

## JPEG 静止图像压缩方法

JPEG (Joint Photographic Expert Group) 是 JPEG 标准的产物, 该标准由国际标准化组织 (ISO) 制定, 是面向连续色调静止图像的一种压缩标准。JPEG 属于有损压缩方法, 它能够将图像压缩在很小的储存空间, 一定程度上会造成图像数据的损伤, 尤其是使用过高的压缩比例, 将使解压缩后恢复的图像质量降低。JPEG 格式的压缩率是目前各种图像文件格式中最高的, 它用有损压缩的方式去除图像的冗余数据, 由于其高效的压缩效率和标准化要求, 目前已广泛用于彩色传真、静止图像、电话会议、印刷及新闻图片的传送。由于各种浏览器都支持 JPEG 这种图像格式, 因此它也被广泛用于图像预览和制作 HTML 网页。

### 1. 总体方案

本次实验使用的测试图像均为三通道的 RGB 图像, 因此将分为以下几部分对图像进行压缩和解压缩处理: (1) 色彩空间转换; (2) 对 U、V 分量采样; (3) 图像分块; (4) 离散余弦变换; (5) 数据量化; (6) 编码; (7) 解码与其他反变换。

#### 1.1 色彩空间转换

将图像由 RGB 色彩空间转换到 YUV 色彩空间, 变换矩阵与反变换矩阵如下。

$$T_{rgb2yuv} = \begin{bmatrix} \frac{77}{256} & \frac{150}{256} & \frac{29}{256} \\ \frac{44}{256} & \frac{87}{256} & \frac{131}{256} \\ -\frac{256}{256} & -\frac{256}{256} & \frac{256}{256} \\ \frac{131}{256} & \frac{110}{256} & \frac{21}{256} \\ \frac{256}{256} & -\frac{256}{256} & -\frac{256}{256} \end{bmatrix} \quad (1.1)$$

$$T_{yuv2rgb} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.3441 & -0.7141 \\ 1 & 1.1772 & 0 \end{bmatrix} \quad (1.2)$$

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = T_{rgb2yuv} * \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \quad (1.3)$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = T_{yuv2rgb} * \begin{bmatrix} Y \\ U \\ V \end{bmatrix} - \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \quad (1.4)$$

#### 1.2 对 U、V 分量采样

研究发现, 人眼对亮度变化的敏感度比色彩变化的敏感度高。因此可以认为 Y 分量比 U、V 分量更为重要, 故通常会对 U、V 分量采样, 本次实验采用  $Y:U:V = 4:2:2$  的采样比例。这样采样虽然失去了一定的精度, 但是也在人眼几乎不可见的前提下减小了数据储存量。

#### 1.3 图像分块

在第四步的离散余弦变换中, 是对  $8 \times 8$  的子块进行处理的, 因此首先需要对图像进行分块。由于不能保证图像刚好能被分成若干个  $8 \times 8$  的子块, 所以需要将图像的长宽补齐到 8 的倍数。对于 Y 分量, 补充的区域补 0; 对于 U、V 分量, 补充的区域补为 128。

#### 1.4 离散余弦变换

离散余弦变换的公式为

$$X(k, l) = \frac{2}{\sqrt{MN}} c(k)c(l) \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(m, n) \cos \frac{(2m+1)k\pi}{2M} \cos \frac{(2n+1)l\pi}{2N} \quad (1.5)$$

$$c(k) = \begin{cases} \frac{1}{\sqrt{2}}, & k = 0 \\ 1, & k = 1, 2, \dots, M-1 \end{cases} \quad (1.6)$$

由于已经明确了每次进行的离散余弦变换的矩阵大小均为  $8 \times 8$ ，因此利用离散余弦变换矩阵实现离散余弦变换以降低运算量，离散余弦变换矩阵的计算公式为

$$T_{ij} = \begin{cases} \frac{1}{\sqrt{M}} & i = 0, 0 \leq j \leq M-1 \\ \sqrt{\frac{2}{M}} \cos \frac{\pi(2j+1)i}{2M} & 1 \leq i \leq M-1, 0 \leq j \leq M-1 \end{cases} \quad (1.7)$$

