbCr 空间对应的 MSE 关于  $lb(QF)$  近似于指数性地下降。此现象背后的数学原理尚待分析。



**Figure 5 |** 不同色彩空间 MSE 与质量因子对数值的关系不同。在合理的 QF 范围内，RGB 空间和 YCbCr 空间所对应的 MSE 与  $lb(QF)$  的关系具有显著的不同。要注意不同色彩空间的 MSE 是没有可比性的。对于同一图片内不同色彩空间的 MSE 值可以进行横向比较，也可选取不同的衡量标准进行分析，如归一化根均方误差(NRMSE)等 [Data Not Shown].

四、图像解压缩结果见 [Supplementary Figure 3]

## SECTION TWO 图像性质对信源编码的影响

我们知道，JPEG 图像压缩的主要原理之一便在于对高频信号的过滤。由于一般图像的相邻像素之间都具有很高的相关性，高频成分很低，因此经过 FDCT 变换和量化之后，强度较低的高频信号得到了很大程度的过滤，从而实现对图像的压缩。具体的算法则体现在 AC 分量的 RLE 编码上。另外一点还在于相邻的 Block 之间也具有比较高的相关性，它们所对应的 DC 分量差距也不大，因此对 DC 分量采取了 DPCM。为了研究这两种因素对于图像压缩的影响，我们采取对原始图像进行混乱处理（Shuffle）的方法，以增加高频成分。具体过程为将原始图像进行切割，分成同等大小的图像块，定义 Cut Size 为图像块的边长（单位为 Pixel），对所得的图像块进行随机排列，重新组成原始大小的图像。我们选取的 4 张图像大小均为 512\*512，因此非常便于处理，所分割成的图像块长宽也相等。以 Lena.tif 为例，Shuffle 的图示见 Figure 6.

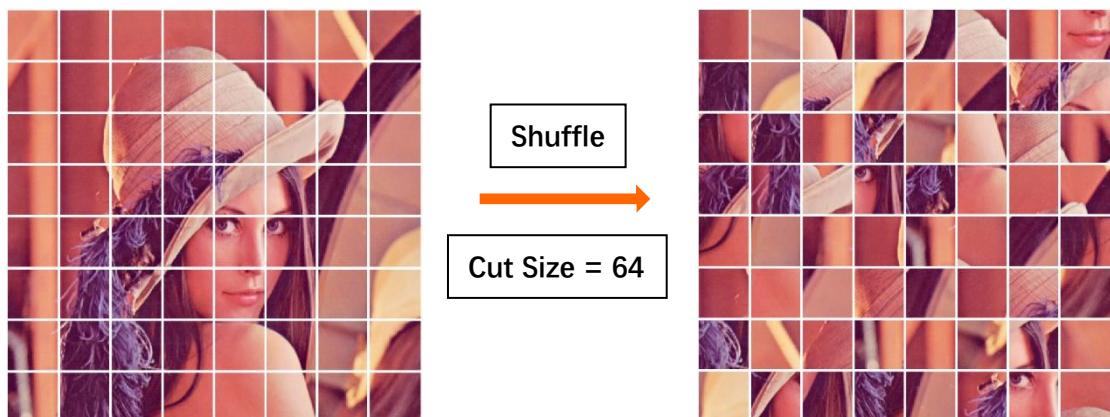


Figure 6 | Shuffle 图示。 左图：原始图片，选择 Cut Size 为 64 个像素对图像进行切割，分成 64 个图像块。右图：Shuffle 后图片， 将 64 个图像块随机排列之后再重新组合成原始大小的图像。白色边缘为便于理解所添加，在实际图片中并不存在。

选取 Cut Size 分别为 [512, 128, 64, 32, 16, 8, 4, 2, 1]，从左到右代表的混乱程度逐渐升高，Cut Size 为 512 时即为原始图像，为 1 时所有像素点均被打乱。我们研究不同混乱程度对于压缩时间和压缩效率的影响。注意前面提到，由于信源和压缩算法均确定，程序运行结果仅取决于计算机状态，因此重复实验并无较大意义。但这里每次 Shuffle 操作得到的图片都不同，即信源不同。我们对每个混乱程度进行 99 次独立实验，所得结果如 Figure 7 所示。

可以明显看出，在 Cut Size 大于 8（图中红色方框所示）的情况下，压缩时间均相对稳定在较小的范围内。同时压缩效率也较高，并且在 99 次独立实验中几乎没有变化；而当 Cut Size 小于 8 时，压缩时间出现显著的上升，压缩效率则显著下降（体现在 1 / Compression Rate）的显著上升。

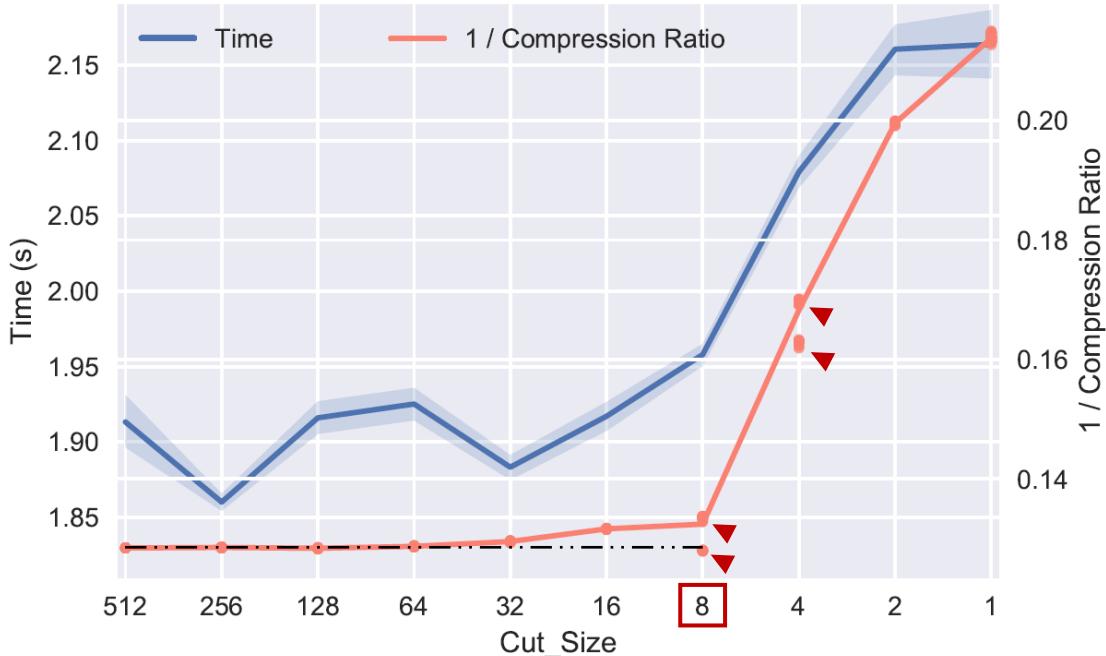


Figure 7 | 图像性质(混乱程度)对压缩时间和压缩效率的影响。对每个混乱程度(Cut Size)显示 99 次独立实验的统计结果。对于压缩时间, 实线表示平均值, 半透明背景表示由 Bootstrap 方法得到的 68% 置信区间 [可参考 [seaborn.tsplot 官方文档](#)]. 对于压缩效率(同样采用压缩率的倒数表示), 实线表示平均值, 实心点代表每次实验所得结果。

分析可知, Cut Size = 8 对应于每个  $8 \times 8$  的 Block (FDCT 变换和量化的最小单元) 进行随机排列 [Figure 8]. 当 Cut Size 大于等于 8 时, 每个 Block 内部的结构并没有发生变化, 这意味着 AC 分量没有发生变化。DC 分量本身也不会发生变化, 但由于 Block 与 Block 之间的排列发生了重组, 将导致 DC 分量进行 DPCM 的结果不同; 而当 Cut Size 小于 8 时, 原始的 Block 结构被打乱, 导致相邻像素点之间的相关性降低, 编码过程中一个 Block 内高频分量的强度上升, 最终导致 JPEG 压缩方法的压缩时间上升而压缩效率下降。

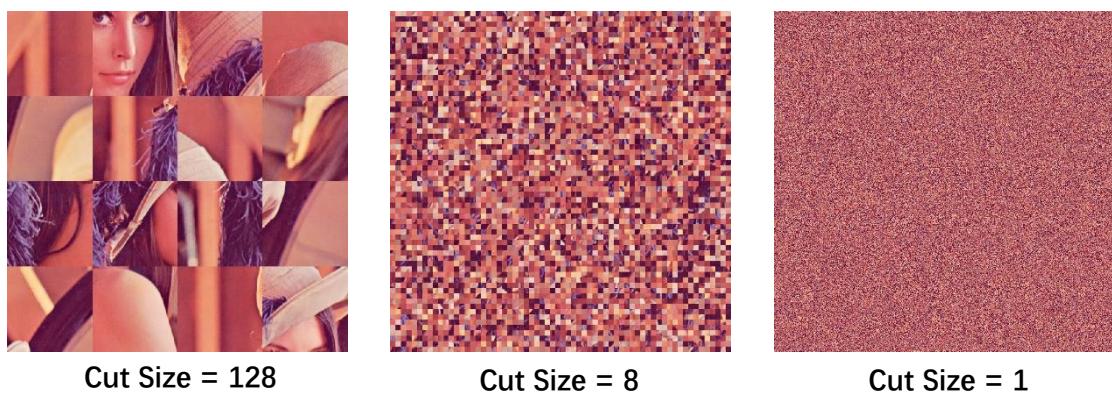


Figure 8 | Cut Size 对图像性质的影响。Cut Size = 128 时, 对于压缩算法而言图片结构几乎没有发生变化, AC 分量与原图完全相同, DC 分量经 DPCM 后的结果差异也非常小。Cut Size = 8 时, DC 分量经 DPCM 后的结果差异较大, 反映在图像中即亮块和暗块杂乱排列, 但单个 Block 内的 DC 分量和 AC 分量都没有发生变化。Cut Size = 1 时, 图片被完全打乱, 单个 Block 内高频分量强度大幅提高, 但实际上 DC 分量变得非常均匀。对应于所有 Cut Size 的图片示例见 Supplementary Figure 4.

由此可以初步推断，压缩时间主要取决于对 AC 分量的编码，这与从计算量上的直观分析结果也是相符的。同样，压缩效率也主要取决于 AC 分量，当 Cut Size 大于 8 时 DC 分量进行 DPCM 所得结果差异相对很小，导致 99 次试验结果几乎完全相同 [Figure 7].

由于压缩时间会受到计算机状态的影响，引入不可控因素，我们现在单独选取编码效率作为研究对象。经过仔细观察编码效率的变化情况，Figure 7 所示实验现象还能够揭示出更为深刻的结论。

首先，通过以上分析我们已经确定当 Cut Size 大于等于 8 时，AC 分量本身的数值完全不受影响，这会同时产生两个结果，一是 99 次实验所得编码效率应该是完全相同的（即组内方差为 0），二是改变 Cut Size 应该不会影响编码效率（即组间方差为 0，如图中黑色水平线所示）。但是我们可以看到，当 Cut Size 低于 64 时编码效率便有所下降，因此唯一可能的原因便在于 DC 分量。不过当 Cut Size 大于 8 时，上述的第一个结果是基本满足的，说明 Cut Size 大于 8 时 DC 分量所带来的组内方差也较小。

更加有趣的是，当 Cut Size = 8 时我们发现编码效率出现了不连续的分化，如图中红色箭头所指，可能意味着存在非连续型随机变量在影响编码效率。当 Cut Size = 4 时也有同样的现象出现，而当 Cut Size = 2 或 1 时这种分化现象便消失了。当 Cut Size 小于 8 时，AC 分量会引入方差，但当 Cut Size = 8 时，这种现象则只可能由 DC 分量引起。

通过推理可知，AC 分量携带的高频成分强度随 Cut Size 的减小在统计上应该是单调上升的。但 DC 分量却不是如此。当 Cut Size 较大时，DC 分量的变化较为平缓，这是由原始图像本身的性质决定的；当 Cut Size 逐渐减小，原始图像被打乱的程度逐渐增大，这便会导致 DC 分量的波动频率和幅值升高；然而当 Cut Size 小于一个 Block 的边长 8 时，每个 Block 中都会含有多个随机分配的切割后图像块，这实际上引入了平均效应，最终使得不同 Block 之间的 DC 分量趋于均一化 [Figure 8]. 由此分析我们可以推断，DC 分量的波动频率和幅值在 Cut Size 和一个 Block 的边长 8 刚好相等时会达到最大，Cut Size 向两侧偏离此值都会导致其波动频率和幅值下降。进而我们可以预测，如果我们钳制住 AC 分量 (Fixed AC)，使其不受 Shuffle 操作的影响，那么所得 1 / Compression Ratio 曲线将不会出现 Cut Size 小于 8 时的显著上升，相反，曲线在 Cut Size = 8 时取得最大值后会发生下降 [预测 1].

要注意的是，DC 分量的波动频率和幅值升高的直接效应该是编码效率的下降，这可以解释 Cut Size = 32, 16, 8 时 Figure 7 中曲线的逐渐上升过程，但这并不能解释 Cut Size = 8 时编码效率出现的不连续分化现象，更何况分化出的一部分对应的编码效率甚至高于原始图像 [黑色水平线标识原始图像编码效率，Cut Size = 8 时存在部分数据点低于该水平线的，说明这一部分数据点对应编码效率更高].

通过分析，我们初步认为此分化现象出现的原因在于，当 Cut Size = 8 时，原始图像被刚好切割成与一个 Block 同样大小的图像块，这些图像块的随机排列即完全对应于编码过程中 DC 分量的排列，那么一定存在某些排列方式使得总的编码效率不高于原图像，而另一些排列方式使得总的编码效率不低于原图像。对于 Lena.tif 的结果，由于 Cut Size = 8 时存在低于黑色水平线的数据点，这说明原始图像中 Block 的排列方式并不是最佳的。然而当 Cut Size > 8 时，图像块较大，由于破坏了原始图像的连续性，图像块交界处便会引入 DC 分量的波动，同时由于图像块数目小，可能的排列情况比起 Cut Size =

8 时要少，那么从中找出编码效率显著高于原图像这一事件发生的概率就会更小 [这里的“显著”，我们姑且认为是在 Figure 7 等图中肉眼可分辨]. 对于 AC 分量，类似的分析同样成立，只是 Cut Size 小于 8 时才会对 AC 分量产生影响。实际上，结合上文关于 DC 分量在 Cut Size 小于 8 时的行为的分析，我们可以推断 Cut Size = 4 时曲线所出现的不连续分化正是来源于 AC 分量。由此我们预测，如果我们钳制住 DC 分量 (Fixed DC)，使其不受 Shuffle 操作的影响，那么所得  $1 / \text{Compression Ratio}$  曲线将不会出现 Cut Size = 8 时的不连续分化，但在 Cut Size = 4 时的不连续分化仍然存在 [预测 2].

我们根据以上分析提出了两点预测，下面我们分别对 AC 分量和 DC 分量进行钳制，重新研究 Cut Size 对编码效率的影响。实验方法与上文完全一致，只是在 Fixed AC 情形下，每次实验都采取和原始图像相同的 AC 分量；同理，在 Fixed DC 情形下，每次试验都采取和原始图像相同的 DC 分量。实验结果如 Figure 9 所示。

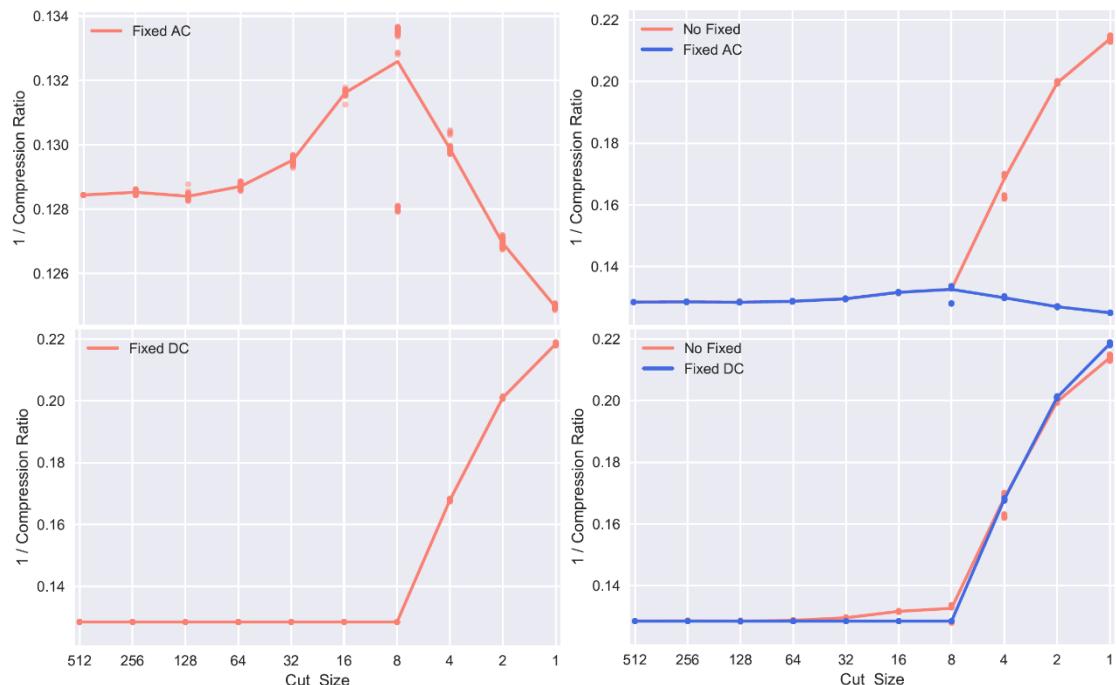


Figure 9 | Cut Size 分别在 Fixed DC 和 Fixed AC 的情况下对 R 值的影响。左上图：Cut Size 在 Fixed AC 情况下对编码效率的影响。左下图：Cut Size 在 Fixed DC 情况下对编码效率的影响。右侧图为 Figure 7 中的 R 值分别与 Fixed AC 和 Fixed DC 情况下的 R 值共同显示。

首先分析 Fixed AC 的情况 (Figure 9 左上)。为了叙述方便，我们在这里将  $1 / \text{Compression Ratio}$  称为 R 值，将 Cut Size 称之为 C. 可见，当 C 大于 8 时 DC 分量引起的 R 值呈逐渐上升趋势；在 C = 8 时达到最大，同时我们可以看到显著的分化现象；而 C 小于 8 时则逐渐下降，甚至最终低于原始图像的水平，说明小的 C 值引起了显著的平均效应，使得 Block 与 Block 之间的 DC 分量趋于平稳。要注意的是，在 C = 128 时同样存在 R 值小于 R<sub>0</sub> 的数据点，这意味着此时便存在编码效率高于原始图像的排列方式，只是影响非常小。此实验结果与我们做出的预测 1 是相符的。

在 Fixed DC 的情况下 (Figure 9 左下)，C 大于等于 8 时 R 值稳定不变，这是必然的，因为 DC 分量被钳制，而 AC 分量不受 Shuffle 的影响。C 小于 8 时 R 值则出现显著的上升。在 C = 8 时不存在分化现象。这与预测 2 也是相符的。但是一个关键的问题在于，

$C = 4$  时的分化也消失了。虽然 Fixed AC 的情况下  $C = 4$  时存在一定程度的分化，但从幅度上看是完全不足以解释 Figure 7 中的分化现象的（在 Figure 9 右上图即可看出）。

