

Multimedia Technologies

Project Report

1. Members

序号	学号	专业班级	姓名	性别	分工
1	3180106148	计科 1802	李向	男	全部

2. Project Introduction

2.1 选题

图像压缩器（基于标准 JPEG 图像压缩流程）

2.2 工作简介

利用 MATLAB 作为开发工具，实现了一个可以读取 BMP 图像文件，并参照标准 JPEG 压缩流程对其进行压缩并保存，之后可以将压缩后的文件读取并解码后重新显示的图像压缩器。最后通过对实验结果的分析，对标准的 JPEG 压缩流程进行了一些修改，探索了要如何获得更好的压缩效果。

2.3 开发环境

- 开发工具：MATLAB R2019a
- 开发环境：Windows 10
- 系统配置：Intel Core i7-8550U 8GB RAM

3. Technical Details

3.1 压缩流程概述

本次实验中的压缩流程如下所示：

1. 读取 BMP 位图。
2. 图像补全，保证其长宽为 8 的倍数。
3. RGB 转换为 YCbCr，并进行颜色下采样（4:2:0）[有信息损失]。
4. 图像分块进行 DCT 变换处理。
5. 对 DCT 变换后的图像进行量化 [有信息损失]。
6. Zigzag 一维化。
7. DC 系数使用 DPCM 编码 [有信息损失]。
8. AC 系数使用 RLC 编码。
9. 对编码后的两个系数使用 Huffman 编码。
10. 保存压缩后的数据。

3.2 读取位图

调用 MATLAB 内置的 `imread()` 函数实现对图像的读取。

函数说明：

```
A = imread(filename)
```

功能：从 `filename` 指定的文件读取图像，并从文件内容推断出其格式。如果 `filename` 为多图像文件，则 `imread` 读取该文件中的第一个图像。

3.3 图像补全

由于后续需要将整个图像分成 8×8 的小块来分别进行处理，所以需要编写 `resize()` 函数来将原始的图像转换为长宽为 8 的倍数的图像。

函数说明：

```
new_img = resize(img)
```

功能：传入参数为 `img`，将 `img` 的长和宽均补全为 8 的倍数，补充的部分均使用边界处的像素值进行填充，最后返回 `new_img`。

实现原理：利用 $\text{ceil}(x/8) * 8$ 这个简单的公式产生对于 x 来说向上取整的 8 的倍数，接着将图像边界的像素值赋给填充部分即可。

3.4 颜色空间转换与下采样

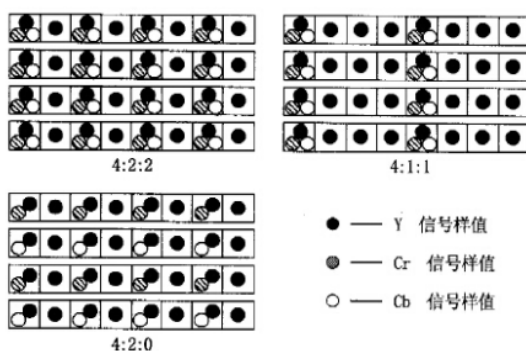
调用 MATLAB 内置的 `rgb2ycbcr()` 函数实现对颜色空间的转换，编写 `downSample()` 函数来对原始 YCbCr 图像进行 4:2:0 颜色下采样，4:2:0 的采样比例可使图像大小压缩到原来的一半。

理论知识：

JPEG 采用的是 YCbCr 颜色空间，而 BMP 采用的是 RGB 颜色空间，要想对 BMP 图片进行压缩，首先需要进行颜色空间的转换。YCbCr 颜色空间中，Y 代表亮度，而 Cb 和 Cr 则为蓝色和红色的浓度偏移量成份（色差）。RGB 和 YCbCr 之间的转换关系如下所示：

$$\begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} + \begin{bmatrix} 0 \\ 0.5 \\ 0.5 \end{bmatrix}$$

研究发现，人眼对亮度变换的敏感度要比对色彩变换的敏感度高出很多。因此，因此，可以认为 Y 分量比 Cb 和 Cr 分量更为重要。故采样时通常会降低 Cb，Cr 分量的采样率，目前主要的采样格式有 YCbCr 4:2:0、YCbCr 4:2:2、YCbCr 4:1:1 和 YCbCr 4:4:4。前三种采样方式的示意图如下所示：



本次实验中采用 4:2:0 采样方式，它表示第一行中，每连续四个点， 采样两个 CbCr，在紧邻的第二行中不采样 CbCr。实际操作的过程中可以分为 2×2 的单元，每个单元中 Y 值采样 4 次，而只采样左上角的 Cb 值和左下角的 Cr 值，作为整个单元的 CbCr 值。这样采样的优点是虽然损失了一定精度，但也在人眼几乎不可见的前提下减小了数据存储量。

函数说明：

`YCBCR = rgb2ycbcr(RGB)`

功能：将 RGB 图像的红色、绿色和蓝色值转换为 YCbCr 图像的亮度 (Y) 和色度 (Cb 和 Cr) 值。

`[y,cb,cr] = downSample(I)`

功能：传入参数为原始的 YCbCr 图像矩阵，输出为 4:2:0 采样后的 Y，Cb 和 Cr 三个分量的矩阵。

实现原理：对输入图像矩阵的 Y 通道不做修改，图像中每个 2×2 的小块，只取左上角的 Cb 值和左下角的 Cr 值，最后将这三个通道分别输出即可。

3.5 DCT 变换

调用 MATLAB 内置的 `dct2()` 函数以及 `blockproc()` 函数实现对图像的分块 DCT 变换处理。

理论知识：

DCT 变换的全称是离散余弦变换(Discrete Cosine Transform)，主要用于数据或图像的压缩，能够将空域的信号转换到频域上，具有良好的去相关性的性能。DCT 变换本身是无损的，但是在图像编码等领域给接下来的量化、霍夫曼编码等创造了很好的条件，同时，由于 DCT 变换是对称的，所以，我们可以在量化编码后利用 DCT 反变换，在接收端恢复原始的图像信息。DCT 变换在当前的图像分析以及压缩领域有着极为广泛的用途，我们常见的 JPEG 静态图像编码以及