离散余弦变换便可简化为  $T_{ij} * B * T_{ij}^T$ ，其中 B 为  $8 \times 8$  的矩阵。

### 1.5 数据量化

JPEG 提供的量化算法如下

$$c(k, l) = INT \left[ \frac{X(k, l)}{Q(k, l)} + \frac{1}{2} \right] (k, l = 0, 1, \dots, 7) \quad (1.8)$$

其中量化矩阵  $Q$  分为亮度量化矩阵  $Q_L$  与色差量化矩阵  $Q_C$ 。

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

图 1-1 亮度量化表

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

图 1-2 色差量化表

### 1.6 编码

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

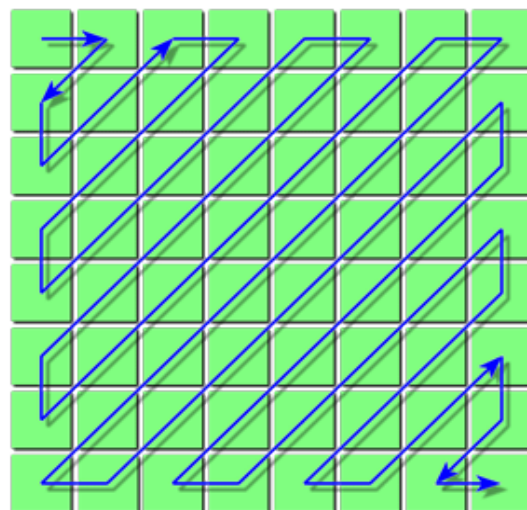


图 1-3 Z 字扫描顺序

图 1-3 中数字为 Z 字扫描的顺序。矩阵左上角的值，被称为直流分量 DC，其余 63 个值被称为交流分量 AC。DC 与前一矩阵的 DC 系数进行差分编码（Differential Pulse Code Modulation, DPCM），AC 系数则使用游程长度编码（Run Length Encoding, RLE）。

本次实验使用的哈夫曼编码表来源自吴乐南编著《数据压缩（第三版）》，P55 表 4.2，P62 表 4.8，P63 表 4.9，P64 表 4.10，这四张表在此依次引为表 1-1，表 1-2，表 1-3 和表 1-4。

### 1.7 解码与其他反变换

解码与其他变换均为上述方法的反变换，本报告后面也只介绍编码的算法，解码算法不再赘述。

## 2. 压缩原理及算法描述

### 2.1 压缩原理

JPEG 的基本结构如下图所示

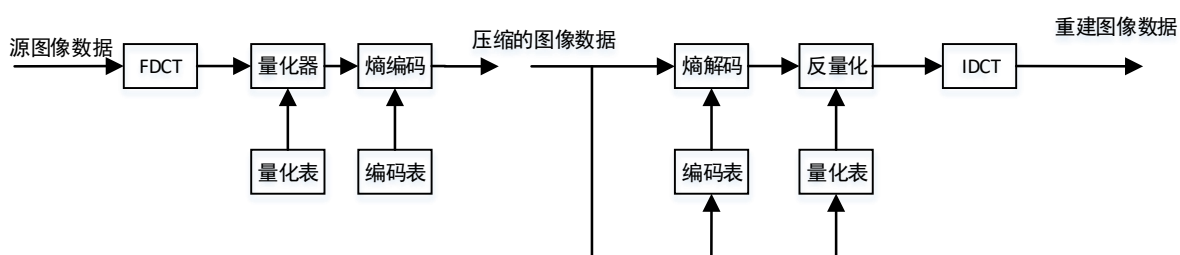


图 2-1JPEG 基本结构

JPEG 对图像的第一次压缩是在 U、V 分量的采样上。由于人眼对色彩信息较于亮度信息更不敏感，所以这里通过采样舍弃了一些人眼不能分辨的色彩信息实现数据量的降低。