这是否意味着我们的理论是错误的呢？通过进一步分析我们提出，由于总码字长度由 DC 分量和 AC 分量的码字两部分组成，那么 R 值便能分解为两个分量的算术和，即

$$R = R_{DC} + R_{AC}$$

另外，在 DC 分量和 AC 分量分别被钳制的情况下，它们都对应于一个恒定的 R 值，我们将其记为  $R_{DC0}$  和  $R_{AC0}$ ，同时另一个分量所引起的 R 值则是关于 C 的变量，我们将其记为  $R_{AC}(C)$  和  $R_{DC}(C)$ 。这样一来，在 Fixed AC、Fixed DC 以及不进行钳制这三种情况下，我们分别可以得出

$$R_{Fixed\ AC} = R_{DC}(C) + R_{AC0}$$

$$R_{Fixed\ DC} = R_{AC}(C) + R_{DC0}$$

$$R(C) = R_{AC}(C) + R_{DC}(C)$$

将以上第三式左右交换，并与前两式求和即可得

$$R(C) = R_{Fixed\ AC} + R_{Fixed\ DC} - (R_{AC0} + R_{DC0})$$

而括号中的项即为  $C = 0$  时的 R 值，因为钳制的状态即为原始图像的状态。因此我们可以将两种钳制状态下的 R 值求和，并减去  $R(C=0)$ ，再与  $R(C)$  共同显示，它们应该是相同的。然而实际结果如 Figure 10 所示。

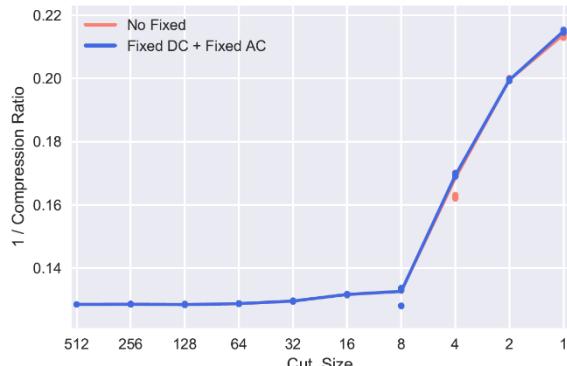


Figure 10 | 不进行钳制与分别钳制 DC 和 AC 分量所得 R 值之间并不吻合。 $R(C) = R_{Fixed\ AC} + R_{Fixed\ DC} - (R_{AC0} + R_{DC0})$  这一关系式并不成立，在  $C = 4$  和  $C = 2$  时可以看出明显的差异，预示着上文做出的假设存在错误。

这一结果说明前面我们所给出的三个等式存在问题。仔细分析我们发现，这个问题在于 Fixed AC 和 Fixed DC 以及不进行钳制这三种情况下， $R_{DC/AC}(C)$  是不同的， $R_{DC0}$  和  $R_{AC0}$  也并不恒定。其原因在于算法层面。在本文开头便提到（说明二）对 DC 和 AC 的 Value 值我们共同进行 Huffman 编码，这意味着 DC 分量和 AC 分量共享同一个码表。而钳制住 DC 分量和不钳制 DC 分量所得到的码表是不同的，这便会同时影响到对 AC 分量的编码。例如，当 DC 分量被钳制时，若 AC 分量也不变化( $C > 8$ )，那么  $R_{DC0}$  将是恒定的，但当  $C < 8$ ，AC 分量发生变化，便会通过改变 Huffman 码表的方式改变  $R_{DC0}$ 。同样的，钳制 AC 分量与不钳制 AC 分量的情况下  $R_{DC}$  随着 C 的变化情况也是不同的，即  $R_{DC}(C)$  不同。也就是说 DC 分量和 AC 分量不能完全分离，这是导致以上等式不成立的根本原因 [注意在 JPEG 的压缩标准中，对 DC 分量和 AC 分量同时采用 VLI 编码，虽然它们共享同一个码表，但是由于该码表始终是恒定不变的，因此这种情况下 DC 分量和 AC 分量仍然可以分离]。

注意到这一点之后，我们便需要重新审视一下前面做出的预测。我们犯的关键错误在于忽略了 DC 分量和 AC 分量之间的相互影响，而其最本质的后果是对于 R 值的影响因素产生了混淆。如何理解呢？以  $C = 8$  为例，我们分析得出此时 AC 分量与原始图像是完全相同的，这一结论确凿无疑，从而我们认为  $C = 8$  时 R 值的分化现象是由 DC 分量引起，这也是逻辑上的必然推理。但需要注意的是，“由 DC 分量所引起的效应”并不等同于“ $R_{DC}$  的效应”，而在前面的预测中我们混淆了这两个概念。

重新对实验现象进行解释，我们现在给出正确的分析。 $C = 8$  时 R 值出现的分化是由 DC 分量所引起的，因为在 Fixed DC 的情况下该分化消失了，而在 Fixed AC 的情况下该分化依然存在 [Figure 9]。但对于  $C = 4$  时 R 值出现的分化，分别在 Fixed DC 和 Fixed AC 的情况下它都消失了，这说明 DC 分量和 AC 分量是导致这一处分化的共同原因。然而对于导致这两处分化的“直接原因”我们还并不清楚，也就是说我们还想要知道是  $R_{DC}$  和  $R_{AC}$  中的哪个部分促成了分化现象的产生，亦或是它们的共同效应。因此我们必须在 No Fixed 的情况下将  $R_{DC}$  和  $R_{AC}$  单独提取出来，结果如 Figure 11 所示。

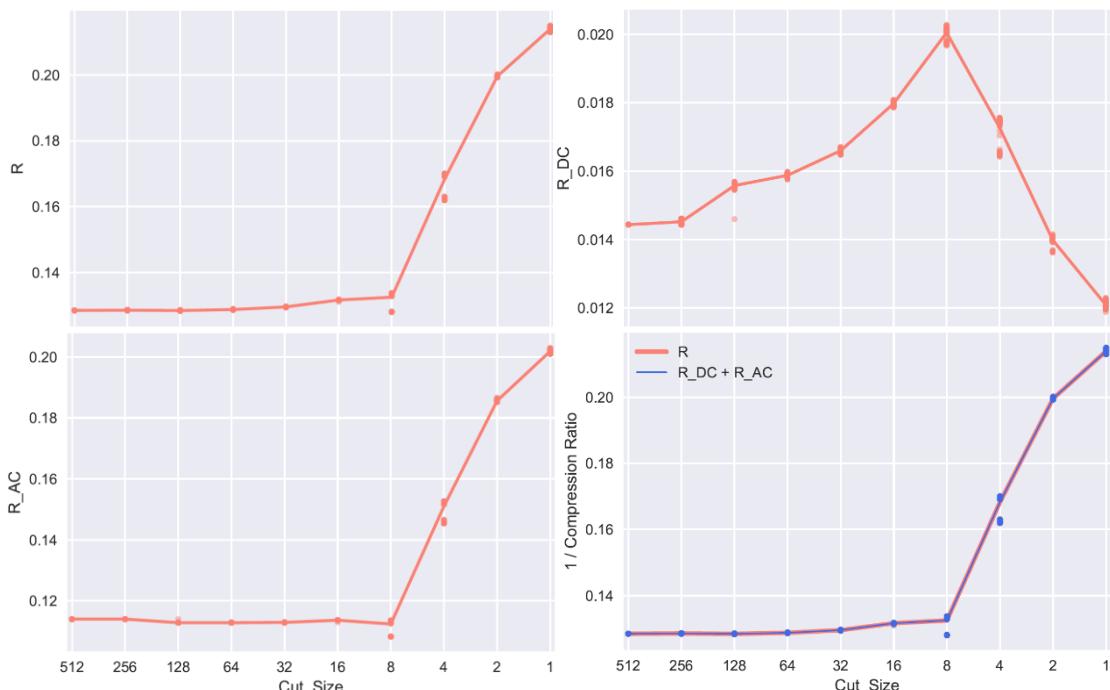


Figure 11 | Cut Size 分别对  $R_{DC}$  和  $R_{AC}$  的影响。左上图：Cut Size 对 R 值的影响。右上图：Cut Size 对  $R_{DC}$  的影响。左下图：Cut Size 对  $R_{AC}$  的影响。右下图： $R_{DC} + R_{AC}$  与 R 值共同显示。

Figure 11 左上图即与 Figure 7 中相同。在右上图中 [注意它与 Figure 9 左上图表达的是完全不同的含义] 我们可以看出  $R_{DC}$  随 C 值的减小逐渐上升，在  $C = 8$  时达到最大值，之后则发生下降，甚至低于原始水平。这一结果才真正说明我们得出预测 1 所作出的推理是正确的，尽管由这些推理得出预测 1 是不严谨的。在左下图中，C 大于 8 时  $R_{AC}$  便存在波动，注意到此时 AC 分量是恒定的，这便确凿地说明 DC 分量的变化通过 Huffman 表影响了对 AC 分量的编码。右下图说明  $R = R_{DC} + R_{AC}$ ，这是必然成立的。

最关键的是，在  $R_{DC}$  中并不存在分化现象，而  $C = 8$  和  $C = 4$  的分化均出现在  $R_{AC}$  中。结合 Figure 9 中的结果，这说明分化现象产生根本原因在于 DC 分量的变化，而直接原因是 DC 分量的变化引起 Huffman 表的变化，从而在  $R_{AC}$  中产生了明显的分化 ( $R_{DC}$  本身的分化是不显著的)，进而体现在了 R 中。

综合 Figure 7-12 的实验结果以及以上的分析，现在我们可以对 Figure 7 中的诸多现象做出一个比较完整的解释了：

首先关于 R 值的总体变化趋势。当 Cut Size 大于 8 的时候，AC 分量是完全不变的，而随着 Cut Size 的逐渐减小，由于图片的混乱程度逐渐增大，对 DC 分量的编码效率会逐渐降低，同时 DC 分量的变化会通过 Huffman 表为介质影响对 AC 分量的编码效率，也就是说  $R_{DC}$  和  $R_{AC}$  同时在发生改变，但前者占据主导地位，这便引起了总的 R 值逐渐上升的过程。当 Cut Size = 8 时  $R_{DC}$  达到最大值，随后则由于混乱程度继续增大，引起了平均效应，从而导致  $R_{DC}$  逐渐减小，甚至低于原始水平。而 Cut Size 小于 8 时 AC 分量开始发生变化，高频分量的强度上升，并且这一效应占据了主导地位，从而导致总的 R 值显著上升。

然后便是关于分化现象。当 Cut Size = 8 时，存在一部分图像块的排列方式使得 DC 分量呈现出特定的性质，这一部分 DC 分量通过影响 Huffman 表，从而导致 AC 分量的编码效率显著升高，引起 R 值的分化。Cut Size = 4 时，DC 分量和 AC 分量均有随机性，它们共同影响 Huffman 表，并且也存在一部分特定的排列方式导致 AC 分量的编码效率显著升高，进而使得 R 值出现分化。它们的共同点都在于 AC 分量的编码效率是影响 R 值的主导因素，这是与 AC 分量对应的码字占总码字的大部分这一事实相吻合的，但同时 DC 分量的排列情况则是诱导分化出现的关键因素。

这样一来，我们还可以根据这一理论给出新的预测。我们预测，在 No Fixed 的情况下，如果 DC 分量和 AC 分量不共享同一个码表，即如果我们切断它们之间的联系，那么 Figure 7 中 R 值的变化趋势并不会发生变化，但是分化现象将会消失。同时如果和 Figure 11 相比较，我们将会看到 DC 分量和 AC 分量之间相互影响的消失。为了验证这一预测，我们更改算法，对 DC 分量和 AC 分量分别进行 Huffman 编码，所得结果如 Figure 12 所示 [当然，我们还可以在对 DC 分量和 AC 分量分别进行 Huffman 编码的情况下，再分别对 DC 分量和 AC 分量进行钳制，只是这样得到的结果并不会提供更多的信息，我们将其展示在 Supplementary Figure 5 中]。

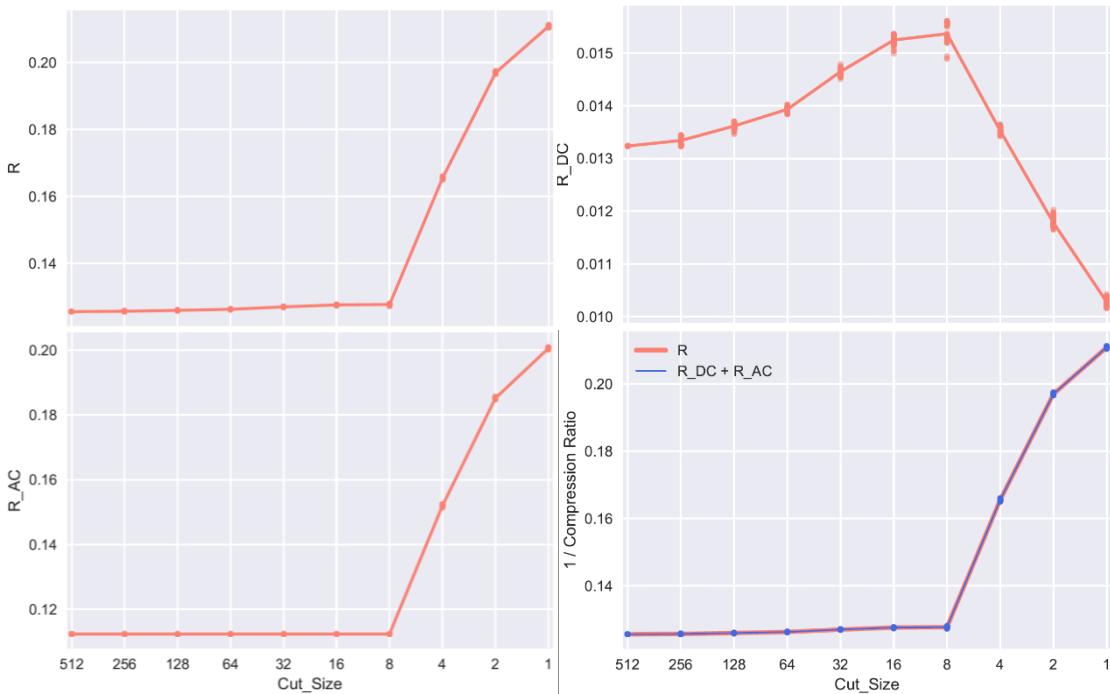


Figure 12 | 对 DC 分量和 AC 分量分别进行 Huffman 编码的情况下，Cut Size 分别对  $R_{DC}$  和  $R_{AC}$  的影响。左上图：Cut Size 对 R 值的影响。右上图：Cut Size 对  $R_{DC}$  的影响。左下图：Cut Size 对  $R_{AC}$  的影响。右下图： $R_{DC} + R_{AC}$  与 R 值共同显示。

可以看出 Figure 7 中 R 值的分化现象消失了[左上]，Figure 11 中 DC 分量与 AC 分量之间的相互影响也消失了，这体现在  $R_{DC}$  和  $R_{AC}$  的绝对大小上[右上，左下]，将使用同一个与不同 Huffman 码表的情况下分别对应的 R 值共同显示可以清楚地看到这一现象，如 Figure 13 所示。在大多数情况下，采用同一码表会导致 DC 分量和 AC 分量编码效率降低，总体编码效率也相应降低。但对于特定的 Cut Size 值，采用同一 Huffman 码表会引起分化现象的出现。

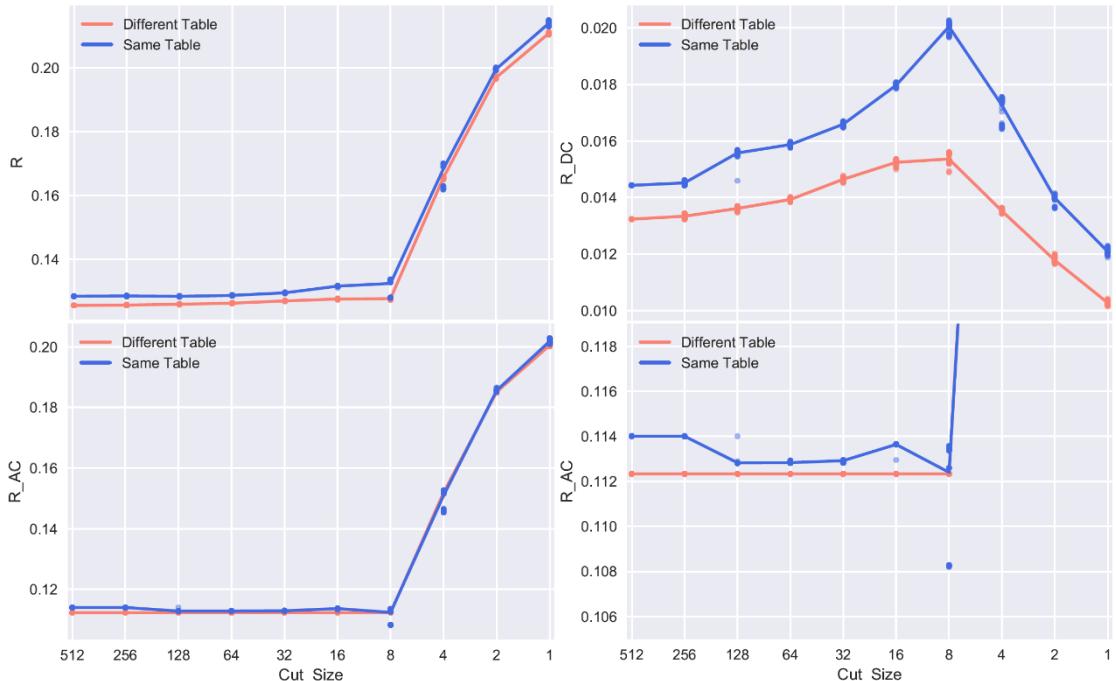


Figure 13 | 采用同一 Huffman 码表与采用不同码表对编码效率的影响。左上图：采用同一 Huffman 码表与采用不同码表对 R 值的影响。右上图：采用同一 Huffman 码表与采用不同码表对  $R_{DC}$  的影响。左下图：采用同一 Huffman 码表与采用不同码表对  $R_{AC}$  的影响。右下图：为左下图的局部放大。

以上所示实验数据充分验证了我们的推测，但到目前为止，我们阐明了 DC 分量和 AC 分量分别对编码效率的影响，以及它们之间的相互作用关系，还有它们与 Shuffle 程度之间的关系，另外还定性地分析了分化现象的存在性问题。但是对于分化现象的产生机制以及它的分布情况和离散性质我们还没有做出合理的解释。因此我们认为需要在现有理论的基础上进一步做出更为定量化的分析。

以  $C = 8$  时的分化现象为例，当 Cut Size 逐渐减小，那么图像块排列的可能数将迅速扩大，并且其增长速度极极极极极极极高 [分别为  $1!$ ,  $4!$ ,  $16!$ ,  $64!$ ,  $256!$ ,  $1024!$ ,  $4096!$  … 即  $(2^{2k})!$ ]。 $4! = 24$ ，而  $16!$  已经达到  $10^{13}$  量级， $64!$  则达到了  $10^{89}$  量级，在 MATLAB 中输入 `>>factorial(256)` 所得结果便已经是 Inf 了！现在我们想要知道某种图像块的排列方式所引起的编码效率显著小于原图像的概率  $n / N$ ，已知的是光速增长的  $N$ ，那么还需要知道  $n$  随 Cut Size 的变化情况。

当 Cut Size 较小时，N 极大，而原始图像已经是最优排列这一事件的概率又是很小的，那么几乎可以百分之百地确定 N 中存在 n 种排列方式使得编码效率显著高于原始图像。另一方面，当 Cut Size 逐渐增大时，N 急剧下降，那么存在某个 Cut Size 的值，使得 N 种排列中不存在使得编码效率显著高于原始图像的排列方式，即  $n = 0$ 。由此可知，一定会存在某个 Cut Size 值，使得  $n / N$  达到最大，在 Figure 7 中，这个 Cut Size 值便是 8，使得我们在此处得到了部分编码效率高于原始图像的数据点。