MJPEG、MPEG 动态编码等标准中都使用了 DCT 变换。

首先来看一维 DCT 变换，一维 DCT 变换共有 8 种形式，其中最常用的是第二种形式，由于其运算简单、适用范围广，我们在这里只讨论这种形式，其表达式如下：

$$F(u) = c(u) \sum_{i=0}^{N-1} f(i) \cos \left[\frac{(i+0.5)\pi}{N} u \right]$$

$$c(u) = \begin{cases} \sqrt{\frac{1}{N}}, & u = 0 \\ \sqrt{\frac{2}{N}}, & u \neq 0 \end{cases}$$

其中， $f(i)$ 为原始的信号， $F(u)$ 是 DCT 变换后的系数， N 为原始信号的点数， $c(u)$ 可以认为是一个补偿系数，可以使 DCT 变换矩阵为正交矩阵。

二维 DCT 变换其实是在一维 DCT 变换的基础上再做了一次 DCT 变换，其公式如下：

$$F(u, v) = c(u)c(v) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i, j) \cos \left[\frac{(i+0.5)\pi}{N} u \right] \cos \left[\frac{(j+0.5)\pi}{N} v \right]$$

$$c(u) = \begin{cases} \sqrt{\frac{1}{N}}, & u = 0 \\ \sqrt{\frac{2}{N}}, & u \neq 0 \end{cases}$$

另外，由于 DCT 变换高度的对称性，在使用 Matlab 进行相关的运算时，我们可以使用更简单的矩阵处理方式：

$$F = AfA^T$$

$$A(i, j) = c(i) \cos \left[\frac{(j+0.5)\pi}{N} i \right]$$

函数说明：

`B = blockproc(A, [m n], fun)`

功能：通过对大小为 $[m \ n]$ 的每个非重叠图像块应用 `fun` 函数并将结果串联到输出矩阵 `B` 中，来处理图像 `A`。

```
B = dct2(A)
```

功能：返回 A 的二维离散余弦变换。矩阵 B 包含离散余弦变换系数 B(k1,k2)。

3.6 量化

编写 `quantify()` 函数对 DCT 变换后的矩阵分块进行量化。

理论知识：

由于大多数图像的高频分量比较小，相应的图像高频分量的 DCT 系数经常接近于 0，再加上高频分量中只包含了图像的细微的细节变化信息，而人眼对这种高频成分的失真不太敏感，所以，可以考虑将这一些高频成分予以抛弃，从而降低需要传输的数据量。这样一来，传送 DCT 变换系数的所需要的编码长度要远远小于传送图像像素的编码长度。到达接收端之后通过反离散余弦变换就可以得到原来的数据，虽然这么做存在一定的失真，但人眼是可接受的，而且对这种微小的变换是不敏感的，而实现这种抛弃的方法就是量化。

量化阶段需要两个量化表，一个针对亮度(细粒度)，另一个则针对色度(粗粒度)，如下所示：

Table 9.1 The luminance quantization table	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

Table 9.2 The chrominance quantization table	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
	99	99	99	99	99	99	99	99

所谓量化就是用像素值 ÷ 量化表对应值所得的结果。由于量化表左上角的值

较小，右上角的值较大，这样就起到了保持低频分量，抑制高频分量的目的。之前我们提到过，在 YCbCr 色彩空间下，Y 分量更重要一些。因此我们可以对 Y 通道采用细量化，对 CbCr 通道采用粗量化，可进一步提高压缩比。这也就是为什么色度量化表不如亮度量化表精细的原因。

函数说明：

`[qy,qcb,qcr] = quantify(y,cb,cr)`

功能：传入参数为原始的 dct 变换后的 YCbCr 图像矩阵，输出为经亮度色度两张量化表量化后的图像矩阵。

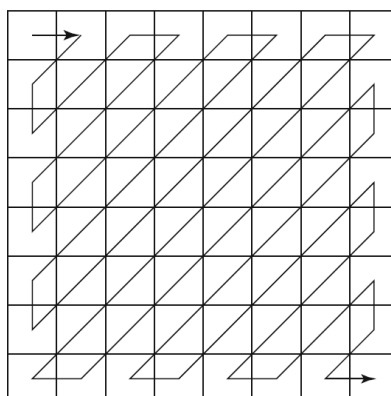
实现原理：对输入图像矩阵的 Y 通道，每个 8×8 小块中的值与亮度量化表中的值对应相除并取整；对输入图像矩阵的 Cb、Cr 通道，每个 8×8 小块中的值与色度量化表中的值对应相除并取整，将处理后的三个矩阵返回即可。

3.7 Zigzag

编写 `zigzag()` 函数实现对矩阵数据的一维化，需要利用到 MATLAB 内置的 `reshape()` 函数。

理论知识：

DCT 将一个 8×8 的数组变换成另一个 8×8 的数组，但是由于内存里所有数据都是线形存放的， 8×8 的数组中每行的结尾的点和下行开始的点原本在空间上是没有任何联系的。因此我们需要按照一个特定的取值顺序来将量化后的二维矩阵转变成一个一维数组，这就是 Zigzag 处理，其取值的示意图如下所示：



Zigzag 处理后，数列里的相邻点在原图片空间上也是相邻的了，且低频部分在一维数列的前端，高频部分放到了一维数列的末端，这样就便于之后的编码处理。

函数说明：

`B = reshape(A,sz1,...,szN)`

功能：将 A 重构为一个 $sz1 \times \dots \times szN$ 数组，其中 $sz1 \dots szN$ 指示每个维度的大小。可以指定 [] 的单个维度大小，以便自动计算维度大小，以使 B 中的元素数与 A 中的元素数相匹配。例如，如果 A 是一个 10×10 矩阵，则 `reshape(A,2,2,[])` 将 A 的 100 个元素重构为一个 $2 \times 2 \times 25$ 数组。但要注意这个函数是按列读取，按列存取的，如下图的实例所示，A 矩阵按列看是 14 25 36，所以调整为 3×2 之后按列看变为了 142 356。

```
A = [1,2,3;4,5,6]

A = 2x3
    1    2    3
    4    5    6

B = reshape(A,[3,2])

B = 3x2
    1    5
    4    3
    2    6
```