JPEG 对图像的第二次压缩是利用离散余弦变换。DCT 有着非常优秀的频域能量集中性能，即它能把图像重要的信息聚集在一起，对于不重要的频率区域和系数就能够直接抛弃掉，在很大程度上降低了数据的规模。

JPEG 对图像的第三次压缩是在量化、编码过程中。图像源数据在经过 DCT 后，重

要系数都集中在了矩阵的左上角区域，但是此时它们还都是浮点数，不便于编码，因此需要首先对矩阵系数进行量化。本次实验中使用亮度和色度的水平样本数为2:1、各样本均为8位的亮度量化表和色差量化表。经过量化后的矩阵系数包含很多0，在使用Z字扫描将矩阵拉成向量之后，向量中便存在许多连续的0，所以使用游程长度编码方法将其编码成“NNNN/SSSS+尾码”的形式，最后通过查哈夫曼编码表将其表达为相应的码字。

JPEG对图像的第四次压缩是将Z字扫描结果分为直流分量DC和交流分量AC，再对其用不同方法进行编码。经过DCT后，矩阵重要系数集中在左上角区域，这就导致最左上角位置的数字较大，为了降低其编码的码字长度，使其与前一矩阵的DC系数进行差分编码，对于差分值DIFF采用将码字截断为“SSSS+尾码”的方法，对码表进行了简化。