以上分析有助于我们理解为何在 R 值逐渐上升的大趋势中会突然出现显著小于原始值的情况，即 Cut Size = 8。而更一般的，我们需要关注对任意一个确定的 Cut Size，其所对应的 R 值出现分化的原因。为此，我们将 Figure 7 中 R 值的分布数据提取出来，作图如 Figure 14 所示。

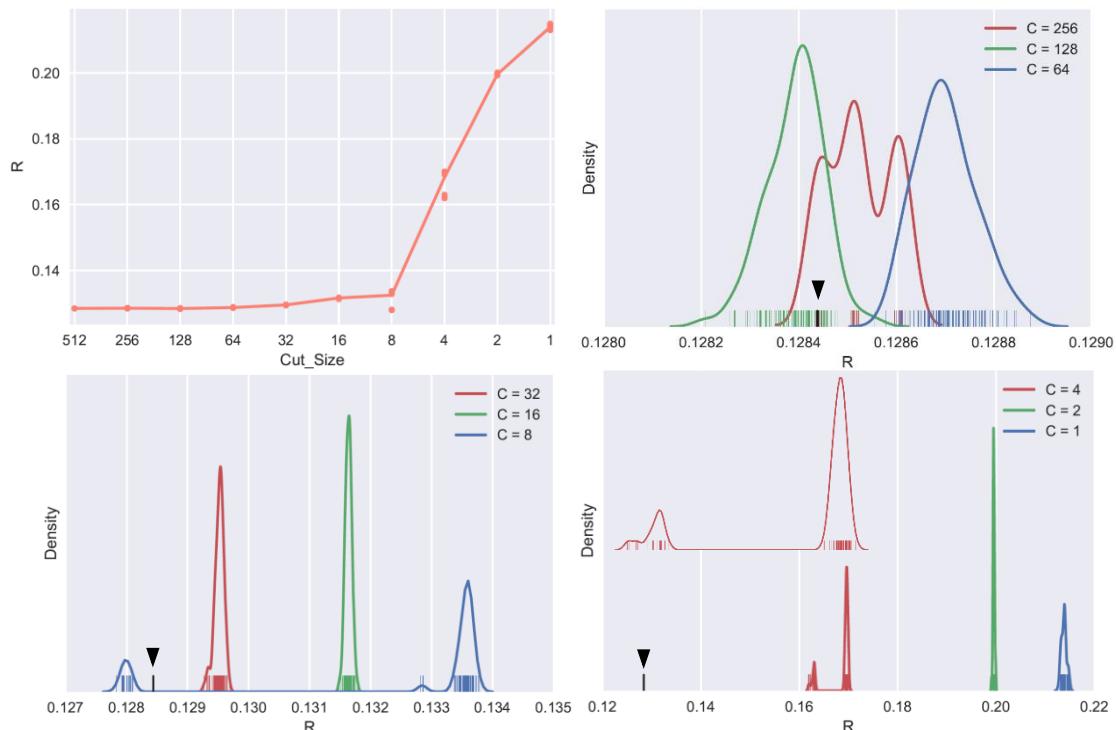


Figure 14 | 不同 Cut Size 条件下 R 值的分布情况。左上图：同 Figure 7，以供参考。右上图：Cut Size = 256, 128, 64。左下图：Cut Size = 32, 16, 8。右下图：Cut Size = 4, 3, 2。Inset 为 Cut Size = 4 时 R 值分布的放大。原始图像对应的 R 值由黑色箭头标出。

由于绝对值差异较大，我们选取临近的 3 个 C 为一组，以便于观察。从 Figure 14 右上图可以看出  $C = 128$  和  $C = 64$  时 R 值基本符合正态分布，前者的均值略小于原始图像的 R 值，而后者则稍大，说明在这两个 C 值下，R 值存在一个均值，而差异来自于不同的 Shuffle 结果。对于  $C = 256$  的情况，看起来并不符合正态分布，这是由于此时的排列情况总共只有  $4! = 16$  种，样本量过小，不足以显现正态分布的特征（中心极限定理），而数据中已经出现了所有的排列情况。左下图中， $C = 32$  和  $C = 16$  时的 R 值也符合正态分布的特征，而  $C = 8$  时出现了显著偏离大部分数据点的分布。右下图类似。

对于这样的分化现象，显然不能用单个符合正态分布的随机变量来描述。我们分析认为，将 R 值作为符合混合分布的随机变量来看待更能说明实验结果，也就是说，R 值满足的概率密度函数为

$$P(R|Y) = \sum_{i=1}^n I_{\{Y=i\}} N(\mu_i, \sigma_i^2); \quad i = 1, 2, \dots, n$$

$$P(Y = i) = \alpha_i; \quad i = 1, 2, \dots, n; \quad \sum_{i=1}^n \alpha_i = 1$$

其中  $N(\mu_i, \sigma_i^2)$  表示均值和方差分别为  $\mu_i$  和  $\sigma_i^2$  的正态分布概率密度函数,  $I_{\{Y=i\}}$  为示性函数 ( $I_{\{Y=i\}} = 1$ , 当且仅当  $Y = i$ , 否则  $I_{\{Y=i\}} = 0$ ), 而  $Y$  为取值范围为 1 到  $n$  之间的整数的离散随机变量。也就是说, 对于任一 Cut Size 值, 其所对应的  $R$  由均值和方差不同的正态分布混合而成, 而在某一次试验中,  $R$  值落入其中某一分布的概率由  $Y$  的分布决定。

当  $C$  不等于 8 和 4 时, 同样可以用混合分布对  $R$  值加以描述, 在这些情况下并无分化现象出现的原因则可能在于只存在一个正态分布, 即混合分布退化为单一的分布, 也有可能是不同的正态分布之间均值相差很小。而对于  $C = 8$  和  $C = 4$  的情况, 我们可以清楚地将  $R$  值分为两簇均值存在显著差异的正态分布。之所以称为两簇, 是因为每一簇都可能单独为一个混合分布, 但从现象上来看我们也可以将其描述为两个正态分布。

以此为基础, 我们便可以认为  $R(C=128)$  对应一个正态分布、 $R(C=64)$  对应另一个正态分布, 而  $R(C=8)$  对应两个均值存在显著差异的正态分布……但是一个关键的认识在于任何在较大  $C$  值的情况下存在的图像块排列在更小的  $C$  值的情况下一定也存在。例如,  $C = 8$  时  $R$  的可能取值范围至少应该涵盖  $C$  更大时所出现的所有情况, 但实际结果却仅仅表现出两个正态分布。事实上, 几乎任何一个  $C$  值所对应的  $R$  值分布都没有在更小的  $C$  值下表现出来。

为了解释这一现象, 我们提出  $R$  值存在一种“涌现”特征。前面提到图像块可能的排列数随着 Cut Size 的减小增长速度极快, 也就是说  $R$  的可能取值数目在急剧的增长, 这样一来, 任何在更大的 Cut Size 中存在的分布在下一个 Cut Size 情况下的发生概率都几乎为 0, 进而无法体现。而对应于某个 Cut Size, 实验数据所表现出的几乎都是新增可能排列所对应的  $R$  值。也就是说, 每次减小 Cut Size 都会涌现出巨大数量的新的  $R$  值, 这些新的  $R$  值同样可以用混合正态分布来描述, 而不同正态分布出现的概率(即  $\alpha_i$ )相差极大, 因此总是会出现占据主导地位的正态分布, 在大部分情况下占据主导地位的分布只有一个, 而当两个  $\alpha_i$  值的差距相对比较小时, 便导致了分化现象的产生。设想如果我们能穷尽所有的排列, 那么所得的  $R$  值则应该表现出连续分布的特性, 但由于  $N$  是个天文数字, 实际所得实验结果便只能显现出“离散随机变量”的特征了。

至此为止, 我们终于比较完美地解释了 Figure 7 中的实验现象, 不过在结束本节之前还有几点需要粗略地说明一下。

首先是本节所得出的结论均基于 Lena.tif, 并且统一选取 Quality Factor 为 1, 因此所得出的一些具体的结论并不全是普适性的, 但其原理却应该是一致的。在本节前半部分提到的 DC 分量和 AC 分量分别关于 Cut Size 的变化规律, 对于绝大多数正常图片来说都应当成立, 特别是 Cut Size = 8 的特殊性, 因为 JPEG 压缩算法选取的 Block 大小即为 8\*8 的。DC 分量和 AC 分量之间的相互影响也应该是普适性的, 例如分别进行 Huffman 编码会提高编码效率 (但在传输过程中则需要传输两个 Huffman 码表)。需要注意的是

Cut Size 和分化现象之间的关系，对于 Lena.tif 来说分化出现在 C = 8 和 4 这两种情况，对于其它图像来说则不一定如此了。另外，如果我们采用不同的 QF 值，那么所得的规律可能会发生质变。

对于 Quality Factor = 1 时四张图片 R 值随 Cut Size 的变化见 Supplementary Figure 6. 可以看出不同的图片各自存在自己特征性的分化，有的在我们研究的 Cut Size 内则没有分化存在。有趣的是，如果我们一开始不是选择 Lena.tif 作为研究对象，那么可能就不会发现 C = 8 和 4 的两处分化，也就很可能没有以上一系列的探究结果了。

对于 Lena.tif 在 QF = [1/8, 1/4, 1/2, 1, 2, 4, 8] 七种情况下 R 值随 Cut Size 的变化见 Supplementary Figure 7. 可以看出 QF 值对于 R 值随 Cut Size 的变化情况是比较显著的。需要注意的是，对于小的 QF 值，Cut Size 较小时甚至会引起 R 值的骤降。关于这一现象的分析较为复杂，因为所要编码的数据（即量化之后的图像）同时受到来自 Cut Size 和 QF 两方面的影响。关键是我们缺少准确预测编码效率的指标，因此较难设计相应的实验进行探究。不过利用本文中用到的研究方法，我们应该能够通过分别进行 Huffman 编码的方式分离出 DC 分量和 AC 分量的效应，以及通过 Fixed 的方式研究它们之间的相互作用，从而对于 R 值骤降这一现象给出一些可能的解释，在本文中我们就不对此做具体分析了。同样有趣的是，如果 ISO 的推荐量化表与现在的量化表有所不同，那么我们可能也不会发现 C = 8 和 4 的两处分化，同样也就很可能没有以上一系列的探究结果了。

第二点在于，如果我们仔细观察 Figure 7-13 中变量的绝对值，再与上一节中的比较，可以发现 Shuffle 对于编码时间和编码效率的影响相比起 QF 值是很小的，因此要考虑优化这两个因素还应该主要从压缩后的图像质量上来考虑，不过对图片进行 Shuffle 操作这一实验范式得出了不少有趣的结果，通过细致的分析，这些实验结果极大的加深了我们对于 JPEG 压缩原理的理解。

还有一点细节在于，前面我们讨论 DC 分量的编码效率受 Cut Size 影响的时候，粗略地采取了“波动频率和幅度”对 DC 分量的数值进行描述，定性地认为波动频率和幅度越大压缩效率也就越低。这只是从 DPCM 编码的原理上做出的大体考虑，但实际上 DPCM 的编码效率最终还是取决于 DC 分量的具体排列，这也是分化现象出现的根本原因。

总结看来，可以发现在 JPEG 压缩的原理当中存在大量的变化因素，它们不仅会影响我们关注的一些因变量，而且这些变化因素之间又存在相互影响，从而会引起不少有趣的实验现象的发生。我们在本文中所做出的探究只是阐明了其中一小部分的原理，还存在着许多现象尚待解释。并且由于在我们关注的因变量和大量的自变量之间往往不存在确定的函数关系，例如编码效率不仅仅取决于 RC 分量和 AC 分量的值及其分布，更取决于这些值的排列情况，而排列情况又是难以给出确切的数学描述的，因此我们还需要更严谨和系统的研究方法才能对许多实验现象给出确凿的阐释。

## PART TWO 信道编解码

在本部分中，我们以三种方式进行信道编解码，分别是 FEC 的汉明码、ARQ 的 CRC 码、以及 FEC 的卷积码。对于 CRC 码和卷积码，编解码方式并不唯一。例如 CRC 码可以采用不同的生成多项式和不同长度的信息多项式，而卷积码的编解码方式则更加丰富，(N, K, L)卷积码中的三个关键参数都能调整，转移函数矩阵并不唯一，其解码方式也各不相同。由于时间有限，我们在本文中对于 CRC 码和卷积码仅选取一种实现方式进行简单的性能测试。而对于相对简单的汉明码，我们较为深入地研究信道噪声对误码率的影响。

### CRC 码

首先来看 CRC 码。对于生成多项式，我们采用 CRC-ITU-T 标准，即

$$g(x) = x^{16} + x^{12} + x^5 + 1 \text{ OR } 10001000000100001$$

对于信息多项式的长度我们选取为 22，即构成了(38, 22)的线性分组码。信道传输采取 ARQ 模式，即如果收信端检测到收码有误，则反馈给发信端重新发送该组信息。对于信道噪声，我们定义 Error Rate (记为 Pe) 为任一码字发生差错的概率 (即 BSC 信道)。对于最终收到的码字，我们计算其与真实信息的汉明距离，并进行归一化，实际得到的便是收码的差错概率，记为 Pr。同时我们记录实际发送信息组的次数 (Send Time, ST)。显然，Pe 越大则会导致 ST 越大，同时由于某些差错可使得一码字变为另一码字而逃避检错，从而导致 Pr，那么 Pe 越大也会导致 Pr 越大。我们选取  $Pe = 0.00 : 0.01 : 0.18$  进行信道模拟，所得结果如 Figure 15 所示。

明显的规律是 ST 和 Pr 都随着 Pe 的增大呈现指数形式的上升，尤其是 ST 的增长幅度巨大，同时也导致信息传输所需时间迅速增长，可以想见当 Pe 接近于 1 时 ST 将趋近于无穷大。Pr 的增长规律也类似于指数函数，但实际上 Pr 是不可能超过 1 的，并且在我们所研究的 Pe 范围中 Pr 均维持在一个相对较小的范围内。由于信息传输所需时间将会超过人类所能承受的极限，我们在本文中没有研究 Pr 在 Pe 较大时的行为。

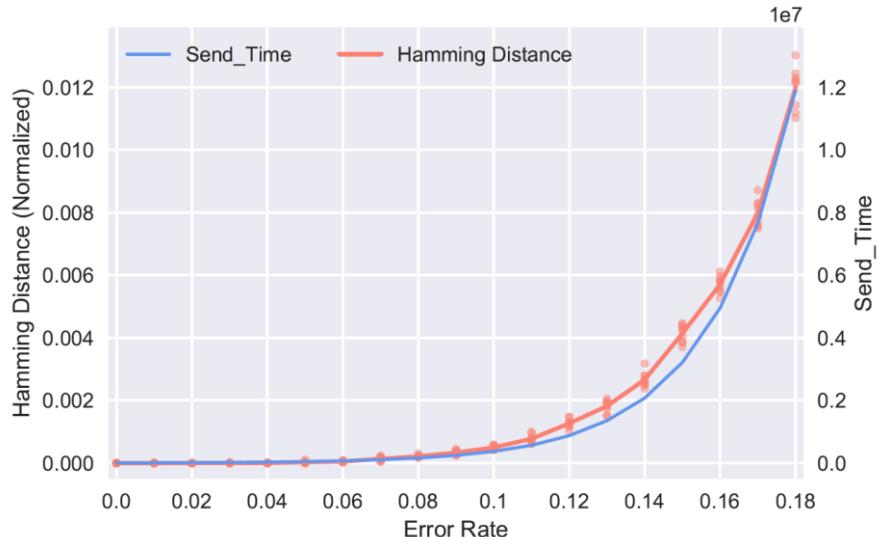


Figure 15 | Error Rate 对 CRC 码的影响。ST (蓝) 随着 Error Rate 的增加呈指数形式的增长。原始码字组数为 6880 组，即  $ST_0 = 6880$ ，在 Pe 增长到 0.18 时 ST 数便增长了超过 1000 倍。而 Pr (橙) 虽然也随 Error Rate 的增加呈类似指数形式的增长，但始终维持在较小的范围内。图示数据展示 9 组独立实验的结果。

## 卷积码

在本文中我们以(3,1,2)卷积码为例，转移函数矩阵选取为

$$G(D) = [1, 1 + D, 1 + D + D^2]$$

我们分别选取时延  $D = 3, 9, 27$ ，研究 Error Rate ( $Pe = 0.00 : 0.01 : 1.00$ )对实际误码率的影响，所得结果如 Figure 16 所示。

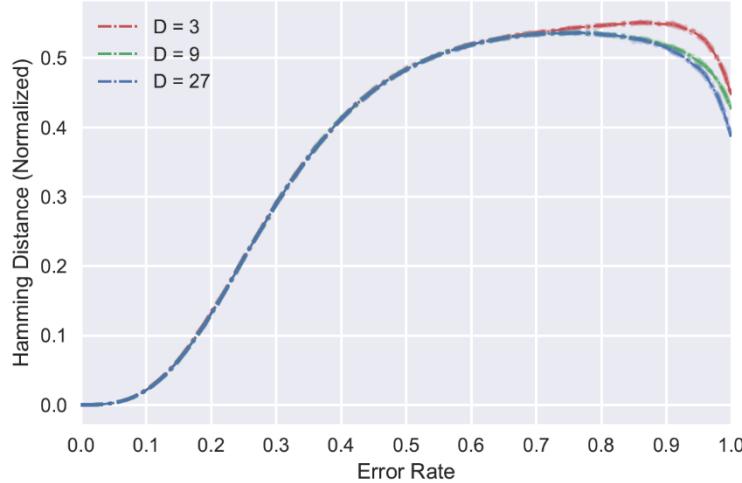


Figure 16 | Error Rate 对卷积码的影响。当 Error Rate 在较小的范围内增长时，误码率  $Pr$  近似于指数形式地上升，而当 Error Rate 较大时， $Pr$  的增长趋于平缓，当 Error Rate 接近于 1 时  $Pr$  反而开始下降。对于不同的时延  $D$ , Error Rate 较小时的  $Pr$  值几乎相同，对于接近于 1 的 Error Rate，增大时延可以减小  $Pr$ . 图示数据展示 9 组独立实验的结果。

可以看出，卷积码的误码率  $Pr$  一般随着 Error Rate 的上升而上升，但有趣的是，当 Error Rate 接近于 1 时， $Pr$  会出现反常的下降，其最大值始终维持在 0.5 和 0.6 之间。同时，在  $Pr$  的上升阶段，时延  $D$  对于  $Pr$  几乎没有影响，但时延增大却能使得下降阶段的  $Pr$  值减小。

由于时间关系，对于这一现象的成因我们并没有深入研究，不过首先要注意的是我们在本文中采用的卷积码仅仅是一个简单的示例，并不能揭示该现象是由信号本身引起还是卷积码的固有属性所导致，或者和卷积码的参数设置有关。进一步的结论需要通过对更一般的卷积码进行研究才能得出，因此我们现在还不能对卷积码的性质做出肯定性地理解。例如当时延选取为 2 时（最小极限），误码率在 Error Rate 接近于 1 时的下降幅度甚至了超过以上三者 [Figure 17]，与 Figure 16 中的规律恰好相反。