`zigx = zigzag(x)`

功能：传入参数为量化后的图像矩阵，输出为每个原图像矩阵的每个 8×8 小块都经 zigzag 一维化后的图像矩阵。

实现原理：比较常规的思路是按照 zigzag 的顺序，在原图像矩阵上通过 for 循环来一一取出对应的元素填充到一维数组中，但这样的实现过程比较复杂，且容易出错，于是我采用了一种先打表，后查表的办法来实现 zigzag。如下图所示，首先在一个 8×8 矩阵中标出各个位置的下标，并用红色线条表示出 zigzag 遍历数组的顺序。接着根据这个遍历顺序，将下标值依次填到另一个 8×8 矩阵

中（如下图右半部分所示），这样就构造出了一张 **zigzag** 查询表，这张表中的值对应原矩阵的下标。例如要找 Zigzag 处理后数列的第三个数，先看这张表的第三个数为 9，所以原数组中下标为 9 的数值即为所求。通过这样的方式就可以很容易地将一个 8×8 矩阵按 zigzag 的顺序输出为一维数组了。

1	2	3	4	5	6	7	8	1	2	9	17	10	3	4	11
9	10	11	12	13	14	15	16	18	25	33	26	19	12	5	6
17	18	19	20	21	22	23	24	13	20	27	34	41	49	42	35
25	26	27	28	29	30	31	32	28	21	14	7	8	15	22	29
33	34	35	36	37	38	39	40	36	43	50	57	58	51	44	37
41	42	43	44	45	46	47	48	30	23	16	24	31	38	45	52
49	50	51	52	53	54	55	56	59	60	53	46	39	32	40	47
57	58	59	60	61	62	63	64	54	61	62	55	48	56	63	64

算法实现过程中的一个比较重要的细节是，对每个 8×8 小块 zigzag 处理后，生成的一维数组最好是利用 `reshape()` 函数转换为列向量，这样新矩阵的第一行即为 DC 系数，其余行为 AC 系数，便于后续对 DC 系数和 AC 系数分别进行不同的编码处理。

3.8 DPCM 编码

编写 `dpcm()` 函数对 DC 系数进行编码。

理论知识：

因为得到的 DC 系数有两个特点：（1）系数的数值比较大；（2）相邻的 8×8 图像块的 DC 系数值变化不大。根据这两个特点，DC 系数一般采用差分脉冲调制编码 DPCM（Difference Pulse Code Modulation），即：取同一个图像分量中每个 DC 值与前一个 DC 值的差值来进行编码。对差值进行编码所需要的位数会比对原值进行编码所需要的位数少了很多。

DPCM 是对模拟信号幅度抽样的差值进行量化编码的调制方式，这种方式是用过去的抽样值来预测当前的抽样值，对它们的差值进行编码。差值编码可以提高编码频率，这种技术已应用于模拟信号的数字通信之中。DPCM 与预测编码类似，只是它有一个量化步骤，这也导致了其为有损编码。DPCM 的操作流程是，首先生成预测值，然后用原始信号值减去预测信号生成误差值，接着量化误差值，

得到量化后的误差值，并进行传输。重建时，利用量化后的误差值和预测信号来重构信号。

函数说明：

`en = dpcm(x)`

功能：传入参数为 DC 系数构成的矩阵，输出为量化后的误差值 `en` (`en(1)` 为原始信号 `f(1)`，其余为误差值)。

实现原理：首先进行变量的初始化，获得原始信号的长度为 `num`，重构信号 `f_rec`、预测信号 `f_pre` 和误差值 `e`、量化后的误差值 `en` 都是与原始信号 `f` 长度相同的零矩阵。`f_pre(1)`，`f_pre(2)`，`f_rec(1)`以及 `en(1)`的值均初始化为 `f(1)`。接着进行主体部分的计算，伪代码如下所示：

```
for i = 2 → num do
    if i ≠ 2 then
        f_pre[i] ← (f_rec[i - 1] + f_rec[i - 2])/2
    end if
    e[i] ← f[i] - f_pre[i]
    en[i] ← 16 * floor((255 + e[i])/16) - 256 + 8
    f_rec[i] ← f_pre[i] + en[i]
end for
```

3.9 RLC 编码

编写 `rlc()` 函数对 AC 系数进行编码。

理论知识：

量化之后的 AC 系数的特点是，63 个系数中含有很多值为 0 的系数。因此，可以采用游程编码 RLC (Run Length Coding) 来更进一步降低数据的传输量。它的原理是，用一个符号值或串长代替具有相同值的连续符号，使符号长度少于原始数据的长度。只在各行或者各列数据的代码发生变化时，一次记录该代码及相同代码重复的个数，从而实现数据的压缩。在 JPEG 编码中，假设 RLC 编码之后得到了一个 (M,N) 的数据对，其中 M 是两个非零 AC 系数之间连续的 0 的个数 (即，行程长度)，N 是下一个非零的 AC 系数的值。采用这样的方式进

行表示，是因为 AC 系数当中原本就有大量的 0，而采用 Zigzag 扫描会使得 AC 系数中有很多连续的 0 的存在，如此一来，便非常适合于用 RLC 进行编码。

函数说明：

```
rlc_table = rlc(x)
```

功能：传入参数为 AC 系数构成的矩阵，输出为 RLC 编码后的码表。

实现原理：首先得到 AC 系数构成的矩阵的元素总个数为 num，接着初始化变量 zero_cnt 为 0，表示当前已经扫描到的 0 的个数，rlc_table 为空矩阵，然后进行主体部分的计算，伪代码如下所示：

```
for i = 1 → num do
    if ac[i] == 0 then
        zero_cnt ← zero_cnt + 1
    else
        rlc_table ← [rlc_table; [zero_cnt, ac[i]]]
        zero_cnt ← 0
    end if
end for
rlc_table ← [rlc_table; [0, 0]]
```

3.10 Huffman 编码

对于编码后的 DC 系数和 AC 系数，首先调用 MATLAB 内置的 `tabulate()` 函数统计其中各个值出现的频率，然后调用 MATLAB 内置的 `huffmandict()` 函数生成字典，接着调用 MATLAB 内置的 `huffmanenco()` 函数根据已有的字典将输入流进行编码。

理论知识：

Huffman 编码是可变字长编码(VLC)的一种，是由 Huffman 于 1952 年提出，它的基本思想是：对出现概率大的字符分配字符长度较短的二进制编码，对出现概率小的字符分配字符长度较长的二进制编码，从而使得字符的平均编码长度最短。它的实现方法如下所述：