## 2.2 算法描述

图2-2展示了本次实验算法流程框图。

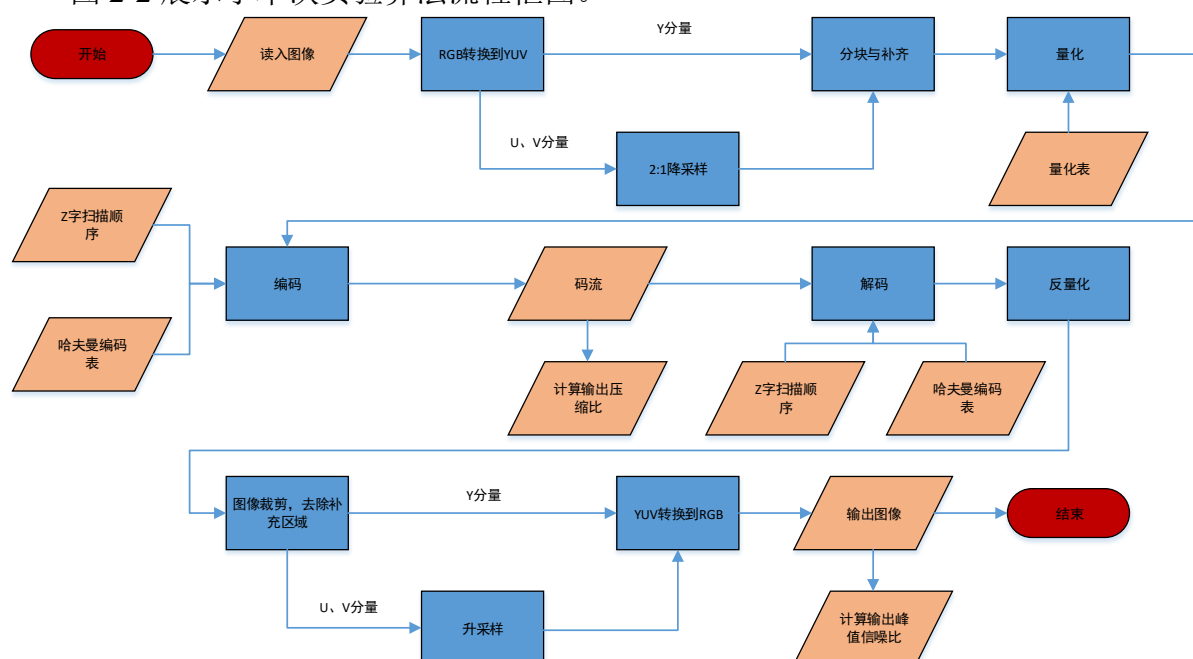


图 2-2 算法流程图

本次实验中将前文提到的表1-1到表1-4作为常量数组保存并使用。

Z字扫描顺序使用深度优先搜索，即递归方法获得。分析Z字扫描规律，可以发现它仅有四种基本方向选择，即右，左下，下和右上，这里把四种方向依次用数字1~4表示。在抵达扫描半程之前扫描方向始终保持(1,2,3,4)的循环，在半程时，突变发生在方向2之后，随后保持(1,4,3,2)的循环直到到达右下角，可以看出突变后的顺序是原先规律的反向。因此函数设置为 $dfs\_z(x,y,dir,mode,cnt)$ ，其中 $(x,y)$ 为当前位置， $dir$ 为下次前进的方向， $mode$ 为是否发生突变标志， $cnt$ 为计数器，在递归到 $cnt = 64$ 时退出，在回溯过程中生成Z字扫描结果并保存，递归起点为 $dfs\_z(1,1,1,1,1)$ 。需要注意的一点是，递归途中存在需要修改下次前进方向的情况，这里认为只要下次到达的位置仍在矩阵边界上就修改方向。

实验中使用的YUV采样比例为4:2:2，因此在降采样时对U、V隔行采样。

编码过程使用Z字扫描得到图像块向量后，首先对DC分量进行编码，DC与前一矩阵DC系数的差值记为DIFF，查表编码，将其编码为“SSSS+尾码”。如图2-3框图

所示，依次扫描其余 63 个 AC 系数，当扫描到 0 时跳过，直到扫描到一个不为 0 的数，此时开始查表编码，利用记录的本次不为 0 数的位置和上次不为 0 数的位置便可以得到“NNNN/SSSS+尾码”的码字。需要注意的是，若连续 0 的长度大于 16 则需要编码为 ZRL，剩余再单独编码；若直到结束均为连续的 0，则编码为 EOB；框图右上角部分是为了处理第 64 个数不为 0 的情况。

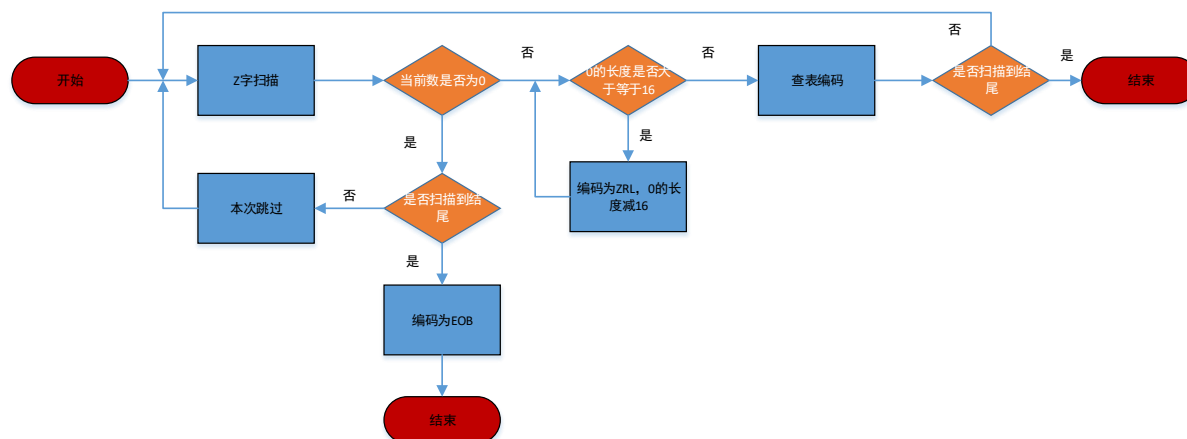


图 2-3 AC 系数编码算法框图

### 3. 程序及注释

```

clearvars
% I_in = double(imread('lena.bmp'));
I_in = double(imread('风景.bmp'));
[m, n, dim] = size(I_in);
figure
imshow(uint8(I_in))
Get_Constant_Matrix(); %生成字典和变换系数
load constant.mat
%% 初始化、色彩空间转换、4:2:2 降采样
Y = zeros(m, n);
U = zeros(m, n);
V = zeros(m, n);
for i = 1 : m
    for j = 1 : n
        tmp = [I_in(i, j, 1); I_in(i, j, 2); I_in(i, j, 3)];
        Y(i, j) = T_rgb2yuv(1, :) * tmp;
        U(i, j) = T_rgb2yuv(2, :) * tmp;
        V(i, j) = T_rgb2yuv(3, :) * tmp;
    end
end
U = U + 128;
V = V + 128;
U_d = Downsampling(U);
  