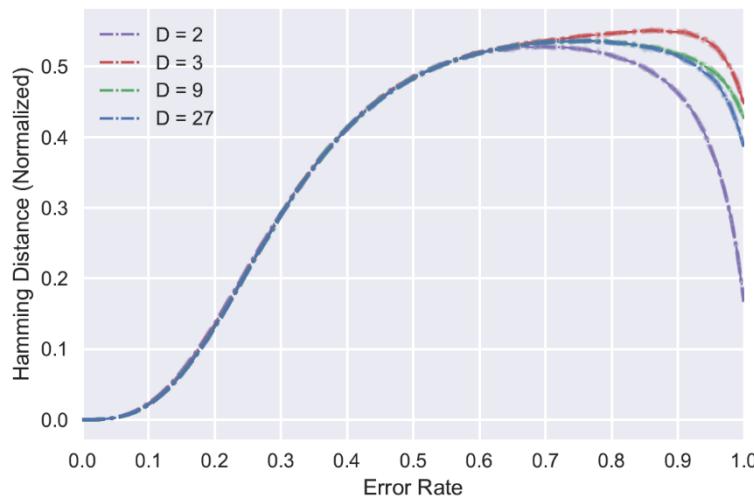


Figure 17 | 时延为 2 时引起的误码率下降超过更大的时延。

## 汉明码

最后我们来重点关注汉明码。我们知道汉明码的纠错能力为 1，也就是说当所收到的码字中只包含一个差错或者没有差错的情况下，汉明码能自动纠正错误得到正确的码字。如果所收到的码字中含有超过一个差错，前向纠错的汉明码可能发生检错遗漏，而如果检测到差错，则会将差错认定为只有一个进行纠正，即可能产生错误的纠错。

我们对于汉明码在差错概率 (Error Rate,  $Pe$ ) 超过其纠错能力的情况下行为进行了探究。同样采用  $Pr$  为指标，我们选取  $Pe = 0.00 : 0.01 : 1.00$ ，以(7,4), (15,11), (31,26) 三种汉明码为研究对象，所得结果如 Figure 18 所示。

该实验显示出了非常有趣的结果。对于三种不同的汉明码， $Pr$  均呈现有规律的波动 [Figure 18 左]。当  $Pe$  较小时， $Pr$  首先低于  $Pe$ ，这是显然的，因为解码对信道噪声进行了纠错，从而降低了误码率。然而随着  $Pe$  的逐渐增大， $Pr$  将从某一点开始高于  $Pe$ ，然后在  $Pe = 0.5$  左右再次低于  $Pe$ ，随后又重新超过  $Pe$ 。

将  $Pd \triangleq Pr - Pe$  对  $Pe$  作图 [Figure 18 右] 能更明显地看出这一规律。三种不同的汉明码分别具有自己的特征，但  $Pd$  的波形都可以大致分为四个区间 [0-a-0.5-b-1]，区间端点分别对应四个  $Pd$  为 0 的点，并且具有四个峰值 [m, s, t, n]。当  $Pe = 0$  时  $Pr$  全都 0，而  $Pe = 1$  时  $Pr$  全都为 1， $Pe = 0.5$  时  $Pr$  并不为 0.5，但都与 0.5 非常接近。 $Pd$  的波形关于(0.5,0)则呈现很规则的对称。

随着码长的增大，a 逐渐靠近于 0，b 逐渐靠近于 1，峰值点也同样逐渐向两边移动。对于(7,4)码，m, n 峰值绝对值大小大于 s, t 峰值，而对于(15,11)和(31,26)码则相反。m,n 峰值随着码长增大逐渐减小，而 s,t 峰值除了(7,4)码也随着码长增大逐渐减小。

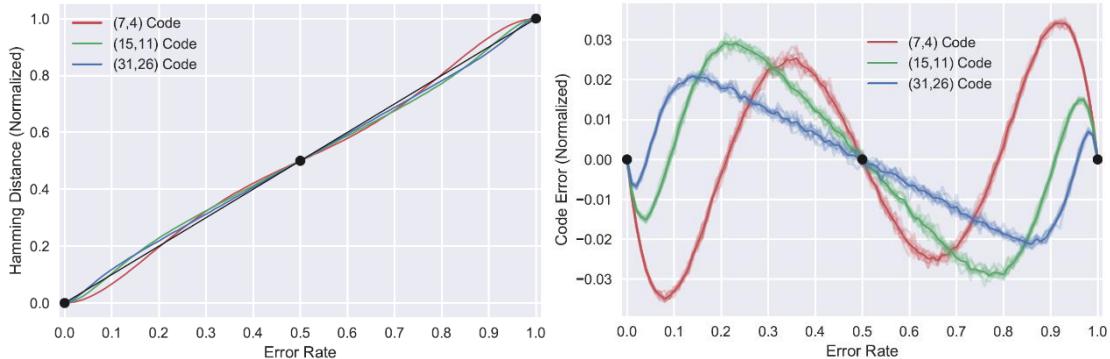


Figure 18 | Error Rate 对汉明码的影响。左图： $Pr$  关于  $Pe$  的变化规律，三种不同的汉明码分别用不同的颜色表示，黑色实心点分别表示  $(0,0)$ ,  $(0.5,0.5)$ ,  $(1,1)$  三个坐标点。三种汉明码的  $Pr$  均相对于  $Pe$  呈现有规律的波动。右图：为了便于观察，将  $(Pr - Pe)$  关于  $Pe$  作图，可见三种汉明码的  $Pr - Pe$  关于 0 呈现有规律的波动，并且波形特点与码长相关 [见正文]。图示数据来自 9 次独立实验，不透明实线表示均值，半透明实线表示单次实验结果。

上述的许多实验现象即使在直观上也并不难以解释。

首先  $Pr$  会小于  $Pe$  也会大于  $Pe$ ，这是由汉明码的编解码原理所决定的。对于任一差错，汉明码总会找到其对应的伴随式，然后将其看成一次差错进行纠错。对于差错数大于 1 的差错图样，汉明码所认定的差错位置可能与真实的差错位置不同，这便导致错误纠错，

使得差错数在信道噪声的基础上再增加 1，如果‘歪打正着’，那么发生正确纠错，使得差错数减少 1，这便导致了  $P_r$  既有可能小于  $P_e$  也有可能大于  $P_e$ 。

我们可以把  $[0,a]$  看作汉明码的有效纠错区间，因为在此区间内  $P_d$  小于 0。我们发现随着码字长度的增长，有效纠错区间逐渐变窄，这也是显然的。因为在 BSC 的信道模型下，码字越长，一组码字中差错数的期望值就越高。

在  $P_e = 1$  时， $P_r$  总是为 1，说明在码字全部出错的情况下汉明码不具备纠错能力，这也是由汉明码本身的结构直接导致的，因为当差错图样全为 1 时，其对应的伴随式（也就是校验矩阵所有列向量的模二和）一定全为 0，即汉明码认为此时码字中并无差错（如果认为有差错的话，那么对任一位置的码字进行纠错都会使得误码率降低）。

在  $P_e = 0.5$  时， $P_r$  也近似为 0.5，对此我们也可以给出非常简单的解释。从统计的角度来看，此时的差错图样中有一半为 1，如果我们假设伴随式与差错图样独立的话，那么根据随机选取的伴随式进行纠错，纠错成功（差错数减 1）和纠错失败（差错数加 1）的概率则是相等的，这就导致总体上的  $P_r$  与  $P_e$  相等。但我们知道伴随式是由差错图样所确定的，因此实际上我们可以认为在所有一半元素为 1 的差错图样中，诱导汉明码进行错误纠错和正确纠错的差错图样数目大致相同，而在  $P_e = 0.5$  时这一差错图样的集合占据了所有差错图样的绝大多数。

从上面的几点初步分析可以看出，Figure 18 中的实验现象一方面反映了汉明码本身的结构，另一方面则反映了 BSC 信道模型的统计特征。由于汉明码完全基于确定的矩阵运算，而 BSC 信道也仅仅基于简单的伯努利分布，我们断定可以通过结合近世代数和随机分析的方法对 Figure 18 中的实验现象做出完美的理论解释，乃至给出  $P_d$  的期望值关于  $P_e$  的函数的近似表达式，并且这一工作的难度并不会很高（和对本文第一部分中 R 值和 Cut Size 的关系给出理论解释这一工作相比应该完全不在一个等级）。

不过在进一步研究之前有一点需要说明，Figure 18 中的实验数据均来源于 Lena.tif，但实际上对于我们采用的 4 张图片所得实验结果是完全相同的 [Data Not Shown]，因此我们可以认定该实验现象是由汉明码的结构和 BSC 信道的特性所引起。其实这在理论上也是显然的，因为汉明码并不关心信号码字，而只关心差错图样，因此传输的信息并不会影响我们的实验结果。

分离变量法是本文第一部分普遍采用的研究方法，在这里我们采用同样的方法首先研究汉明码本身的特性，那么便应该排除 BSC 信道特征的影响。我们知道，当设置 Error Rate 为某一个值时，实际上每一组码字所含的差错数仍然是随机的，只是差错数的期望占总码字数的比例等于 Error Rate。为此我们消除 BSC 信道的随机性，但仍然保留差错的出现，具体方法即将 Error Rate 更改为 Error Number，使得每组码字中出现的差错个数固定，但差错出现的位置仍然随机。以 Lena.tif 和(7,4)汉明码为例，所得结果如 Figure 19 所示。

由于差错位置仍然随机，并且码字数目很大 (37846)，可以认为对于每个 Error Number，所得数据反映了汉明码在该 Error Number 下的固有特征。我们首先观察到当 Error Number = 1 时， $P_d$  总是为  $-1/7$ ，这代表着汉明码对于 1 位错误的完全纠错。而需要注意的是，当 Error Number = 6 时， $P_d$  总是为  $1/7$ ，这代表着此时总是进行错误纠正，使得差错数增加 1。前面我们已经提到，Error Rate 为 1（即 Error Number 为 7 时

汉明码无法检出错误，原因在于校验矩阵所有列向量的模二和为 0. 同样，这里的错误纠正同样源于汉明码本身的结构，而与 BSC 信道无关。

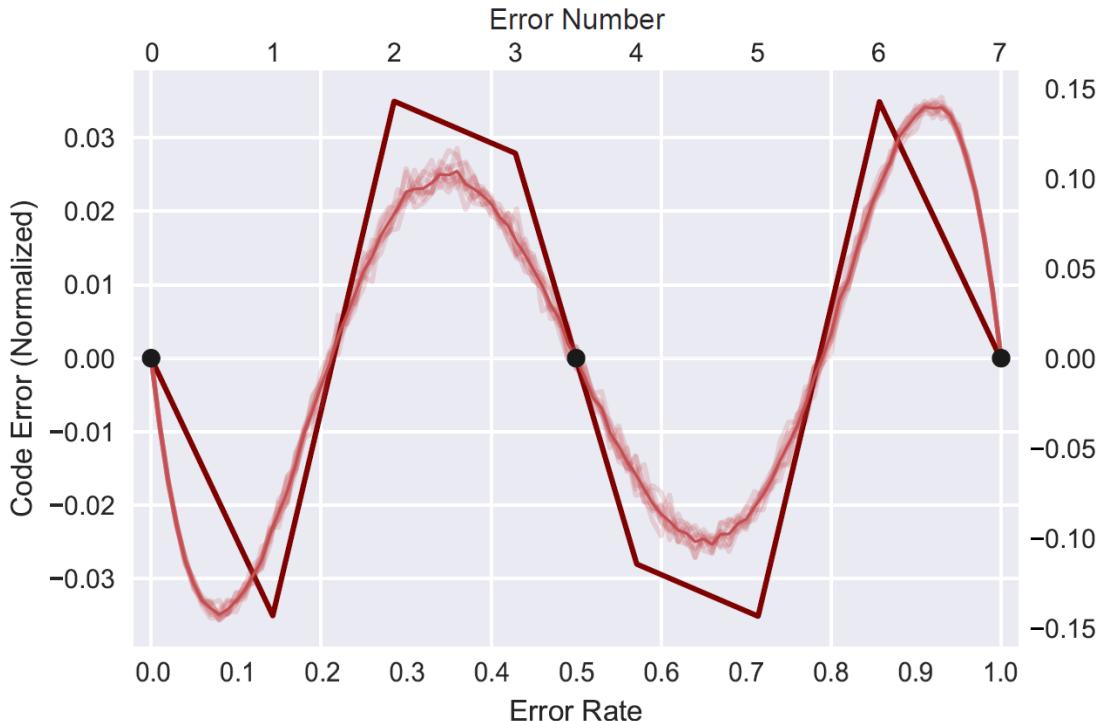


Figure 19 | Error Rate 和 Error Number 对汉明码的影响。棕色：Error Number 对汉明码的影响，数据显示为来自 9 次独立实验的平均值（9 次实验结果几乎完全重合），坐标值位于右侧和顶部；红色：Error Rate 对汉明码的影响，数据显示所有 9 次独立实验数据，不透明实线表示均值，坐标轴位于左侧和底部。

具体说来，在  $\text{Error Number} = 1$  时差错图样中只有一位为 0，我们不妨称为第  $k$  位，则其所得到的伴随式则是校验矩阵除开第  $k$  列的所有列的模二和，这等价于校验矩阵所有列的模二和再与第  $k$  列做模二和，而我们又已经知道校验矩阵所有列向量的模二和为 0，因此我们得到的伴随式就是校验矩阵的第  $k$  列，那么根据汉明码的原理，此时汉明码认为发生差错的位置就是第  $k$  位，正好是实际上没有发生差错的位置，最终导致了差错数增加 1.

根据同样的原理，我们可以对所有 Error Number 对应的现象进行分析。为方便起见，我们列出(7,4)汉明码的校验矩阵如下

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

当  $\text{Error Number} = 2$  时， $P_d$  也几乎为  $1/7$ ，只是并不完全等于  $1/7$ 。此时伴随式为校验矩阵中任意两列的模二和，而直接观察校验矩阵我们便可得知，任意两列之和不会等于其中任一列（因为校验矩阵不含零向量），同时也不会等于零向量（因为校验矩阵每一列都不相同），这就导致了当差错数为 2 时，汉明码总是会在错误的位置进行纠错，因此实际差错数总是为 3. 至于此时的  $P_d$  并不完全等于  $1/7$ ，是因为  $P_r$  的计算只基于信息位，而差错数是包含校验位的，而当纠错位落在校验位时不会对信息位造成影响，这种情况要求差错位分别发生在前三位中的 1 位以及第 4 位。当  $\text{Error Number} = 3$  时

伴随式为校验矩阵中任意三列的模二和，此时是存在三列之和等于其中某一列的情况的，即会引起正确纠错，因此总体来说此时的  $P_d$  会比差错数为 2 时要小。 $P_d$  的对称性也是汉明码的必然结果，因为校验矩阵所有列之和为零向量，那么任意  $x$  列之和必然等于  $n-x$  列之和，也就是说纠错位相同，但由于误码位正好相反，因此正确纠错与错误纠错出现的概率也正好相反，最终导致  $P_d$  关于(0.5,0)呈中心对称。

显然，上述分析可以应用于任意码长的汉明码，不同汉明码之间的变化规律也可以直接导出。那么现在我们重新对采用 Error Rate 的实验结果进行分析，从 Figure 19 可以看出， $P_d$  在 Error Rate 的情况下和 Error Number 的情况下所得实验结果并不相符，一方面是曲线峰值的位置并非重合，另一方面曲线的绝对大小具有很大的差异，在 Error Rate 的情况下  $P_d$  的变化幅度远小于 Error Number 的情况。

我们同样先进行直观的分析。当 Error Rate 取某个值时，如果我们考察一组码字中实际出现的差错数  $T$ ，那么  $T$  应该是一个随机变量，它的期望等于 Error Rate 与码长的乘积。如果  $T$  完全等于其期望值，那么在 Error Rate 和 Error Number 两种情况下应该得到相同的结果。而实际情况中， $T$  值会向两侧偏离，也就出现了实际的 Error Number 不等于期望值的情况，由于  $P_r$  是所有码字的平均效应所得，那么在理论上  $P_r$  应该等于可能出现的 Error Number 所对应的  $P_r$  的加权平均（对于  $P_d$  同样成立，因为  $P_r$  和  $P_d$  只相差一个常数  $P_e$ ）。

例如当 Error Rate = 0.3 左右时， $T$  的期望值为  $7 \times 0.3 = 2.1$ （依然以(7,4)汉明码为例），我们已经知道 Error Rate 为 2 时对应的  $P_d$  约为  $1/7$ ，而此时实际的  $P_d$  仅为 0.025 左右，这是因为实际的  $T$  值也可能为 0、1、3、4 等，它们的平均效应才导致了  $P_d$  约为 0.025。

下面我们便对 Error Rate 的情况进行理论分析。对于任意(n,k)汉明码，和任意(0,1)之间的 Error Rate (即  $P_e$ ，以下简记为  $p$ )，一组码字中出现的差错数  $T$  符合二项分布

$$P(T = t) = \binom{n}{t} * p^t * (1-p)^{n-t}$$

而不同的差错数对应不同的  $P_d$ ，记为  $P_d(t)$ ，于是对任意  $p$ ，其  $P_d$  值在理论上为

$$P_d(p) = \sum_{t=0}^n \binom{n}{t} * p^t * (1-p)^{n-t} * P_d(t)$$

对于  $P_d(t)$ ，我们可以通过对校验矩阵直接进行分析得出  $P_d(t)$  的准确值。例如  $t = 1$  时  $P_d(1) = 0$ ，而  $P_d(n-1) = 1/n$ 。不过对于一般的  $t$ ， $P_d(t)$  值的计算较为复杂，因为需要计算所有可能的差错图样所导致的  $P_d$  值。但其实在 Error Number 的情况下，我们已经得到了  $P_d(t)$  的估计值，它们是对于真实值非常准确的估计。

现在我们便能根据  $p$  值直接计算  $P_d(p)$  了。同样对于上文所用的三种汉明码，通过理论计算得到的  $P_d(p)$  与实验所得值如 Figure 20 所示，可见以上理论分析的结果完美地解释了实验数据。

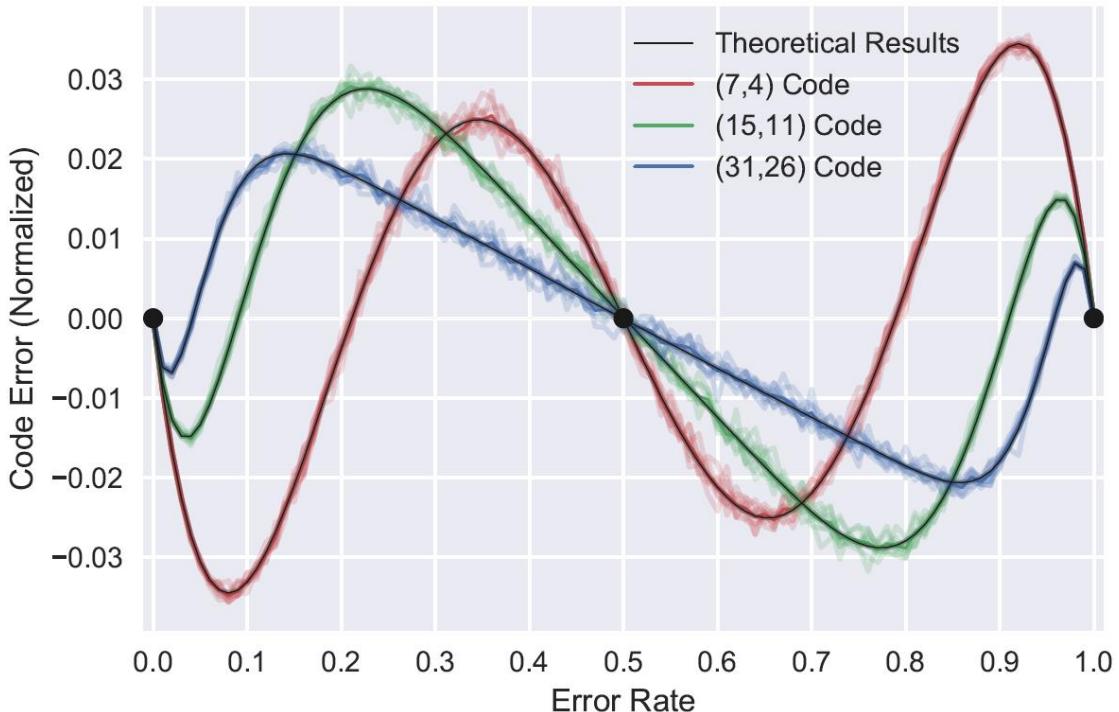


Figure 20 | 随机分析的理论结果与实验数据相符。理论结果所得值用黑色实线表示。

## References

[1]. Digital Compression and Coding of Continuous-tone Still Images. ISO/IEC International Standard 10918-1. 1991.