1. 为每个符号建立一个叶子节点，并加上其相应的发生频。
2. 当有一个以上的节点存在时，进行下列循环：
 - a) 把这些节点作为带权值的二叉树的根节点，左右子树为。
 - b) 选择两棵根结点权值最小的树作为左右子树构造一棵新的二叉树，且至新的二叉树的根结点的权值为其左右子树上根结点的权值之和。
 - c) 把权值最小的两个根节点移。
 - d) 将新的二叉树加入队列中。
3. 最后剩下的节点即为根节点，此时 Huffman 编码的二叉树已经完成。
4. 通过遍历这颗二叉树，得到每个符号对应的二进制码流，并形成一张表。
5. 通过这张表将输入的符号流转换为对应的二进制码流，即完成 Huffman 编码。

由于 Huffman 编码是前缀码，故解码过程中不会有冲突，通过编码表即可将二进制码流重新转换为符号流，也即完成解码过程。

函数说明：

`tbl = tabulate(x)`

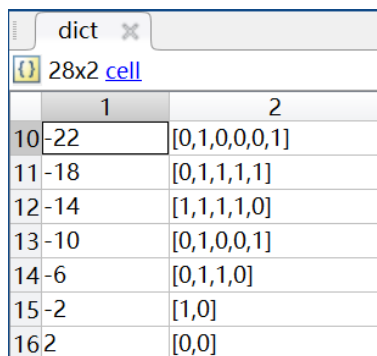
功能：显示向量 `x` 中数据的频率表。对于 `x` 中的每个唯一值，该函数返回一个统计其出现次数与出现频率的表格。使用效果如下图所示，生成表格的三列分别为值，出现次数和出现频率。

table			
28x3 double			
	1	2	3
1	-58	2	0.0488
2	-54	4	0.0977
3	-50	3	0.0732
4	-46	13	0.3174
5	-42	11	0.2686
6	-38	14	0.3418
7	-34	27	0.6592
8	-30	26	0.6348

`[dict,avglen] = huffmandict(symbols,prob)`

功能：为具有已知概率模型的数据源生成霍夫曼代码字典。输入 `symbols` 表

示符号列表，`prob` 表示每个符号出现频率的列表。输出 `avglen` 为字典的平均码字长度，`dict` 为生成的字典。输出 `dict` 的一个示例如下所示，第一列为符号的值，第二列为该符号对应的二进制码流。



	1	2
10-22		[0,1,0,0,0,1]
11-18		[0,1,1,1,1]
12-14		[1,1,1,1,0]
13-10		[0,1,0,0,1]
14-6		[0,1,1,0]
15-2		[1,0]
162		[0,0]

```
code = huffmanenco(sig,dict)
```

功能：使用输入字典 `dict` 所描述的 Huffman 编码方式对输入信号 `sig` 进行编码，返回其二进制码流 `code`。`sig` 可以具有矢量，单元格数组或字母数字单元格数组的形式。如果 `sig` 是单元格数组，则它必须是行或列。`dict` 是一个 $N \times 2$ 单元数组，其中 N 是要编码的不同可能符号的数量。`dict` 的第一列代表不同的符号，第二列代表相应的代码字。每个代码字都表示为行向量，并且 `dict` 中的任何代码字都不能是 `dict` 中任何其他代码字的前缀。可以使用 `huffmandict` 函数生成 `dict`。

3.11 保存数据

调用 MATLAB 内置的 `save()` 函数对图像压缩后的数据进行保存，需要保存的数据有：编码后 DC 系数，AC 系数各自三个分量的字典和码流，还有原始图像的长宽。

函数说明：

```
save(filename,variables)
```

功能：将当前工作区中 `variables` 指定的结构体数组的变量或字段保存在 MATLAB 格式的二进制文件（MAT 文件）`filename` 中，如果 `filename` 已存在，

`save` 会覆盖该文件。

3.12 解压流程概述

本次实验中的解压流程如下所示：

1. 读取压缩后的数据。
2. Huffman 解码得到 DPCM 编码后的 DC 系数和 RLC 编码后的 AC 系数
3. DPCM 解码得到 DC 系数。
4. RLC 解码得到 AC 系数。
5. AC 系数和 DC 系数重新组合后，进行逆 zigzag 变换。
6. 根据量化表进行反量化。
7. 逆 DCT 变换。
8. 重新生成 YCbCr 图像。
9. YCbCr 图像转换为 RGB 图像并显示。

3.13 读取压缩数据

调用 MATLAB 内置的 `load()` 函数实现对压缩数据的读取，

函数说明：

`load(filename)`

功能：将 MAT 文件中的变量加载到 MATLAB 工作区。

3.14 Huffman 解码

调用 MATLAB 内置的 `huffmandeco()` 函数实现 Huffman 解码。

函数说明：

`sig = huffmandeco(code,dict)`

功能：通过使用输入字典 `dict` 所描述的霍夫曼编码方式，对二进制码流 `code` 进行解码，并输出解码后的符号流 `sig`。

3.15 DPCM 解码

编写 `dpcm_decode()` 函数解码得到 DC 系数。

函数说明：

```
f_rec = dpcm_decode(en)
```

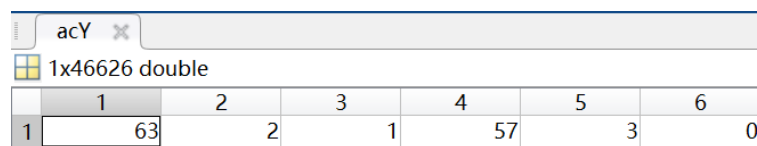
功能：传入参数为量化后的误差值 `en` (`en(1)` 为原始信号 `f(1)`，其余为误差值)，输出为 `dpcm` 解码后的 DC 系数。

实现原理：首先进行变量的初始化，获得 `en` 的长度为 `num`，重构信号 `f_rec`、预测信号 `f_pre` 初始化为长度为 `num` 的零矩阵。`f_pre(1)`，`f_pre(2)` 以及 `f_rec(1)` 的值均初始化为 `f(1)`。接着进行主体部分的计算，伪代码如下所示：

```
for i = 2 → num do
    if i ≠ 2 then
        f_pre[i] ← (f_rec[i - 1] + f_rec[i - 2])/2
    end if
    f_rec[i] ← f_pre[i] + en[i]
end for
```