```

---

```

V_d = Downsampling(V);
%% 生成 z 字扫描坐标顺序
global mode_dir mapping
x = 1 : 10;
mapping = 2.^x - 1;
mode_dir = [ 0, 1; %→
            1, -1; %↖
            1, 0; %↓
            -1, 1]; %↗
dfs_z(1, 1, 1, 1, 1); %dfs_z(x, y, dir, mode, cnt) 当前位置(x,y),下次方向
dir, 方向模式 mode, mode = ±1, 计数器 cnt, 结果保存在全局变量 index 中
%% 编码
[Y_Q, Y_code, Y_cnt, Y_len] = Block_Encode(Y, Q_L, Dictionary_L,
Dictionary_DC(1, :), 0);
[U_Q, U_code, U_cnt, U_len] = Block_Encode(U_d, Q_C, Dictionary_C,
Dictionary_DC(2, :), 1);
[V_Q, V_code, V_cnt, V_len] = Block_Encode(V_d, Q_C, Dictionary_C,
Dictionary_DC(2, :), 1);
CR = (m * n * 8 * 3) / (Y_len + U_len + V_len);
disp(sprintf("The Compression Rate CR = %s", num2str(CR))) %#ok<DSPS>

%% 解码
[Y_decode] = Block_Decode(Y_code, Y_cnt, Q_L, Dictionary_L,
Dictionary_DC(1, :), size(Y));
[U_decode] = Block_Decode(U_code, U_cnt, Q_C, Dictionary_C,
Dictionary_DC(2, :), size(U_d));
[V_decode] = Block_Decode(V_code, V_cnt, Q_C, Dictionary_C,
Dictionary_DC(2, :), size(V_d));
%% 升采样, 色彩空间转换
% U_reconstruct = upsampling(U_decode);
% V_reconstruct = upsampling(V_decode);
U_reconstruct = imresize(U_decode, size(U), 'bilinear');
V_reconstruct = imresize(V_decode, size(V), 'bilinear');
U_reconstruct = U_reconstruct - 128;
V_reconstruct = V_reconstruct - 128;
for i = 1 : m
    for j = 1 : n
        tmp = [Y_decode(i, j); U_reconstruct(i, j); V_reconstruct(i, j)];
        I_out(i, j, 1) = T_yuv2rgb(1, :) * tmp;
        I_out(i, j, 2) = T_yuv2rgb(2, :) * tmp;
        I_out(i, j, 3) = T_yuv2rgb(3, :) * tmp;
    end
end
end
figure

```

---

```

imshow(uint8(I_out))
MSE = sum(sum(sum((I_out - I_in) .^ 2))) / (m * n * 3);
PSNR = 10 * log10(255 ^ 2 / MSE);
disp(sprintf("The Peak Signal-to-Noise Ratio PSNR = %s",
num2str(PSNR))) %#ok<DSEPS>

%% 函数部分
function Img_out = Downsampling(Img_in)
    [m, n] = size(Img_in);
    Img_out = zeros(floor(m / 2), n);
    for i = 1 : floor(m / 2)
        Img_out(i, :) = Img_in(i * 2 - 1, :); %采样 1 3 5 7 9 ... 行
    end
end

function dfs_z(x, y, dir, mode, cnt)
    global mode_dir index
    index(cnt, 1) = x;
    index(cnt, 2) = y;
    if x == 8 && y == 8
        return;
    end
    dir_new = dir;
    if x == 8 && y == 1
        mode = -1;
        dir = 1;
        dir_new = 4;
    end
    if judge(x, y, mode_dir(dir, 1), mode_dir(dir, 2)) %只要下次的位置在边界上
就修改方向
        if mode == 1
            dir_new = mod(dir, 4) + 1;
        else
            dir_new = dir - 1;
            if dir_new == 0
                dir_new = 4;
            end
        end
    end
    dfs_z(x + mode_dir(dir, 1), y + mode_dir(dir, 2), dir_new, mode, cnt +
1);
end

function flag = judge(x, y, dx, dy)
    nx = x + dx;
    ny = y + dy;