[2]. G. K., Wallace. The JPEG Still Picture Compression Standard. IEEE Transactions on Consumer Electronics. 1992 (revised). Vol. 38 (1): xviii-xxxiv.

## Resources

1. JPEG [Wikipedia](#) [Link](#)
2. YCbCr [Wikipedia](#) [Link](#)
3. 基于 DCT 变换的 JPEG 图像压缩原理. [CSDN 博客](#) [Link](#)
4. JPEG 系列三 JPEG 图像压缩. [CSDN 博客](#) [Link](#)
5. [数据压缩] JPEG 标准与原理解析. [CSDN 博客](#) [Link](#)
6. 什么是 4:4:4、4:2:2、4:2:0？了解图像压缩取样的方式. [CSDN 博客](#) [Link](#)

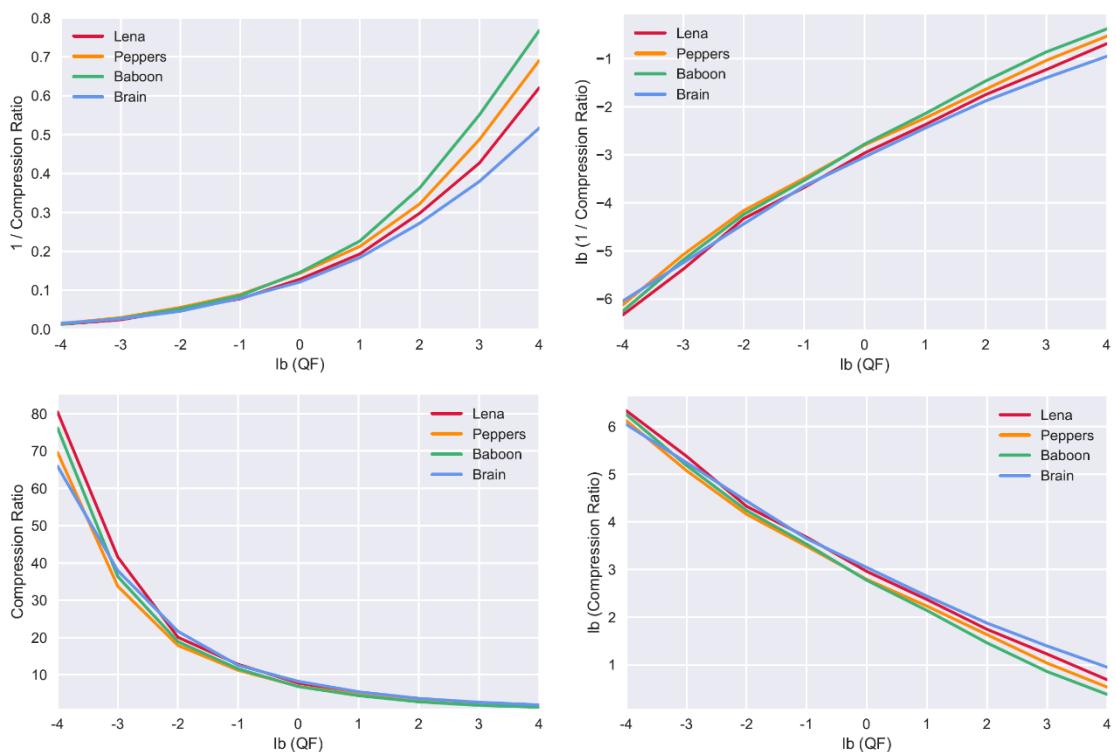
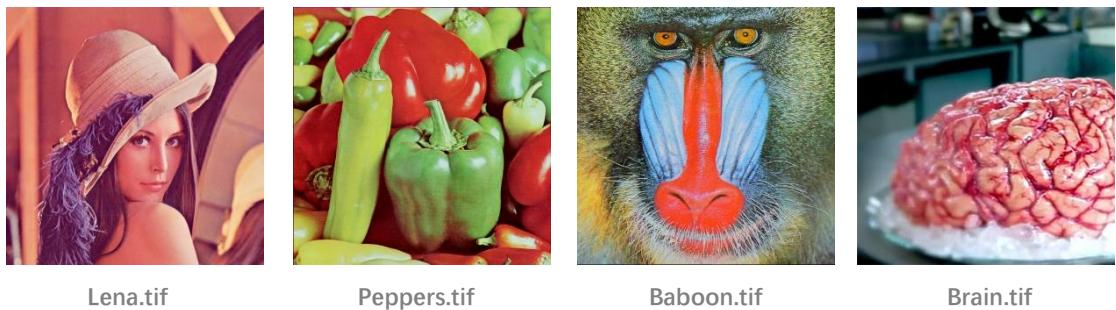
## Supplementary Materials

本文的 Supplementary Materials 提供了 JACS 的完整代码脚本，其中包含对 JPEG 压缩原理的讲解，使用方法见本文开头说明三。代码中所调用的外部函数均包含在附件中。

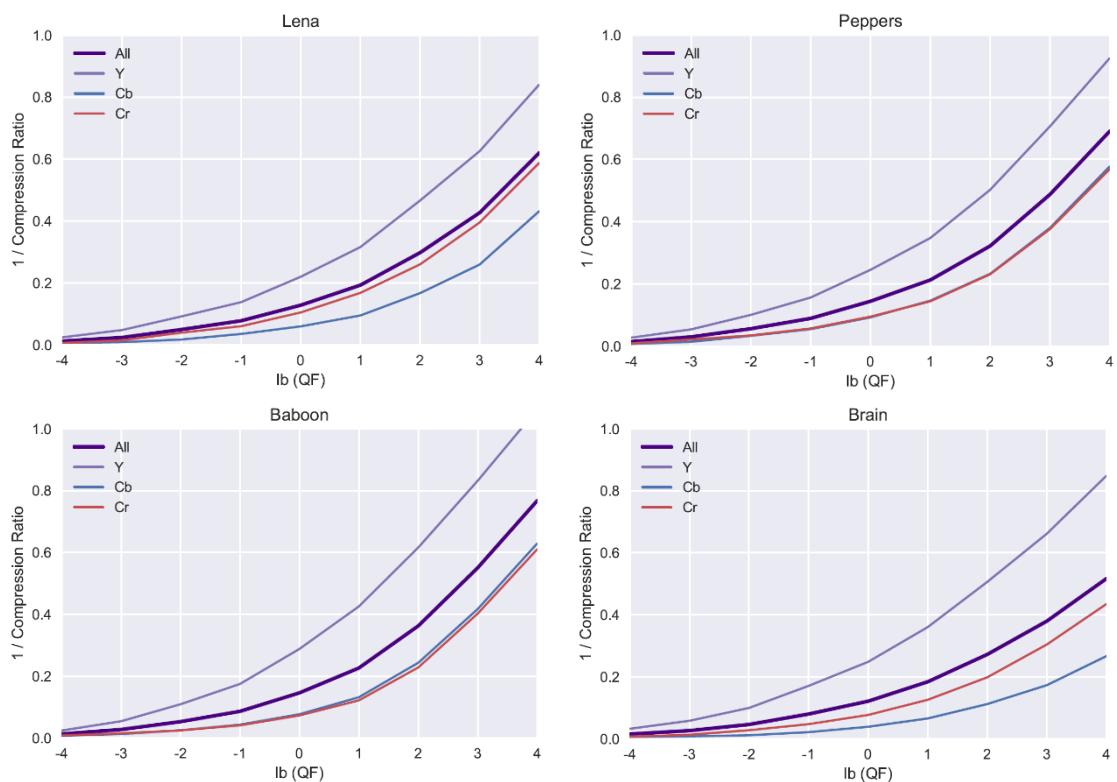
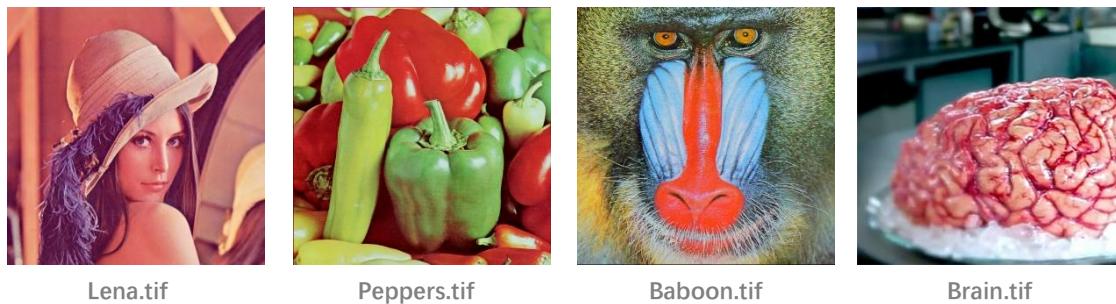
本文中的超链接可通过电子版本访问。



## Supplementary Figure 1



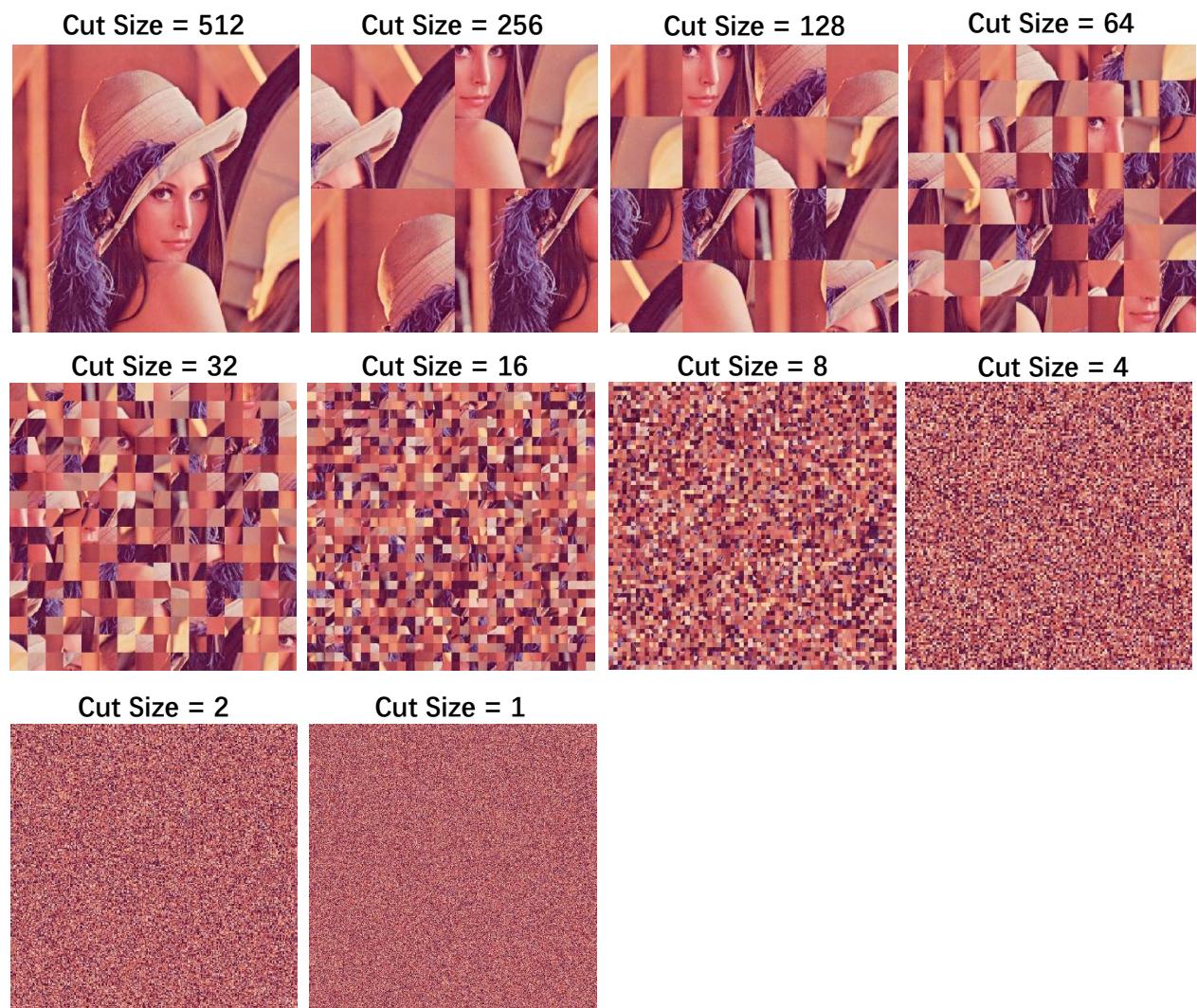
## Supplementary Figure 2



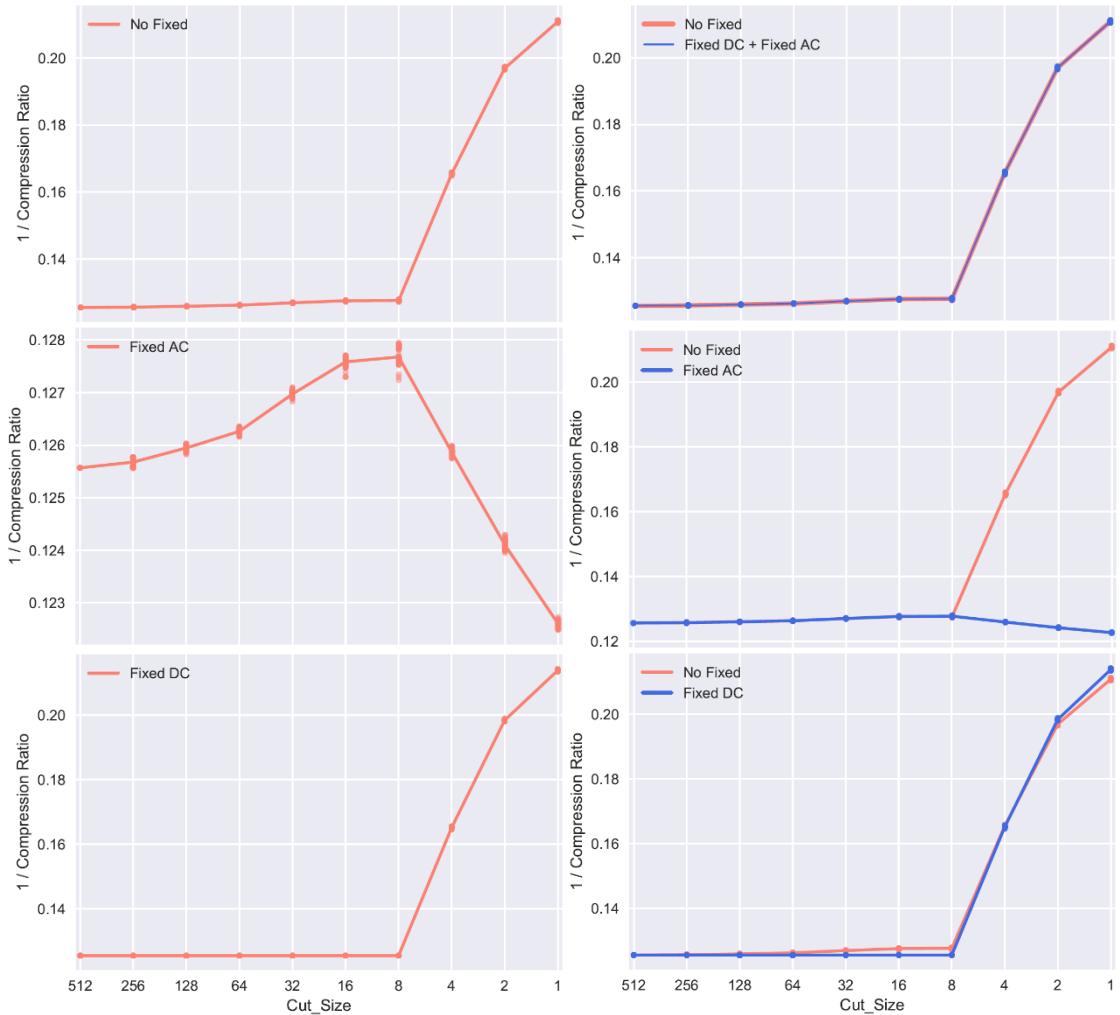
## Supplementary Figure 3



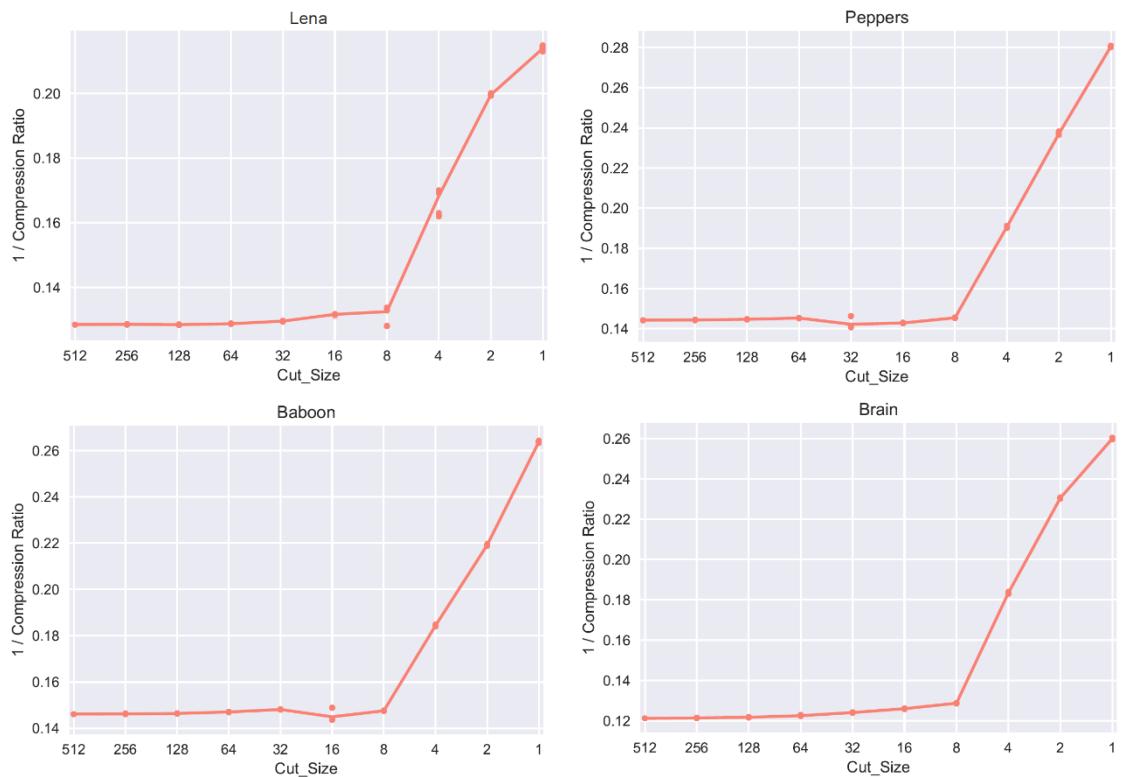
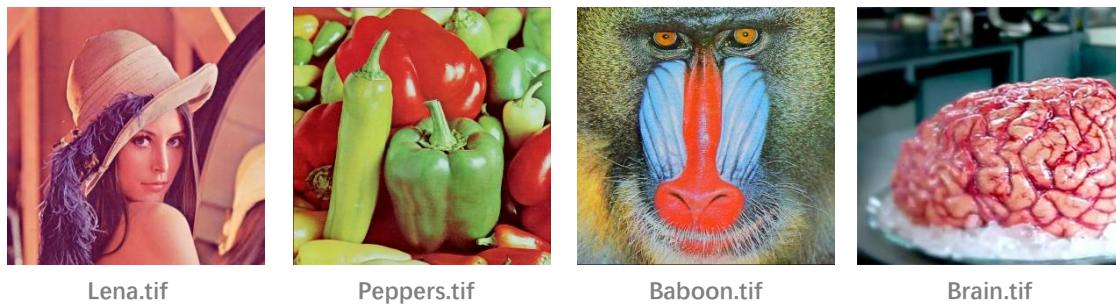
## Supplementary Figure 4