3.16 RLC 解码

编写 `rlc_decode()` 函数解码得到 AC 系数。在 RLC 解码的过程，需要注意矩阵的维度和内容含义。直接 `huffman` 解码出来的 AC 系数(RLC 编码过的)是一个行向量，示例如下所示：



	1	2	3	4	5	6					
1	63	2	1	57	3	0					

这时最好是将这个行向量先转换为熟悉的 RLC 表的形式，最后解码并调整为 63 行即可还原 AC 系数。

函数说明：

```
out = rlc_decode(in,col_cnt)
```

功能：传入参数为 RLC 编码后的 AC 系数 `in`，以及经历了 `dct` 变换和 `zigzag`

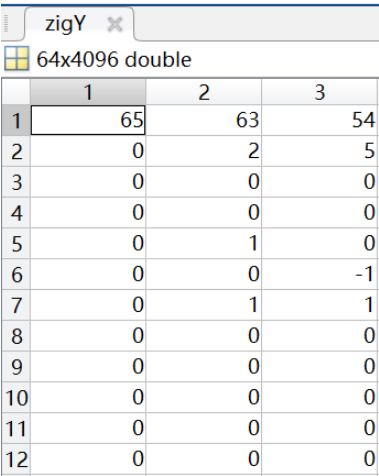
转换后矩阵的列数 `col_cnt`，输出为 RLC 解码后的 AC 系数 `out`。

实现原理：设输入 `in` 的长度为 `col`，首先将输入转换为熟悉的 RLC 表的形式，这样 RLC 表就有 `col/2` 行，再设 `ac_vec` 为长度为 $63 \times \text{col_cnt}$ 的向量，接着按如下所示的伪代码进行操作即可解码得到 AC 系数：

```
for i = 1 → cnt/2 do
  if rlc_table[i, 1] == 0 then
    ac_vec[j] ← rlc_table[i, 2]
    j ← j + 1
  else
    for k = 1 → rlc_table[i, 1] do
      ac_vec[j] ← 0
      j ← j + 1
    end for
    ac_vec[j] ← rlc_table[i, 2]
    j ← j + 1
  end if
end for
out ← reshape(ac_vec, 63, col_cnt)
```

3.17 逆 zigzag

编写 `inv_zigzag ()` 函数进行逆 zigzag 变换重新得到 DCT 变换后的量化矩阵。这个过程中需要注意到的细节是，zigzag 变换后的矩阵中，每一列代表了原图像中的一个 8×8 的块（如下图中的示例所示），此外还需要知道原图的宽度来计算原图的一行有多少块，这样分别对每一列进行逆 zigzag 变换之后重新组合这些块，就可以得到原始 DCT 系数矩阵了。

A screenshot of a MATLAB variable viewer window titled 'zigY'. It shows a 64x4096 double matrix. The first three columns are expanded to show the first 12 rows. The values are as follows:

	1	2	3
1	65	63	54
2	0	2	5
3	0	0	0
4	0	0	0
5	0	1	0
6	0	0	-1
7	0	1	1
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0

函数说明：

```
invzigx = inv_zigzag(x,resize_width)
```

功能：传入参数为需要进行 zigzag 逆变换的矩阵 x ，以及原图像的宽度 $resize_width$ ，输出为 zigzag 逆变换后的 DCT 系数矩阵 $invzigx$ 。

实现原理：与 zigzag 变换的思路很相似，仍然使用直接生成的那张 zigzag 变换表，对传入矩阵的每一列，也即一个长度为 64 的向量，通过查表将其重新填入一个 8×8 的矩阵中，最后将他们合并即可。

3.18 反量化

编写 `inv_quantify()` 函数进行反量化重新得到 DCT 系数矩阵。

函数说明：

```
[y,cb,cr] = inv_quantify(qy,qcb,qcr)
```

功能：传入参数为需要进行反量化的矩阵 qy ， qcb ， qcr ，输出为反量化后的三个通道的 DCT 系数矩阵 y ， cb ， cr 。

实现原理：对输入的 qy ，每个 8×8 小块中的值与亮度量化表中的值对应相乘；对输入的 qcb ， qcr ，每个 8×8 小块中的值与色度量化表中的值对应相乘，将处理后的三个矩阵返回即可。

3.19 逆 DCT 变换

调用 MATLAB 内置的 `idct2()` 函数以及 `blockproc()` 函数实现对图像的分块逆 DCT 变换处理。相对于 DCT 变换来说，仅将 `blockproc()` 中用于块处理的函数从 `dct2()` 换成了 `idct2()`，其他处理不变。

理论知识：

在图像的接收端，根据 DCT 变化的可逆性，我们可以通过 DCT 反变换恢复出原始的图像信息，其公式如下：

$$f(i, j) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} c(u)c(v) F(u, v) \cos\left[\frac{(i+0.5)\pi}{N}u\right] \cos\left[\frac{(j+0.5)\pi}{N}v\right]$$

$$c(u) = \begin{cases} \sqrt{\frac{1}{N}}, & u = 0 \\ \sqrt{\frac{2}{N}}, & u \neq 0 \end{cases}$$

同样的道理，我们利用之前的矩阵运算公式可以推导出 DCT 反变换相应的矩阵形式：

$$F = AfA^T$$

$$\therefore A^{-1} = A^T$$

$$\therefore f = A^{-1}F(A^T)^{-1} = A^TFA$$

3.20 重新生成 RGB 图像

编写 `regenerate_Ycbcr()` 函数利用颜色下采样后的结果重新生成 YCbCr 图像。再调用 MATLAB 内置的 `ycbcr2rgb()` 函数将 YCbCr 图像转换为 RGB 图像，之后调用内置的 `imshow()` 函数直接显示图像或者调用 `imwrite()` 函数对恢复的图像进行保存。

函数说明：

`down_imgYCbCr = regenerate_Ycbcr(imgY, imgCb, imgCr)`

功能：传入参数为经过了颜色下采样后的三个通道的矩阵 `imgY`，`imgCb`，`imgCr`，输出为重新生成的 YCbCr 图像矩阵 `down_imgYCbCr`。