```

---

```

    if nx == 1 || nx == 8 || ny == 1 || ny == 8
        flag = 1;
    else
        flag = 0;
    end
end
function [Quantization_out, code, CNT, LENGTH] = Block_Encode(Img_in, Q,
Dictionary, Dictionary_DC, flag)
    %% 凑齐 8*8 子块, 扩充区域补 0
    [m, n] = size(Img_in);
    m_new = ceil(m / 8) * 8;
    n_new = ceil(n / 8) * 8;
    if flag
        Quantization_out = 128 * ones(m_new, n_new);
    else
        Quantization_out = zeros(m_new, n_new);
    end
    Quantization_out(1 : m, 1 : n) = Img_in;
    %% 8*8 量化
    DC = 0;
    LENGTH = 0;
    code = {};
    CNT = 0;
    for i = 1 : 8 : m_new - 7
        for j = 1 : 8 : n_new - 7
            block = Quantization_out(i : i + 7, j : j + 7);
            block = dct2(block);
            block = floor(block ./ Q + 0.5);
            [code_ij, cnt, len] = compress(block, Dictionary, Dictionary_DC,
DC); %len 表示压缩后码流的长度
            for k = 1 : cnt
                code{1, CNT + k} = code_ij{1, k};
                code{2, CNT + k} = code_ij{2, k};
            end
            CNT = CNT + k;
            LENGTH = LENGTH + len;
            DC = block(1, 1);
        end
    end
end
function [code, cnt, len] = compress(block, Dictionary, Dictionary_DC, DC)
    global index mapping
    [m, ~] = size(index);
    vec = zeros(1, m);

```



---

```

pre = 1;%上一个非 0 位置
pos = 2;%当前位置，第一个位置是 DC 系数
code = cell(2, m);
cnt = 0;%码的长度
len = 0;
for i = 1 : m
    vec(i) = block(index(i, 1), index(i, 2));
end
%% DC 系数
DIFF = vec(1) - DC;
SSSS = ceil(log2(abs(DIFF) + 1));
cnt = cnt + 1;
code{1, cnt} = Dictionary_DC(SSSS + 1);
if DIFF <= 0
    flag = 0;
else
    flag = 1;
end
if ~flag
    if DIFF ~= 0
        code{2, cnt} = dec2bin(bitxor(abs(DIFF), mapping(SSSS)), SSSS);
    else
        code{2, cnt} = dec2bin(0);
    end
else
    code{2, cnt} = dec2bin(DIFF, SSSS);
end
len = len + strlength(code{1, cnt}) + length(code{2, cnt});
%% AC 系数
while 1
    if (pos == m + 1) && (vec(pos - 1) == 0)
        cnt = cnt + 1;
        code{1, cnt} = Dictionary(17, 1);%编码为 EOB
        code{2, cnt} = '';
        len = len + strlength(code{1, cnt}) + length(code{2, cnt});
        break;
    end
    if pos > m
        break;
    end
    if vec(pos) == 0
        pos = pos + 1;
        continue;
    end
end