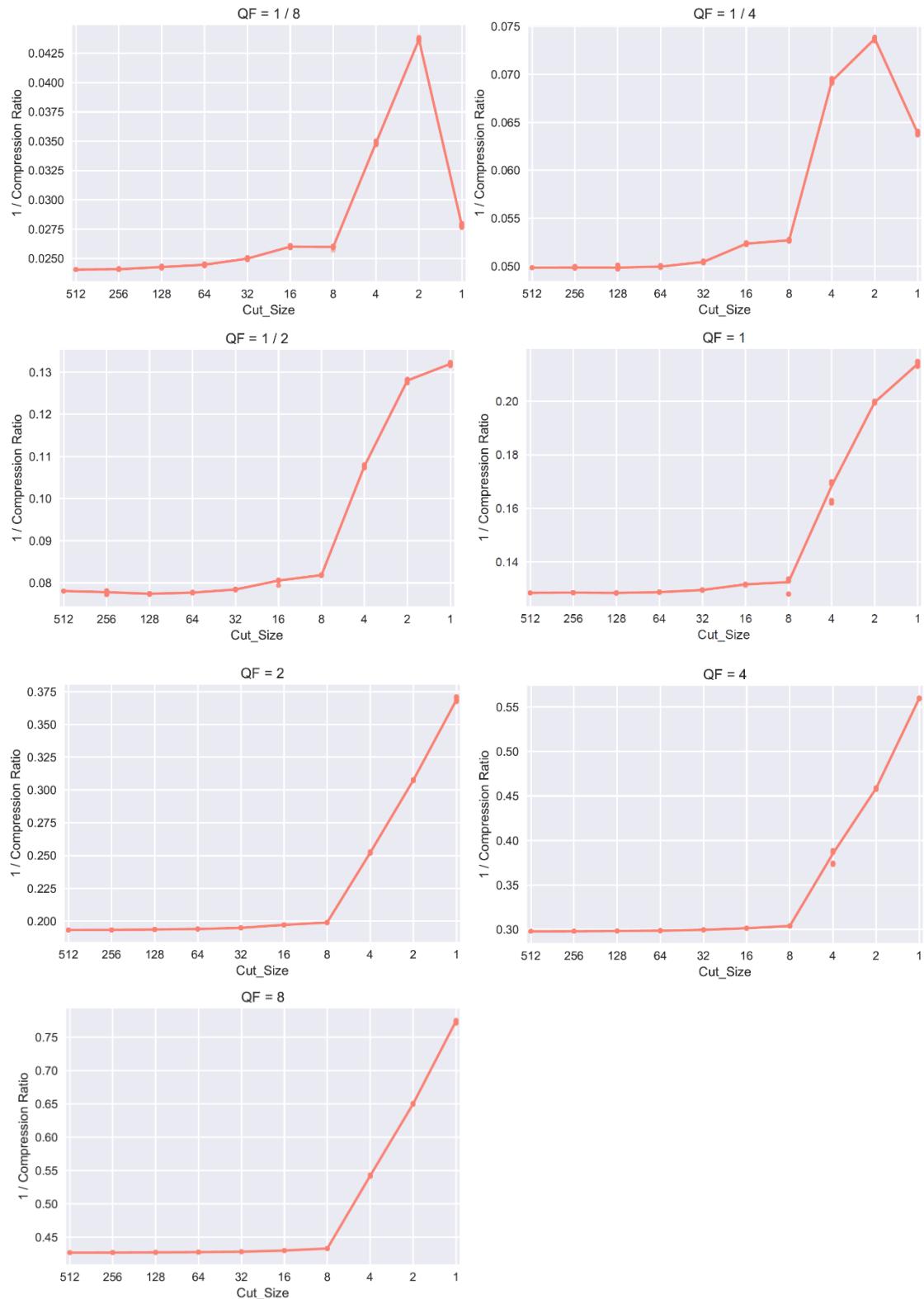
## Supplementary Figure 5



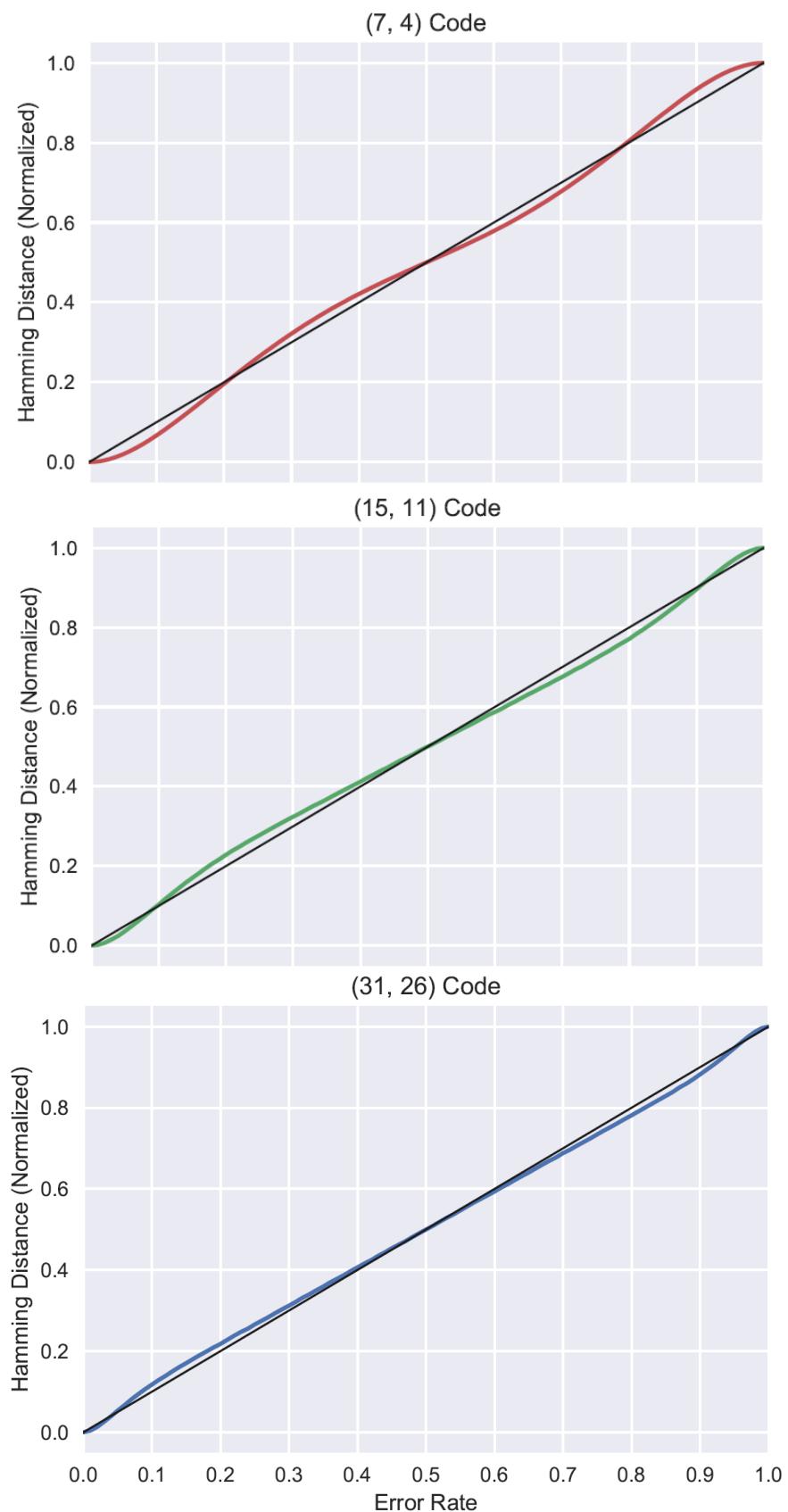
## Supplementary Figure 6



## Supplementary Figure 7



## Supplementary Figure 8



# JPEG: A COMMUNICATION SYSTEM

JACS

## Supplementary Materials

Copyright 2017-2018 Yiheng Wang Penicillin.

王一恒 15307130257

2017/12 - 2018/01

## JPEG ENCODING

### Procedure Preview:

```
oriimage (uint8) —— image (uint8) —— imageYUV (double)
—— imageDCT (double) —— imageQ (double) —— messege
```

### Step Zero: Read the Image

```
image_File_Name = 'Lena.tif';
oriimage = imread(image_File_Name);
Size = size(oriimage);
height = Size(1); % height variable is used for later algorithm
width = Size(2); % height variable is used for later algorithm
oriheight = height; % Store the original height for image recovery
oriwidth = width; % Store the original weight for image recovery
```

### Step One: Sampling

```
% The image may not be exactly divided into 8*8 blocks,
% so we need to extend its edges in such situation.
if mod(height,8) || mod(width,8)
    % If the image cannot be divided perfectly
    extendh = mod(8-mod(height,8),8);
    extendw = mod(8-mod(width,8),8);
    image = zeros(height+extendh,width+extendw,3);
    % Extend the right and bottom edge of the original image
    % The extended pixels are set to be black
    image(1:height,1:width,:) = oriimage;
    image = double(image);
    % Use image for later algorithm, and convert it to double
    Size = size(image);
    height = Size(1);
    width = Size(2);
    NY = height/8; % Record the block number in the y axis
    NX = width/8; % Record the block number in the x axis
else
    % If the image can be divided perfectly
    image = double(oriimage);
```

```

NY = height/8;
NX = width/8;
end

```

## Step Two: Convert RGB to YUV

```

imageYUV = zeros(height,width,3);

% RGB to YUV

K_R = 0.299;
K_G = 0.587;
K_B = 0.114;

% Y = K_R.*R + K_G.*G + K_B.*B;
% U = 1/2.* (B-Y)./(1-K_B);
% V = 1/2.* (R-Y)./(1-K_R);

% Y = 0.299*R + 0.587*G + 0.114*B
% U(Cb) = -0.1687*R - 0.3313*G + 0.5*B
% V(Cr) = 0.5*R - 0.4187*G - 0.0813*B

R = image(:,:,1) - 128;
G = image(:,:,2) - 128;
B = image(:,:,3) - 128;

Y = K_R.*R + K_G.*G + K_B.*B;
U = 1/2.* (B-Y)./(1-K_B);
V = 1/2.* (R-Y)./(1-K_R);

imageYUV(:,:,:) = cat(3,Y,U,V);

```

## Step Three: DCT

The principle of DCT is briefly summarized below:

$$F(u,v) = \frac{1}{4} C(u) C(v) \sum_{x=0}^7 \sum_{y=0}^7 f(x,y) \cos\left[\frac{\pi(2x+1)u}{16}\right] \cos\left[\frac{\pi(2y+1)v}{16}\right]$$

for  $u = 0, \dots, 7$  and  $v = 0, \dots, 7$

$$\text{where } C(k) = \begin{cases} 1/\sqrt{2} & \text{for } k = 0 \\ 1 & \text{otherwise} \end{cases}$$

$$b[x,y] = \cos\left[\frac{(2x+1)u\pi}{16}\right] \cos\left[\frac{(2y+1)v\pi}{16}\right]$$

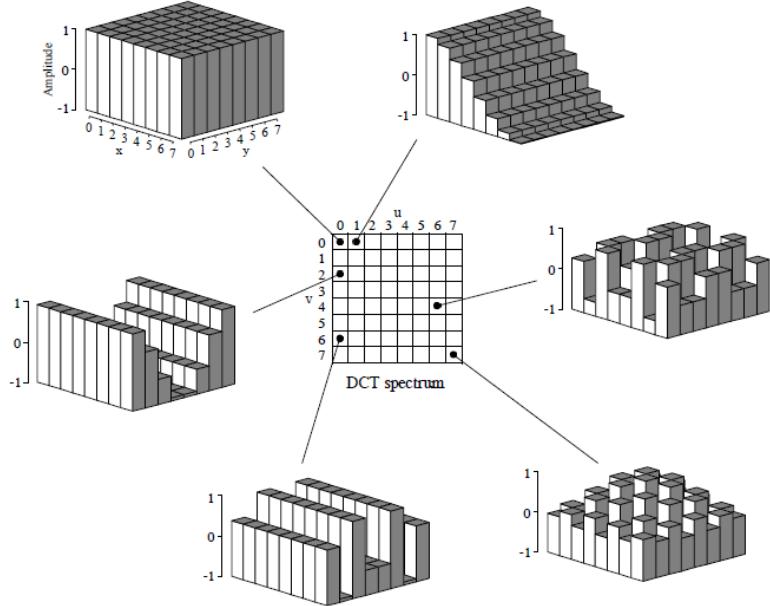


FIGURE 27-10  
The DCT basis functions. The DCT spectrum consists of an  $8 \times 8$  array, with each element in the array being an amplitude of one of the 64 basis functions. Six of these basis functions are shown here, referenced to where the corresponding amplitude resides.

$$\text{FDCT: } F = C^T * f * C$$

$$C = \sqrt{\frac{2}{N}} * \begin{pmatrix} \sqrt{1/2} & \sqrt{1/2} & \dots & \sqrt{1/2} \\ \cos(\frac{1}{2N}\pi) & \cos(\frac{3}{2N}\pi) & \dots & \cos(\frac{2N-1}{2N}\pi) \\ \dots & \dots & \dots & \dots \\ \cos(\frac{N-1}{2N}\pi) & \cos(\frac{3*(N-1)}{2N}\pi) & \dots & \cos(\frac{(2N-1)*(N-1)}{2N}\pi) \end{pmatrix}$$

$$\text{IDCT: } f = C * F * C^T$$

```

N = 8;      % Block size
temp = (0:N-1)'*(1:2:2*N-1).*pi./(2*N);
temp(1,:) = pi/4.*ones(1,N);
C = sqrt(2/N).*cos(temp);
% Create the base matrix for DCT

imageDCT = zeros(height,width,3);

% We accomplish this task here in a block-by-block fashion.
for yindex = 1:NY
    for xindex = 1:NX
        DCT_Y = C'*imageYUV(8*(yindex-1)+1:8*(yindex-1)+ ...
            8,8*(xindex-1)+1:8*(xindex-1)+8,1)*C;
        DCT_U = C'*imageYUV(8*(yindex-1)+1:8*(yindex-1)+ ...
            8,8*(xindex-1)+1:8*(xindex-1)+8,2)*C;
        DCT_V = C'*imageYUV(8*(yindex-1)+1:8*(yindex-1)+ ...
            8,8*(xindex-1)+1:8*(xindex-1)+8,3)*C;
        imageDCT(8*(yindex-1)+1:8*(yindex-1)+8,8*(xindex-1)+ ...
            1:8*(xindex-1)+8,:) = cat(3,DCT_Y,DCT_U,DCT_V);
    end
end

```

## Step Four: Quantization

The principle of quantization and the recommended quantization tables are shown below:

$$F_q(u,v) = \text{Round} \left( \frac{F(u,v)}{Q(u,v)} \right)$$

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

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

Y Quantization Table:

```
YQTable = [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 13 16 24 40 57 69 56; ...
    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];
```

UV Quantization Table:

```
UVQTable = [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];
```

Quantization:

```
% Create a full-size quantization table
YQT = ones(height,width);
UVQT = ones(height,width);
for yindex = 1:NY
    for xindex = 1:NX
        YQT(8*(yindex-1)+1:8*(yindex-1)+8, ...
            8*(xindex-1)+1:8*(xindex-1)+8) = YQTable;
```

```

UVQT(8*(yindex-1)+1:8*(yindex-1)+8, ...
      8*(xindex-1)+1:8*(xindex-1)+8) = UVQTable;
end
end

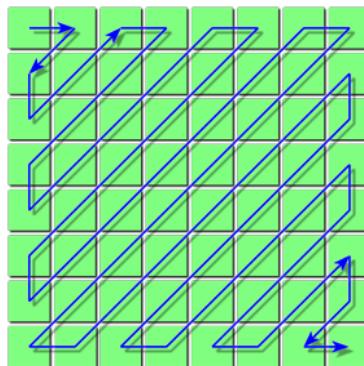
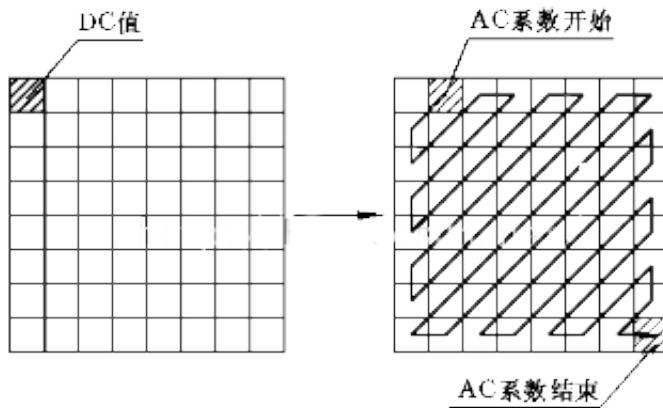
quality_factor = 1/8;
% Change the quality
% The bigger the quality_factor is, the higher the quality will be.
% The default quantization table roughly corresponds to 50% quality.
YQT = YQT./quality_factor;
UVQT = UVQT./quality_factor;

imageQ = cat(3,round(imageDCT(:,:,:1)./YQT), ...
            round(imageDCT(:,:,:2)./UVQT),round(imageDCT(:,:,:3)./UVQT)));

```

## Step Five: Z Scanning

Diagram:



```

Zig_Zag = [1 2 6 7 15 16 28 29; ...
           3 5 8 14 17 27 30 43; ...
           4 9 13 18 26 31 42 44; ...
           10 12 19 25 32 41 45 54; ...
           11 20 24 33 40 46 53 55; ...
           21 23 34 39 47 52 56 61; ...
           22 35 38 48 51 57 60 62; ...
           36 37 49 50 58 59 63 64]; ...
% Call function 'Zscan()'
% This function converts a two dimensional image
% into a matrix of size (NY*NX,64),
% with each row representing a 8*8 block

```

```

% and the first column representing the DC components.
Scan = cat(3,Zscan(imageQ(:,:,1),NY,NX,Zig_Zag), ...
           Zscan(imageQ(:,:,2),NY,NX,Zig_Zag),Zscan(imageQ(:,:,3),NY,NX,Zig_Zag));
% Divide the results into DC components and AC components
DC = Scan(:,:,1);           % This is a (NY*NX,1,3) matrix
DC = DC(:,:,1:3);          % Convert it into a (NY*NX,3) matrix
AC = Scan(:,:,2:64,:);      % This is a (NY*NX,63,3) matrix

```

# Encoding Example

Here we demonstrate how a quantized image is encoded for communication.