实现原理：由于颜色下采样的过程中对 Y 通道没有修改，所以 `imgY` 保持不变即可；对于 Cb 和 Cr 通道，其每一个值都要填充重构图像的每个 2×2 小块的 CbCr 值，最后将这三个通道重新合并并返回即可。

`RGB = ycbcr2rgb(YCBCR)`




功能：将 YCbCr 图像的亮度 (Y) 和色度 (Cb 和 Cr) 值 转换为 RGB 图像

的红色、绿色和蓝色值。

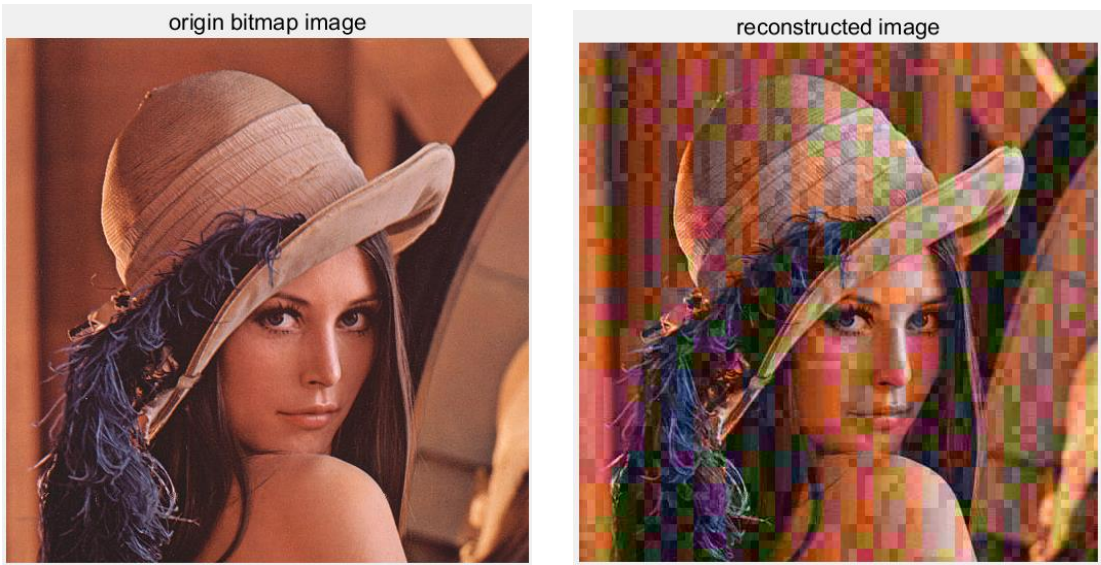
4. Experiment Results

4.1 原始实验结果

我选取的需要压缩的目标图片是一个 509×508 的 BMP 图片（大小为 757KB）。完整执行我编写的压缩算法后保存的文件大小为 34.1KB，压缩比为 $757\text{KB} / 34.1\text{KB} = 22.2$ 。

	<input type="text" value="origin_lena"/>		<input type="text" value="compressed_image"/>
文件类型:	BMP 文件 (.bmp)	文件类型:	MATLAB Data (.mat)
打开方式:	照片 <input type="button" value="更改(C)..."/>	打开方式:	 MATLAB R2019a <input type="button" value="更改(C)..."/>
位置:	C:\Users\86525\Desktop\作业\大三(下)\多媒体技术\pr	位置:	C:\Users\86525\Desktop\作业\大三(下)\多媒体技术\pr
大小:	757 KB (775,770 字节)	大小:	34.1 KB (34,941 字节)

原图像如左下图所示，压缩并解压后的图像如右下所示。



可以很明显的看到，尽管压缩比还不错，但压缩后解压出来的效果却不尽人意。仔细观察一下解压出来的图像，可以看到整个图像有很多比较明显的正方形小色块，并且整体的颜色比较凌乱，有点像坏掉的彩色电视机。

4.2 原始实验结果分析与改进

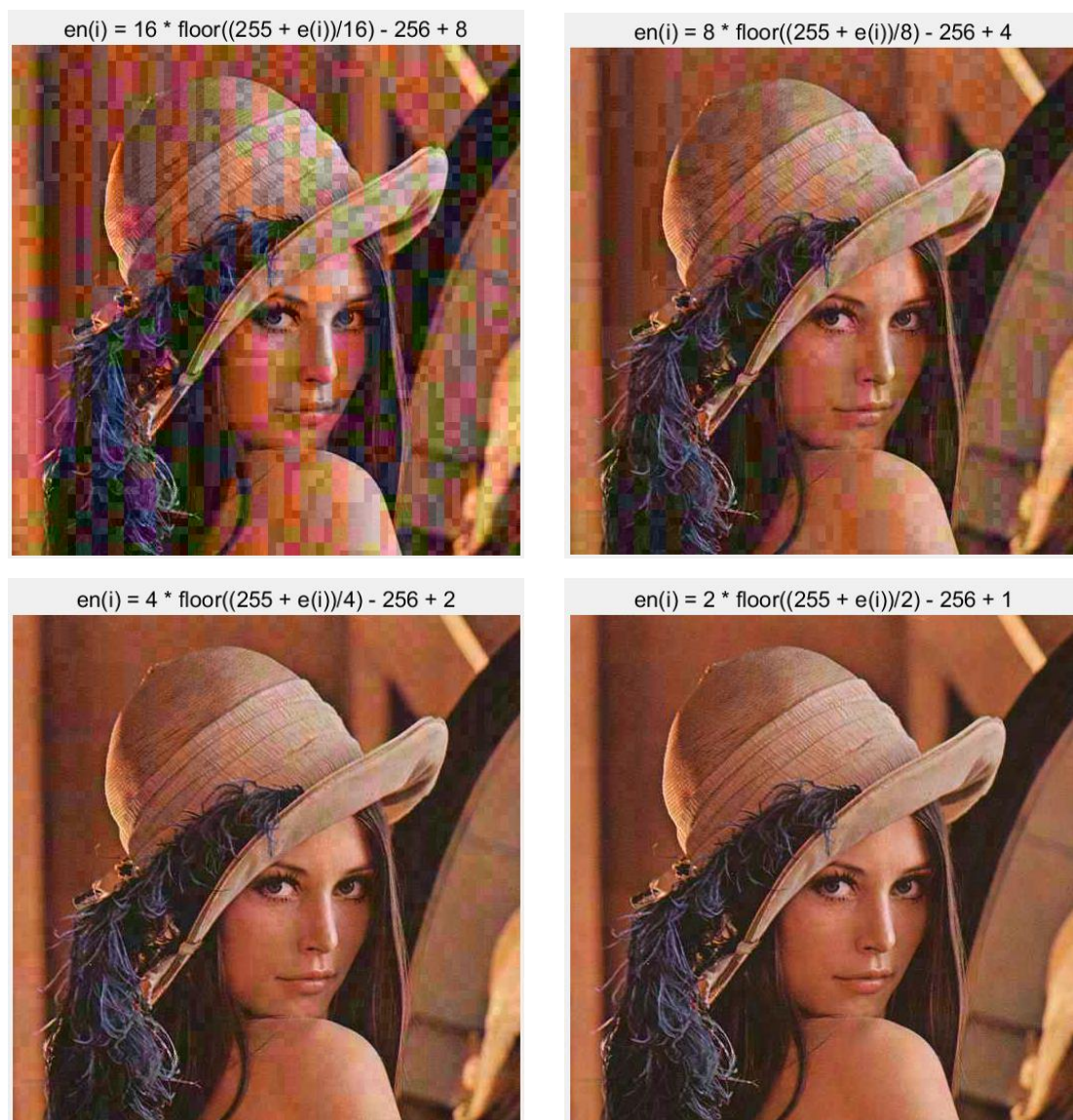
起初我猜测是压缩解压过程中的某个算法实现错了导致的，但是在一一验证过后发现算法实现上没有问题。接着我想到可以用排除法，毕竟整个压缩过程中就只有三步有信息的损失（颜色下采样，量化，DPCM），于是我导出了一个仅进行了下采样和量化后的恢复图像（如左下图所示），与原图比较（右下图）发现这时的效果还是很好的，那基本可以确定问题出在 DPCM 上。