The DC coefficients and the AC coefficients are encoded differently.

## **Step Six: DPCM Coding for DC Coefficients**

$$\Delta = DC_K(0,0) - DC_{(K-1)}(0,0)$$

Represented by **(Size, Value)**

**Size:** Bit size of DC Delta value --- **Huffman Coding**

**Value:** DC Delta value --- **VLI Coding**

## For example:

$$D(K-1)(0,0) = 12$$

$$D_K(0,0) = 15$$

Thus the code for P K is (2,3)

### Step Seven: *RLE Coding* for AC Coefficients

Run Length Encoding: Represented by (Run/Size, Value)

Run: Number of continuous 0 --- Huffman Coding

**Size:** Bit size of non-zero value --- **Huffman Coding**

**Value:** The next non-zero value --- VI | Coding

**For example:-**

the code is

(0 35) (0 7) (3 -6) (0 -2) (2 -9) (15 0) (2 8) EOF

if size exceeds 16, then every 16 '0' is grouped and represented by (15, 0):

if all the left values are '0' then it is represented by 'EOF'

## Step Eight: VLI Coding for 'Values'

Value		Size	Bits	
	0	0		-
-1	1	1	0	1
-3,-2	2,3	2	00,01	10,11
-7,-6,-5,-4	4,5,6,7	3	000,001,010,011	100,101,110,111
-15,...,-8	8,...,15	4	0000,...,0111	1000,...,1111
-31,...,-16	16,...,31	5	0 0000,...,0 1111	1 0000,...,1 1111
-63,...,-32	32,...,63	6	00 0000,...	...,11 1111
-127,...,-64	64,...,127	7	000 0000,...	...,111 1111
-255,...,-128	128,...,255	8	0000 0000,...	...,1111 1111
-511,...,-256	256,...,511	9	0 0000 0000,...	...,1 1111 1111
-1023,...,-512	512,...,1023	10	00 0000 0000,...	...,11 1111 1111
-2047,...,-1024	1024,...,2047	11	000 0000 0000,...	...,111 1111 1111

(0, 35) (0, 7) (3, -6) (0, -2) (2, -9) (15, 0) (2, 8) EOB

(0, 6, 100011) (0, 3, 111) (3, 3, 001) (0, 2, 01) (2, 4, 0110) (15, -) (2, 4, 1000) EOB(0, 0)

The first two numbers lie in the range 0~15, thus can be combined into a single byte,

with the first four digits representing **Run** and the last four digits representing **Value**

(0x6,100011) (0x3,111) (0x33,001) (0x2,01) (0x24,0110) (0xF0,-) (0x24,1000) EOB(0, 0)

## Step Nine: Huffman Coding for 'Size' or 'Run/Size'

Huffman table for DC and AC are different, for Y and UV are different as well,  
so we need in total 4 Huffman tables

### For example:

For DC values, there is no '0' ahead, and 'Size' lies in the range 0~11

Length	Value	Bits
3 bits	04	000
	05	001
	03	010
	02	011
	06	100
	01	101
	00 (EOB)	110
4 bits	07	1110
5 bits	08	1111 0
6 bits	09	1111 10
7 bits	0A	1111 110
8 bits	0B	1111 1110

0x6, for example, corresponds to code 100

For AC values, 'Run' lies in the range 0~15, and 'Size' lies in the range 0~10

Length	Value	Bits
2 bits	01	00
	02	01
3 bits	03	100
4 bits	00 (EOB)	1010
	04	1011
	11	1100
5 bits	05	1101 0
	12	1101 1
	21	1110 0
6 bits	31	1110 10
	41	1110 11
...	...	...
12 bits	24	1111 1111 0100
	33	1111 1111 0101
	62	1111 1111 0110
	72	1111 1111 0111
15 bits	82	1111 1111 1000 000
16 bits	09	1111 1111 1000 0010
	...	...
	FA	1111 1111 1111 1110

0x33, for example, corresponds to code 1111 1111 0101

**Finally we will get:**

## Encode

We saw above how the quantized image can be encoded into a stream of 0 or 1 for communication.

Here we start to encode our own image.

```

% At first, we need to convert the quantized image into
% a intermediary representation before it can be encoded.

% DPCM Coding for the DC coefficients

% Calculate the differences between DC coefficients
for i = NY*NX:-1:2;
    DC(i,:) = DC(i,:) - DC(i-1,:);
end

% Create a matrix to store the intermediary representations for 'Size'
DC_interS = zeros(NY*NX,3);

% Create a cell to store 'Value' codes
DC_V = cell(NY*NX,3);
% Create a cell to store 'Size' codes

```

```

DC_S = cell(NY*NX,3);

% RLE Coding for the AC coefficients

% Extract the nonzeros 'Value' from the AC coefficients
ACV = AC(:);
ACV(ACV == 0) = [];

% Create two matrix variable to store the
% intermediary representations for AC components
AC_interS = zeros(NY*NX,63,3);
% One for 'Size'
AC_interR = zeros(NY*NX,63,3);
% One for 'Run'

% Create a cell to store AC codes
AC_Code = cell(NY*NX,63,3);
AC_Code(:) = {' '};
% Create a cell to store 'Run/Size' codes
AC_RS = cell(NY*NX,63,3);

```

```

% As shown above, table for VLI coding and table for
% Huffman coding are needed to code the image.
% However, to incorporate such tables in our program here
% will be extremely cumbersome.
% Hence, we shall generate our own Huffman table
% and 'VLI table' for each input image.
% And they will all be based on the Huffman coding scheme.

% First we generate the 'VLI table'.
% Note that we can generate a table for each YUV components,
% and thus achieve a higher compression rate.
% However, the more table we use, the more space
% it will take to run the program.
% The message we send through the channel will grow as well.
% Hence, we will follow the example shown above
% and use one 'VLI table' for all the 'Value'.
% The Huffman table, though, will be tailored
% for Y_DC, UC_DC, Y_AC, UV_DC respectively.

```

## Huffman Table for Value

```

% First get the probabilities of all the 'Value' in both DC and AC components

Sta = tabulate([DC(:);ACV]);
% Call function 'huffman_scan()'
% This function takes a list of signals and their corresponding probabilities
% as input, and output the Huffman codes for these signals.
[Signal_Value,Code_Value] = huffman_code(Sta(:,1)',Sta(:,3)');
Signal_Value = cell2mat(Signal_Value);

```

## DC Intermediary Representation for Size

```
% Now we can tell the 'Size' according to the Code_Value
```

```
% For DC components
for i = 1:length(Signal_Value)
    templogic = DC==Signal_Value(i);
    DC_V(templogic) = Code_Value(i);
    DC_interS(templogic) = length(Code_Value{i});
end
% Now the 'Size' and 'Value' information are stored in two matrix, respectively
```

## DC Huffman Table for Size

```
% Then we generate the two Huffman tables for DC 'Size'
Sta = tabulate(DC_interS(:,1));
[Signal_DCY,Code_DCY] = huffman_code(Sta(:,1)',Sta(:,3)');
Signal_DCY = cell2mat(Signal_DCY);
% One for Y components
Sta = tabulate([DC_interS(:,2);DC_interS(:,3)]);
[Signal_DCUV,Code_DCUV] = huffman_code(Sta(:,1)',Sta(:,3)');
Signal_DCUV = cell2mat(Signal_DCUV);
% One for UV components
```

## DC Huffman Code for Size

```
% Now we can convert DC_interS into code
% For Y components
for i = 1:length(Signal_DCY)
    templogic = DC_interS==Signal_DCY(i);
    templogic(:,2:3) = 0;
    DC_S(templogic) = Code_DCY(i);
end
% For UV components
for i = 1:length(Signal_DCUV)
    templogic = DC_interS==Signal_DCUV(i);
    templogic(:,1) = 0;
    DC_S(templogic) = Code_DCUV(i);
end
```

## AC Intermediary Representation for Run/Size

```
% For AC components
% It is not that easy, because the format is heterogeneous for each block.
% Here, we will scan the AC matrix
% from the 1st (NY*NX,3) two dimentional matrix to the 63th.
% Note that we can also scan the 63 coefficients of each block.
% For higher speed, we do it matrix by matrix,
% although this requires a more sophisticated algorithm.

% We first convert the 'Value' into code, except the '0'
% The length of the code can be stored simutaneously

Signal_Value_X = Signal_Value; % Make a copy
Erase = Signal_Value_X==0;
Signal_Value_X(Erase)=[];
Code_Value_X = Code_Value; % Make a copy
Code_Value_X(Erase) = [];

for i = 1:length(Signal_Value_X)
    templogic = AC==Signal_Value_X(i);
```

```

AC_Code(templogic) = Code_Value_X(i);
AC_interS(templogic) = length(Code_Value_X{i});
end

% Now we encode the Run length of 0

Count = zeros(NY*NX,3); % A count matrix to store the run lenght

% Check if EOB
Check = sum(AC(:,1:63,:)==0,2);
Check = Check(:,1:3);
% Encode the EOB into AC_Code
tempCode = AC_Code(:,1,:);
tempCode(Check==0) = {'EOB'};
% This 'EOB' is simply as mark,
% and will be replaced by its code word later.
AC_Code(:,1,:) = tempCode;
% Disable the counter at the corresponding positon
Count(Check==0) = -127; % Arbitrarily taken

for i = 1:63      % Scan the 63 (NY*NX,3) two dimentional matrix
    temp = AC(:,i,:);      % Take out the ith matrix
    temp = temp(:,1:3);    % Flatten the matrix into tow dimentional

    templogic = temp==0; % If the current 'Value' is 0
    Count(templogic) = Count(templogic)+1;
    % Count the number of encountered 0

    tempcount = AC_interR(:,i,:); % If the current 'Value' is not 0
    tempcount(~templogic) = Count(~templogic);
    tempcount(Count==16) = 15; % If the Run length of 0 exceeds 15
    AC_interR(:,i,:) = tempcount; % Store the number of encountered 0

    % Reset the number of encountered 0
    Count(~templogic) = 0;
    Count(Count==16) = 0;

    % Check if EOB
    if i<63
        Check = sum(AC(:,i+1:63,:)==0,2);
        Check = Check(:,1:3);
        % Encode the EOB into AC_V
        tempCode = AC_Code(:,i+1,:);
        tempCode(Check==0 & Count>=0) = {'EOB'};
        AC_Code(:,i+1,:) = tempCode;
        % Disable the counter at the corresponding positon
        Count(Check==0) = -127; % Arbitrarily taken
    end
end

```

## AC Huffman Table for Run/Size

```

% What we have now is an intermediary AC_interS,
% which is a matrix containing the 'Size' in double format,
% and an intermediary AC_interR,
% which is a matrix containing the 'Run' in double format.
% AC_Code is already a cell containing the codes for 'Value'.

% The three matrix/cell have the same dimension,
% and for each nonempty element in AC_Code,

```

```

% the corresponding element in AC_interS stores the Size for it,
% and the corresponding element in AC_interR stroes the Run.

% However, some exception must be noticed.
% First, some elements in AC_Code are EOB.
% They have no corresponding elements in AC_interS or AC_interR.
% Second, some elements in AC_interR are 15.
% If the corresponding elements in AC_interS are 0,
% this will mean (15,0), which is 16 consecutive 0.
% There will be no corresponding elements in AC_Code.
% If the corresponding elements in AC_interS are not 0,
% this will mean (15,Value), with the code for 'Value' in
% the corresponding position in AC_Code.

% With these information in mind,
% we start to generate the Huffman Table for Run/Size.

% Combine Run and Size
AC_interRS = AC_interR.*100 + AC_interS;

ACY = AC_interRS(:,:,1);
ACY(ACY==0)=[];
Sta = tabulate([ACY(:);ones(NY*NX,1).*-1]);
% Note that we include the 'EOB' here, represented by -1.
% The number of EOB is roughly NY*NX*3 (one for each block)
[Signal_ACY,Code_ACY] = huffman_code(Sta(:,1)',Sta(:,3)');
Signal_ACY = cell2mat(Signal_ACY);
% One for Y components

ACUV = AC_interRS(:,:,2:3);
ACUV(ACUV==0)=[];
Sta = tabulate([ACUV(:);-1;ones(NY*NX*2,1).*-1]);
% Note that we include the 'EOB' here, represented by 0.
% The number of EOB is roughly NY*NX*2 (one for each block)
[Signal_ACUV,Code_ACUV] = huffman_code(Sta(:,1)',Sta(:,3)');
Signal_ACUV = cell2mat(Signal_ACUV);
% One for UV components

```

## AC Huffman Code for Run/Size

```

logic = zeros(NY*NX,63,3);
% Now we can convert AC_interRS into code
for i = 1:length(Signal_ACY)
    % For Y components
    templogic = AC_interRS==Signal_ACY(i);
    templogic(:,:,2:3) = 0;
    logic = logic+templogic;
    AC_RS(templogic) = Code_ACY(i);
end
for i = 1:length(Signal_ACUV)
    % For UV components
    templogic = AC_interRS==Signal_ACUV(i);
    templogic(:,:,1) = 0;
    logic = logic+templogic;
    AC_RS(templogic) = Code_ACUV(i);
end
logic = logical(logic);

```

## Send the Messege

```
% Now we have intotal four cell containing all the codes we need.  
% DC_S and DC_V for DC components  
% AC_RS and AC_V for AC components  
% We encode them together block by block.  
% Both of them contains three layers for Y,U,V  
% We encode them separately.  
  
DC_Code = strcat(DC_S,DC_V);  
  
AC_Code(logic) = strcat(AC_RS(logic),AC_Code(logic));  
% Combine the code for Size/Run and Value  
  
DC_CodeX = cell(NY*NX,1,3);  
DC_CodeX(:,1:3) = DC_Code;  
% Two dimension to Three dimention  
  
Code = [DC_CodeX,AC_Code];  
% Combine the code for DC and AC  
  
messegeY = cell2mat(reshape(Code(:,:,1)',1,NY*NX*64));  
messegeU = cell2mat(reshape(Code(:,:,2)',1,NY*NX*64));  
messegeV = cell2mat(reshape(Code(:,:,3)',1,NY*NX*64));  
% Convert cell into string  
  
messegeY(messegeY==' ')=[];  
messegeU(messegeU==' ')=[];  
messegeV(messegeV==' ')=[];  
% Erase the blank elements  
  
messegeY = strrep(messegeY,'EOB',Code_ACY{Signal_ACY== -1});  
messegeU = strrep(messegeU,'EOB',Code_ACUV{Signal_ACUV== -1});  
messegeV = strrep(messegeV,'EOB',Code_ACUV{Signal_ACUV== -1});  
% Replace the 'EOB' into its code word  
  
lengthY = length(messegeY);  
lengthU = length(messegeU);  
lengthV = length(messegeV);  
Length = lengthY + lengthU + lengthV;  
  
messege = strcat(messegeY,messegeU,messegeV);  
Messege = messege;
```

# JPEG CHANNEL

## Method One: HFC with Hamming Code

(7,4) Hamming Code.

```
% G = [1 0 0 0 1 0 1; ...
%       0 1 0 0 1 1 1; ...
%       0 0 1 0 1 1 0; ...
%       0 0 0 1 0 1 1];
% H = [1 1 1 0 1 0 0; ...
%       0 1 1 1 0 1 0; ...
%       1 1 0 1 0 0 1];
% % First we split the messege into 4-bits pieces
% append = mod(4 - mod(length(messege),4),4);
% messege = strcat(messege,num2str(zeros(1,append)));
% messege = str2num(messege(:))';
% % Add 0 at the end of the messege
% % Now the lenght of the messege is 4*N
%
% currentL = length(messege);
% Num = currentL/4;
% Rec = zeros(1,currentL);
%
% % Initiate the noise
% Perror = 0.01; % Set the error rate
% Noise = rand(1,7*Num);
% Noise(Noise<Perror) = 1;
% Noise(Noise~=1) = 0;
%
% for i = 1:Num % Send the messege piece by piece
%   % Noise
%   currentrec = bitxor(mult2ple(messege(4*(i-1)+1:4*i),G),Noise(7*(i-1)+1:7*i));
%   % Error Correction
%   switch num2str(mult2ple(currentrec,H'))
%     case '0 0 0'
%       Rec(4*(i-1)+1:4*i) = currentrec(1:4);
%     case '1 0 1'
%       temp = bitxor(currentrec,[1 0 0 0 0 0 0]);
%       Rec(4*(i-1)+1:4*i) = temp(1:4);
%     case '1 1 1'
%       temp = bitxor(currentrec,[0 1 0 0 0 0 0]);
%       Rec(4*(i-1)+1:4*i) = temp(1:4);
%     case '1 1 0'
%       temp = bitxor(currentrec,[0 0 1 0 0 0 0]);
%       Rec(4*(i-1)+1:4*i) = temp(1:4);
%     case '0 1 1'
%       temp = bitxor(currentrec,[0 0 0 1 0 0 0]);
%       Rec(4*(i-1)+1:4*i) = temp(1:4);
%     case '1 0 0'
%       temp = bitxor(currentrec,[0 0 0 0 1 0 0]);
%       Rec(4*(i-1)+1:4*i) = temp(1:4);
%     case '0 1 0'
%       temp = bitxor(currentrec,[0 0 0 0 0 1 0]);
%       Rec(4*(i-1)+1:4*i) = temp(1:4);
%     case '0 0 1'
%       temp = bitxor(currentrec,[0 0 0 0 0 0 1]);
%
```

```
%           Rec(4*(i-1)+1:4*i) = temp(1:4);
%       end
% end
```

As shown above, the naive way to do error correction in Hamming code is to use 'switch-case'.

But it will require tremendous amount of work to built a system for large n and k (and is slow).