在对 DC 系数进行 DPCM 编码的过程中，信息的损失发生在对误差值 $e(i)$ 进行量化的过程中，也即下图用红框标出的部分。

```
for  $i = 2 \rightarrow num$  do
  if  $i \neq 2$  then
     $f\_pre[i] \leftarrow (f\_rec[i - 1] + f\_rec[i - 2])/2$ 
  end if
   $e[i] \leftarrow f[i] - f\_pre[i]$ 
   $en[i] \leftarrow 16 * floor((255 + e[i])/16) - 256 + 8$ 
   $f\_rec[i] \leftarrow f\_pre[i] + en[i]$ 
end for
```

如果将这个量化的幅度缩小，是不是就能使得解压后的图像有较好的质量呢？我对这个量化幅度不断缩小，有了以下的实验结果：



可以看到随着量化幅度的降低，图像质量有明显的提升，那我们再来看看这四张图片对应的压缩文件大小：

大小:	34.1 KB (34,941 字节)	大小:	34.8 KB (35,687 字节)
大小:	35.8 KB (36,711 字节)	大小:	37.1 KB (37,998 字节)

我们可以发现对 DC 系数进行量化会对图像质量有较大的影响，但压缩比并没有得到显著的提升。将上面四张图片的压缩文件分别读入 **MATLAB** 的工作区后观察（如下面四张图所示），可以看到在刚刚修改量化函数的过程中，只是下面黑框的部分，也就是 DC 系数发生了变化，而真正占用了很大数据量的是红框的部分，也就是 AC 系数。

acCb_comp	1x7402 double
acCb_dict	84x2 cell
acCr_comp	1x8448 double
acCr_dict	78x2 cell
acY_comp	1x170801 double
acY_dict	170x2 cell
dcCb_comp	1x2853 double
dcCb_dict	14x2 cell
dcCr_comp	1x3041 double
dcCr_dict	16x2 cell
dcY_comp	1x17710 double
dcY_dict	53x2 cell
origin_height	509
origin_width	508

acCb_comp	1x7402 double
acCb_dict	84x2 cell
acCr_comp	1x8448 double
acCr_dict	78x2 cell
acY_comp	1x170801 double
acY_dict	170x2 cell
dcCb_comp	1x2073 double
dcCb_dict	9x2 cell
dcCr_comp	1x2224 double
dcCr_dict	9x2 cell
dcY_comp	1x13756 double
dcY_dict	28x2 cell
origin_height	509
origin_width	508

acCb_comp	1x7402 double
acCb_dict	84x2 cell
acCr_comp	1x8448 double
acCr_dict	78x2 cell
acY_comp	1x170801 double
acY_dict	170x2 cell
dcCb_comp	1x1633 double
dcCb_dict	5x2 cell
dcCr_comp	1x1663 double
dcCr_dict	6x2 cell
dcY_comp	1x10403 double
dcY_dict	15x2 cell
origin_height	509
origin_width	508

acCb_comp	1x7402 double
acCb_dict	84x2 cell
acCr_comp	1x8448 double
acCr_dict	78x2 cell
acY_comp	1x170801 double
acY_dict	170x2 cell
dcCb_comp	1x1537 double
dcCb_dict	4x2 cell
dcCr_comp	1x1548 double
dcCr_dict	5x2 cell
dcY_comp	1x7586 double
dcY_dict	8x2 cell
origin_height	509
origin_width	508

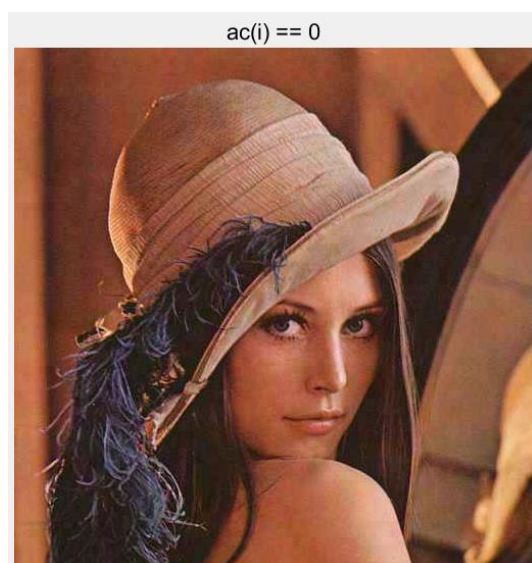
所以我的想法是 DC 系数量化幅度小一点保证图像质量，再从 AC 系数入手增大图像的压缩比。通常的做法是在量化阶段对量化表乘上一定的系数来改变压缩比，我这里选择对 RLC 算法稍作修改，使之成为有损的，通过进一步量化 AC 系数来提高压缩比。我的修改也很简单，把很接近 0 的 AC 系数当做 0 来处理就可以实现，也即修改下面的红框所示的部分。

```

for  $i = 1 \rightarrow num$  do
  if  $ac[i] == 0$  then
     $zero\_cnt \leftarrow zero\_cnt + 1$ 
  else
     $rlc\_table \leftarrow [rlc\_table; [zero\_cnt, ac[i]]]$ 
     $zero\_cnt \leftarrow 0$ 
  end if
end for
 $rlc\_table \leftarrow [rlc\_table; [0, 0]]$ 

```

修改后的实验结果如下所示：



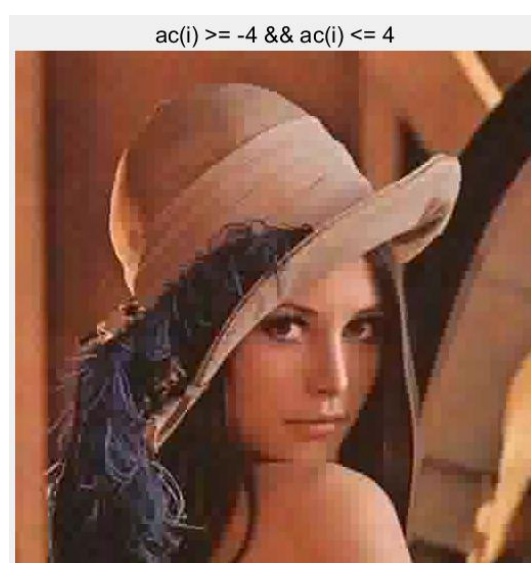
大小: 37.1 KB (37,998 字节)



大小: 20.9 KB (21,429 字节)



大小: 15.7 KB (16,118 字节)



大小: 11.3 KB (11,642 字节)

可以看到，经过这样的修改之后，图像压缩比的提升是非常显著的，且图像质量下降的并不快，只是分块模糊的现象越来越明显。

此外还可以注意到压缩文件分别读入 MATLAB 的工作区后，MATLAB 内置的 `huffmanenco()` 函数对输入编码后的码流是对于每一个位单独用一个 `double` 变量来存储的（如左下图所示），这样其实浪费了很大一部分空间。将其转换为 `uint8` 的格式来存储就可以节省一部分空间（如右下图所示）。

acCb_comp	1x7402 double
acCb_dict	84x2 cell
acCr_comp	1x8448 double
acCr_dict	78x2 cell
acY_comp	1x170801 double
acY_dict	170x2 cell
dcCb_comp	1x2853 double
dcCb_dict	14x2 cell
dcCr_comp	1x3041 double
dcCr_dict	16x2 cell
dcY_comp	1x17710 double
dcY_dict	53x2 cell
origin_height	509
origin_width	508

acCb_comp	1x1594 uint8
acCb_dict	60x2 cell
acCr_comp	1x2291 uint8
acCr_dict	69x2 cell
acY_comp	1x75366 uint8
acY_dict	242x2 cell
dcCb_comp	1x2853 uint8
dcCb_dict	14x2 cell
dcCr_comp	1x3041 uint8
dcCr_dict	16x2 cell
dcY_comp	1x17710 uint8
dcY_dict	53x2 cell
origin_height	509
origin_width	508

4.3 最终实验结果

选取了三张质量较高的壁纸图片进行压缩和解压，其实验结果如下所示。



图 1 原始图像 A



图 2 原始图像 A 的重构图像



图 3 原始图像 B



图 4 原始图像 B 的重构图像



图 5 原始图像 C



图 6 原始图像 C 的重构图像

A、B、C 三张图像的压缩比分别为

$$\text{A: } 6.31\text{MB} / 98.1\text{KB} = 65.9$$

$$\text{B: } 10.5\text{MB} / 80.0\text{KB} = 134.4。$$

$$\text{C: } 25.7\text{MB} / 371.0\text{KB} = 70.9。$$

可以看到在压缩比很大的前提下，压缩后的图片仍然保有较高的质量。图像

B 的压缩比很大是因为其中有大片接近纯色的区域。

4.4 实验总结

本次实验的过程中，我利用 MATLAB 作为开发工具，参照标准 JPEG 压缩流程一步步编写代码进行实现，最后通过对实验结果的分析，对标准的 JPEG 压缩流程进行了一些修改，完成了整个压缩算法的实现。在这个过程中，我对课上讲到的有关 JPEG 压缩的理论知识有了更加深入的了解，对于 MATLAB 工具的使用也从完全没有接触过到比较熟练，总的来说还是十分有收获的。我在整个过程中的心得主要有以下几点：

- MATLAB 在图像处理和矩阵运算上十分方便，但也要注意一些细节问题，比如之前提到的 `reshape()` 函数是按列读取和存取的。
- 在图像压缩的流程中要多关注数据的维度，否则容易出错，还有就是最好每做完一步就提前验证一下，便于找 bug。
- DCT 变换后的图像数据尽量减小对 DC 系数的量化程度来保证图像质量，可以适当提高对 AC 系数的量化程度来提高压缩比。
- 有损压缩首先要做的事情就是“把重要的信息和不重要的信息分开”，颜色空间转换，DCT 变换以及 zigzag 都是为区分重要信息和非重要信息服务的。

References:

- [1] Fundamentals of Multimedia (2rd edition)
- [2] <https://ww2.mathworks.cn/help/> (MathWorks 官网)
- [3] <https://blog.csdn.net/xueyushenzhou/article/details/40817949> (色度抽样 (4:2:0) 到底是什么意思?)
- [4] <https://blog.csdn.net/neverever01/article/details/103430474> (实现 zigzag 扫描 (z 字形扫描))
- [5] <https://www.cnblogs.com/buaaxhzh/p/9138307.html> (JPEG 图像压缩算法流程详解)

|