Here we provide a autonomous 'Hamming code generator' to generate any possible Hamming code.

```
% n = 7;
% k = 4;
% % Only need to change these two lines
%
% append = mod(k - mod(length(messege),k),k);
% messege = strcat(messege,num2str(zeros(1,append)));
% messege = str2num(messege(:))';
%
% H = HGenerate(n-k);
% H = [H;H(2:n-k+1,:)];
% H(1:n-k+1,:) = [];
% H = H';
% G = H(:,1:k);
% G = [eye(k),G'];
%
% currentL = length(messege);
% Num = currentL/k;
% Rec = zeros(1,currentL);
%
% % Initiate the noise
% Perror = 0.01;
% Noise = rand(1,n*Num);
% Noise(Noise<Perror) = 1;
% Noise(Noise~=1) = 0;
%
% for i = 1:Num % Send the messege piece by piece
%     % Noise
%     currentrec = bitxor(mult2ple(messege(k*(i-1)+1:k*i),G),Noise(n*(i-1)+1:n*i));
%     % Error Correction
%     tempS = mult2ple(currentrec,H');
%     tempS = tempS';
%     if sum(tempS) == 0
%         Rec(k*(i-1)+1:k*i) = currentrec(1:k);
%     else
%         position = 1;
%         while sum(tempS == H(:,position)) ~= n-k
%             position = position+1;
%         end
%         error = zeros(1,n);
%         error(position)=1;
%         correction = bitxor(currentrec,error);
%         Rec(k*(i-1)+1:k*i) = correction(1:k);
%     end
% end
```

## Method Two: ARQ with CRC

(38, 22) CRC, with  $g(x) = x^{16} + x^{12} + x^5 + 1$  [1 0 0 0 1 0 0 0 0 0 1 0 0 0 1]

```
% append = mod(22 - mod(length(messege),22),22);
% messege = strcat(messege,num2str(zeros(1,append)));
% messege = str2num(messege(:))';
%
% gx = [1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1];
% currentL = length(messege);
% Num = currentL/22;
% Rec = zeros(1,currentL);
%
% Trytimes = zeors(1,Num);
% % This is used to record how much times it takes
% % to successfully send a piece of messege of length 22.
%
% for i = 1:Num
%     currentmessege = [messege(11*(i-1)+1:11*i) zeros(1,16)];
%     temp = currentmessege;
%
%     while(1) % Compute r(x)
%         if sum(temp)>0
%             pivot = find(temp == 1);
%         else
%             break
%         end
%         if (length(temp)-pivot(1))<16
%             break
%         end
%         temp(pivot(1):pivot(1)+16) = bitxor(temp(pivot(1):pivot(1)+16),gx);
%     end
%     currentsend = [currentmessege temp(length(temp)-15:length(temp))];
%     % r(x) always contains 16 words
%
%     Trytimes(i) = 1;
%     while(1)
%         % Add noise
%         Perror = 0.01; % The probability of error can be variable as well
%         Noise = rand(1,38);
%         Noise(Noise<Perror) = 1;
%         Noise(Noise~=1) = 0;
%         currentrec = bitxor(currentsend,Noise);
%
%         % Check if there is an error
%         temp = currentrec;
%         while(1) % Compute r(x)
%             if sum(temp)>0
%                 pivot = find(temp == 1);
%             else
%                 break
%             end
%             if (length(temp)-pivot(1))<16
%                 break
%             end
%             temp(pivot(1):pivot(1)+16) = bitxor(temp(pivot(1):pivot(1)+16),gx);
%         end
%
%         if sum(temp)==0    % Success
%             break
%         else
%             Trytimes(i) = Trytimes(i) + 1; % Send the messege again
%
```

```
%           end
%       end
% end
```

### Method Three: FEC with Convolutional Code

(3,1,2) convolutional code, with transfer function matrix  $[1, 1+D, 1+D+D^2]$

```
% %% Convolutional Encoding
%
% k = 1;
% n = 3;
% L = 2;
% %%%%%%
% %%% To change the code, these lines should be modified.
% %%%%%%
%
% append = mod(k - mod(length(messeege),k),k);
% messeege = strcat(messeege,num2str(zeros(1,append)));
% messeege = str2num(messeege(:))';
%
% % Specify the transfer matrix
% G = zeros(k,n,L+1);
% G(:,:,1) = [1,1,1]; % G0
% G(:,:,2) = [0,1,1]; % G1
% G(:,:,3) = [0,0,1]; % G2
% %%%%%%
% %%% To change the code, these lines should be modified.
% %%%%%%
%
%
% currentL = length(messeege);
% Num = currentL/k;
% Send = zeros(Num+L,n);
% realmesseege = messeege;
% messeege = [realmesseege, zeros(1,k*L)];
%
% % Initiate the noise
% Perror = 0.01;
% Noise = rand(Num+L,n);
% Noise(Noise<Perror) = 1;
% Noise(Noise~=1) = 0;
%
%
% Stack = zeros(L+1,k); % Create a stack to store the current information
%
%
% for i=1:Num+L
%
%     % Information Flow
%     for j = L+1:-1:2
%         Stack(j,:) = Stack(j-1,:); % Move the information flow
%     end
%     Stack(1,:)= messeege(k*(i-1)+1:k*i); % New information
%
%     % Code Generation
%     for j = 1:L+1
%         Send(i,:) = bitxor(Send(i,:),mult2ple(Stack(j,:),G(:,:,j)));
%         Send(i,:) = bitxor(Send(i,:),Noise(i,:));
%
```

```

%      end
%
% end
%
%
% %% Convolutional Decoding: Viterbi algorithm
%
% D = 9; % Time Delay. Must be non-negative. Can be arbitrarily modified.
% Recbox = zeros(1,n*(Num+D+L));
%
% % Create a memory cell
% State = 4; % Four states for (3,1,2) code
% %%%%%%
% %%% To change the code, this lines should be modified.
% %%% Because different code have different number of states.
% %%%%%%
%
% Memo = cell(State,D);
% for i = 1:State
%     for j = 1:D
%         Memo{i,j} = zeros(1,n); % Initiate the memory cell
%     end
% end
%
% % Create a PM vector
% PM = ones(State,1)*Inf;
% PM(1) = 0;
%
%
% % Create a reference cell for the calculation of BM
% % and for the updating of the memory cell
% Ref = cell(State,State);
% Ref{1,1}=[0 0 0];Ref{1,3}=[1 1 1];
% Ref{2,1}=[0 0 1];Ref{2,3}=[1 1 0];
% Ref{3,2}=[0 1 1];Ref{3,4}=[1 0 0];
% Ref{4,2}=[0 1 0];Ref{4,4}=[1 0 1];
% %%%%%%
% %%% To change the code, these lines should be modified.
% %%%%%%
%
% % Create a BM matrix
% BM = zeros(4,4);
% bm = zeros(4,4);
% bm(1,2)=Inf;bm(1,4)=Inf;
% bm(2,2)=Inf;bm(2,4)=Inf;
% bm(3,1)=Inf;bm(3,3)=Inf;
% bm(4,1)=Inf;bm(4,3)=Inf;
% %%%%%%
% %%% To change the code, this lines should be modified.
% %%%%%%
%
% % Decoding start
%
% for count = 1:Num+L
%     currentcode = Send(count,:);
%
%     % Calculate BM
%     for i = 1:State
%         for j = 1:State
%             if ~isempty(Ref{i,j})
%                 BM(i,j)=sum(bitxor(Ref{i,j},currentcode));
%
```

```

%
%           end
%
%       end
%
%
%       % Update PM and the memory cell
%       currentmin = Inf;
%       newPM = zeros(State,1);
%       tempBM = BM + bm;
%       newMemo = cell(State,D);
%
%       for i = 1:State
%           newPM(i) = min(PM+tempBM(:,i));
%           if newPM(i) ~= Inf
%               Index = find((PM+tempBM(:,i)) == newPM(i));
%               Index = Index(1);
%               newMemo(i,1:D-1) = Memo(Index,2:D);
%               newMemo(i,D) = Ref(Index,i);
%               if newPM(i)<currentmin
%                   currentmin = newPM(i);
%                   output = Memo{Index,1};
%               end
%           end
%       end
%
%       PM = newPM;
%       Memo = newMemo;
%       Recbox(n*(count-1)+1:n*count) = output;
%   end
%
%   % Extract the rest of the memory cell.
%   Recbox(n*(Num+L)+1:n*(Num+L+D)) = cell2mat(Memo(Index,1:D));
%   Recbox = reshape(Recbox,3,Num+L+D)';
%
%   % From code to messege
%
%   % Create a referece matrix. The decoder refer to this matrix,
%   % and find the corresponding k-bit messege of the current n-bit code
%   mRef = zeros(2,2,2);
%  %%%%%
%
%   %% To change the code, these lines should be modified.
%   %% This matrix is specifically designed for our code here.
%   %%%%%%
%
%   mRef(2,2,2)=1;
%   mRef(2,2,1)=1;
%   mRef(2,1,1)=1;
%   mRef(2,1,2)=1;
%
%   % Rec=zeros(1,currentL+L+D);
%   % for i = 1:Num+L+D
%   %     currentcode = Recbox(i,:);
%   %     Rec(i) = mRef(currentcode(1)+1,currentcode(2)+1,currentcode(3)+1);
%   % end
%
%   % The real messege is
%   % Rec = Rec(D+1:D+currentL);

```

# JPEG DECODING

## Procedure Preview:

```
recmessege —— decoded —— revimageZ(double) ——  
revimageQ (double) —— IDCT(double) —— revimage (uint8)
```

```
Rec = Messege;  
Rec = num2str(Rec);  
Rec(Rec==' ') = '';  
recmessegeY = Rec(1:lengthY);  
recmessegeU = Rec(lengthY+1:lengthY+lengthU);  
recmessegeV = Rec(lengthY+lengthU+1:Length);  
  
% Decode the messege into pixel values  
  
% For Y components  
decodedY = zeros(1,NY*NX*64);  
currentelement = 1; % Start with the first pixel  
BlockCount = NY*NX;  
prev = 0;  
while BlockCount > 0  
    Stop = 1;  
    tempcode = recmessegeY(1:Stop); % The current code  
    % The first element of a block is a DC coefficient.  
    % Thus we shall refer to the DC Huffman Table,  
    % which is stored as Signal_DCY and Code_DCY  
    indexDC = find(strcmp(Code_DCY,tempcode));  
    while isempty(indexDC) % If the current code is not a code word  
        Stop = Stop + 1; % Eat the next code bit  
        tempcode = recmessegeY(1:Stop);  
        indexDC = find(strcmp(Code_DCY,tempcode));  
    end % Until the current code is a code word for DC 'Size'  
    Size = Signal_DCY(indexDC);  
    % Successfully decode the DC 'Size'  
    % The next Size bits will be the code word for DC 'Value'  
    indexDC = find(strcmp(Code_Value,recmessegeY(Stop+1:Stop+Size)));  
    decodedY(currentelement) = Signal_Value(indexDC) + prev;  
    prev = decodedY(currentelement);  
    % Reverse DPCM Coding  
    recmessegeY(1:Stop+Size) = []; % Erase the decoded messege  
    currentelement = currentelement + 1;  
    % Move on to AC coefficients  
    %a=input('Move on to AC')  
    ACCount = 63;  
    while ACCount > 0  
        Stop = 1;  
        tempcode = recmessegeY(1:Stop); % The current code  
        indexAC = find(strcmp(Code_ACY,tempcode));  
        while isempty(indexAC) % If the current code is not a code word  
            Stop = Stop + 1; % Eat the next code bit  
            tempcode = recmessegeY(1:Stop);  
            indexAC = find(strcmp(Code_ACY,tempcode));  
        end % Until the current code is a code word for AC 'Run/Size'  
        RunSize = Signal_ACY(indexAC);  
        % Successfully decode the AC 'Run/Size'  
        % Check if EOB first  
        if RunSize == -1
```

```

        decodedY(currentelement:currentelement+ACCount-1) = 0;
        recmessegeY(1:Stop) = []; % Erase the decoded messege
        currentelement = currentelement + ACCOUNT;
        ACCOUNT = 0;
    else
        Run = floor(RunSize/100); % Decode the 'Run' information
        if Run > 0
            decodedY(currentelement:currentelement+Run-1) = 0;
            currentelement = currentelement + Run;
            ACCOUNT = ACCOUNT - Run;
        end
        Size = mod(RunSize,100); % Decode the 'Size' information
        if Size == 0 % This happens only for 16 consecutive 0
            decodedY(currentelement) = 0;
            recmessegeY(1:Stop) = []; % Erase the decoded messege
            currentelement = currentelement + 1;
        else % The next Size bits will be the code word for AC 'Value'
            index = find(strcmp(Code_Value,recmessegeY(Stop+1:Stop+Size)));
            decodedY(currentelement) = Signal_Value(index);
            recmessegeY(1:Stop+Size) = []; % Erase the decoded messege
            currentelement = currentelement + 1;
        end
        ACCOUNT = ACCOUNT - 1; % Move on to the next AC coefficient
    end
end
BlockCount = BlockCount - 1; % Move on to the next Block
end

% For U components
decodedU = zeros(1,NY*NX*64);
currentelement = 1; % Start with the first pixel
BlockCount = NY*NX;
prev = 0;
while BlockCount > 0
    Stop = 1;
    tempcode = recmessegeU(1:Stop); % The current code
    % The first element of a block is a DC coefficient.
    % Thus we shall refer to the DC Huffman Table,
    % which is stored as Signal_DCY and Code_DCY
    indexDC = find(strcmp(Code_DCUV,tempcode));
    while isempty(indexDC) % If the current code is not a code word
        Stop = Stop + 1; % Eat the next code bit
        tempcode = recmessegeU(1:Stop);
        indexDC = find(strcmp(Code_DCUV,tempcode));
    end % Until the current code is a code word for DC 'Size'
    Size = Signal_DCUV(indexDC);
    % Successfully decode the DC 'Size'
    % The next Size bits will be the code word for DC 'Value'
    indexDC = find(strcmp(Code_Value,recmessegeU(Stop+1:Stop+Size)));
    decodedU(currentelement) = Signal_Value(indexDC) + prev;
    prev = decodedU(currentelement);
    % Reverse DPCM Coding
    recmessegeU(1:Stop+Size) = []; % Erase the decoded messege
    currentelement = currentelement + 1;
    % Move on to AC coefficients
    %a=input('Move on to AC')
    ACCOUNT = 63;
    while ACCOUNT > 0
        Stop = 1;
        tempcode = recmessegeU(1:Stop); % The current code
        indexAC = find(strcmp(Code_ACUV,tempcode));

```

```

while isempty(indexAC) % If the current code is not a code word
    Stop = Stop + 1; % Eat the next code bit
    tempcode = recmesegeU(1:Stop);
    indexAC = find(strcmp(Code_ACUV,tempcode));
end % Until the current code is a code word for AC 'Run/Size'
RunSize = Signal_ACUV(indexAC);
% Successfully decode the AC 'Run/Size'
% Check if EOB first
if RunSize == -1
    decodedU(currentelement:currentelement+ACCount-1) = 0;
    recmesegeU(1:Stop) = []; % Erase the decoded messege
    currentelement = currentelement + ACCount;
    ACCount = 0;
else
    Run = floor(RunSize/100); % Decode the 'Run' information
    if Run > 0
        decodedU(currentelement:currentelement+Run-1) = 0;
        currentelement = currentelement + Run;
        ACCount = ACCount - Run;
    end
    Size = mod(RunSize,100); % Decode the 'Size' information
    if Size == 0 % This happens only for 16 consecutive 0
        decodedU(currentelement) = 0;
        recmesegeU(1:Stop) = []; % Erase the decoded messege
        currentelement = currentelement +1;
    else % The next Size bits will be the code word for AC 'Value'
        index = find(strcmp(Code_Value,recmesegeU(Stop+1:Stop+Size)));
        decodedU(currentelement) = Signal_Value(index);
        recmesegeU(1:Stop+Size) = []; % Erase the decoded messege
        currentelement = currentelement + 1;
    end
    ACCount = ACCount - 1; % Move on to the next AC coefficient
end
end
BlockCount = BlockCount - 1; % Move on to the next Block
end

% For V components
decodedV = zeros(1,NY*NX*64);
currentelement = 1; % Start with the first pixel
BlockCount = NY*NX;
prev = 0;
while BlockCount > 0
    Stop = 1;
    tempcode = recmesegeV(1:Stop); % The current code
    % The first element of a block is a DC coefficient.
    % Thus we shall refer to the DC Huffman Table,
    % which is stored as Signal_DCY and Code_DCY
    indexDC = find(strcmp(Code_DCUV,tempcode));
    while isempty(indexDC) % If the current code is not a code word
        Stop = Stop + 1; % Eat the next code bit
        tempcode = recmesegeV(1:Stop);
        indexDC = find(strcmp(Code_DCUV,tempcode));
    end % Until the current code is a code word for DC 'Size'
    Size = Signal_DCUV(indexDC);
    % Successfully decode the DC 'Size'
    % The next Size bits will be the code word for DC 'Value'
    indexDC = find(strcmp(Code_Value,recmesegeV(Stop+1:Stop+Size)));
    decodedV(currentelement) = Signal_Value(indexDC) + prev;
    prev = decodedV(currentelement);
    % Reverse DPCM Coding

```

```

recmessegeV(1:Stop+Size) = []; % Erase the decoded messege
currentelement = currentelement + 1;
% Move on to AC coefficients
%a=input('Move on to AC')
ACCount = 63;
while ACCount > 0
    Stop = 1;
    tempcode = recmessegeV(1:Stop); % The current code
    indexAC = find(strcmp(Code_ACUV,tempcode));
    while isempty(indexAC) % If the current code is not a code word
        Stop = Stop + 1; % Eat the next code bit
        tempcode = recmessegeV(1:Stop);
        indexAC = find(strcmp(Code_ACUV,tempcode));
    end % Until the current code is a code word for AC 'Run/Size'
    RunSize = Signal_ACUV(indexAC);
    % Successfully decode the AC 'Run/Size'
    % Check if EOB first
    if RunSize == -1
        decodedV(currentelement:currentelement+ACCount-1) = 0;
        recmessegeV(1:Stop) = []; % Erase the decoded messege
        currentelement = currentelement + ACCOUNT;
        ACCOUNT = 0;
    else
        Run = floor(RunSize/100); % Decode the 'Run' information
        if Run > 0
            decodedV(currentelement:currentelement+Run-1) = 0;
            currentelement = currentelement + Run;
            ACCOUNT = ACCOUNT - Run;
        end
        Size = mod(RunSize,100); % Decode the 'Size' information
        if Size == 0 % This happens only for 16 consecutive 0
            decodedV(currentelement) = 0;
            recmessegeV(1:Stop) = []; % Erase the decoded messege
            currentelement = currentelement + 1;
        else % The next Size bits will be the code word for AC 'Value'
            index = find(strcmp(Code_Value,recmessegeV(Stop+1:Stop+Size)));
            decodedV(currentelement) = Signal_Value(index);
            recmessegeV(1:Stop+Size) = []; % Erase the decoded messege
            currentelement = currentelement + 1;
        end
        ACCOUNT = ACCOUNT - 1; % Move on to the next AC coefficient
    end
end
BlockCount = BlockCount - 1; % Move on to the next Block
end

% Reverse Z Scanning
revimageZY = revZscan(decodedY,NY,NX,Zig_Zag);
revimageZU = revZscan(decodedU,NY,NX,Zig_Zag);
revimageZV = revZscan(decodedV,NY,NX,Zig_Zag);

% Reverse quantization
revimageQY = revimageZY.*YQT;
revimageQU = revimageZU.*UVQT;
revimageQV = revimageZV.*UVQT;

% Reverse DCT
IDCT_Y = zeros(height,width);
IDCT_U = IDCT_Y;
IDCT_V = IDCT_Y;

```

```

for yindex = 1:NY
    for xindex = 1:NX
        IDCT_Y(8*(yindex-1)+1:8*(yindex-1)+8,8*(xindex-1)+1:8*(xindex-1)+8) ...
            = C*revimageQY(8*(yindex-1)+1:8*(yindex-1)+8,... ...
            8*(xindex-1)+1:8*(xindex-1)+8,1)*C';
        IDCT_U(8*(yindex-1)+1:8*(yindex-1)+8,8*(xindex-1)+1:8*(xindex-1)+8) ...
            = C*revimageQU(8*(yindex-1)+1:8*(yindex-1)+8,... ...
            8*(xindex-1)+1:8*(xindex-1)+8,1)*C';
        IDCT_V(8*(yindex-1)+1:8*(yindex-1)+8,8*(xindex-1)+1:8*(xindex-1)+8) ...
            = C*revimageQV(8*(yindex-1)+1:8*(yindex-1)+8,... ...
            8*(xindex-1)+1:8*(xindex-1)+8,1)*C';
    end
end

% YUV to RGB

% R = Y + 1.402*V
% G = Y - 0.34414*U - 0.71414*V
% B = Y + 1.772*U

R = IDCT_Y + 1.402.*IDCT_V + 128;
G = IDCT_Y - 0.34414.*IDCT_U - 0.71414.*IDCT_V + 128;
B = IDCT_Y + 1.772.*IDCT_U + 128;

revimage = uint8(cat(3,R,G,B));

% Reframe if necessary
revimage = revimage(1:oriheight,1:oriwidth,:);

```