```

---

```

NNNN = pos - pre;%NNNN - 1 对应 0 游程长度, NNNN 是在字典中的行号
if NNNN >= 17 %NNNN - 1 >= 16 处理中间连续 0 的个数大于 16 的情况, 每 16 个编
为 ZRL, 剩余少于 16 个的补在当前非零值前面编码
    rep = floor((NNNN - 1) / 16);
    for i = 1 : rep
        cnt = cnt + 1;
        code{1, cnt} = Dictionary(17, 2);%编码为 ZRL
        code{2, cnt} = '';
        len = len + strlen(code{1, cnt}) + length(code{2, cnt});
    end
    NNNN = NNNN - 16 * rep;
end
pre = pos;
if vec(pos) < 0
    SSSS = ceil(log2(-vec(pos) + 1));
    flag = 0;
else
    SSSS = ceil(log2(vec(pos) + 1));
    flag = 1;%正数标记为 1, 负数标记为 0, 便于取反
end
cnt = cnt + 1;
code{1, cnt} = Dictionary(NNNN, SSSS);
if ~flag
    code{2, cnt} = dec2bin(bitxor(abs(vec(pos)), mapping(SSSS)),
SSSS);
else
    code{2, cnt} = dec2bin(vec(pos), SSSS);
end
len = len + strlen(code{1, cnt}) + length(code{2, cnt});
pos = pos + 1;
end
end
function Img_out = Block_Decode(code, CNT, Q, Dictionary, Dictionary_DC,
sz)
m = sz(1); m_new = ceil(m / 8) * 8;
n = sz(2); n_new = ceil(n / 8) * 8;
global index mapping
Img = zeros(m_new, n_new);
block = zeros(8, 8);
DC = 0;
pos = 1;%非 0 的位置
pos_x = 1;
pos_y = 1;
for i = 1 : CNT

```

---

```

%      disp(sprintf("i = %s", num2str(i))) %#ok<DSPS>
%      disp(sprintf("pos_x = %s   pos_y = %s", num2str(pos_x),
num2str(pos_y))) %#ok<DSPS>
%% DC 系数
if pos == 1
    SSSS = find(Dictionary_DC == code{1, i}) - 1;
    DIFF = code{2, i};
    if judge_positive(SSSS, DIFF)%正返回 1
        block(1, 1) = DC + bin2dec(DIFF);
    else
        block(1, 1) = DC - bitxor(bin2dec(DIFF), mapping(SSSS));
    end
    DC = block(1, 1);
    pos = pos + 1;
    continue;
end
%% AC 系数
[NNNN, SSSS] = find(Dictionary == code{1, i});
DIFF = code{2, i};%尾码
if NNNN == 17
    if SSSS == 1 %EOB
        pos = 1;
        Img(pos_x : pos_x + 7, pos_y : pos_y + 7) = idct2(block .*
Q);

        pos_y = pos_y + 8;
        if pos_y == n_new + 1
            pos_y = 1;
            pos_x = pos_x + 8;
        end
        block = zeros(8, 8);
    else %ZRL
        pos = pos + 16;
    end
    continue;
end
pos = pos + NNNN - 1;%有 NNNN - 1 个 0
if judge_positive(SSSS, DIFF)
    block(index(pos, 1), index(pos, 2)) = bin2dec(DIFF);
else
    block(index(pos, 1), index(pos, 2)) = -bitxor(bin2dec(DIFF),
mapping(SSSS));
end
if pos == 64
    pos = 1;

```

---

```

        Img(pos_x : pos_x + 7, pos_y : pos_y + 7) = idct2(block .* Q);
        pos_y = pos_y + 8;
        if pos_y == n_new + 1
            pos_y = 1;
            pos_x = pos_x + 8;
        end
        block= zeros(8, 8);
        continue;
    end
    pos = pos + 1;
end
Img_out = Img(1 : m, 1 : n);
Img_out(Img_out > 255) = 255;
Img_out(Img_out < 0) = 0;
end
function flag = judge_positive(SSSS, DIFF)
    if SSSS == 0
        flag = 1;
        return
    end
    num = bin2dec(DIFF);
    if (num < 2 ^ (SSSS - 1)) || (num > 2 ^ SSSS - 1)
        flag = 0;
    else
        flag = 1;
    end
end
end

```

#### 4. 实验结果及分析

本次实验使用了两幅测试图像，分别为 lena.bmp 和风景.bmp，均为三通道 RGB 图像。



图 4-1 lena 原始图像



图 4-2 恢复的 lena 图像

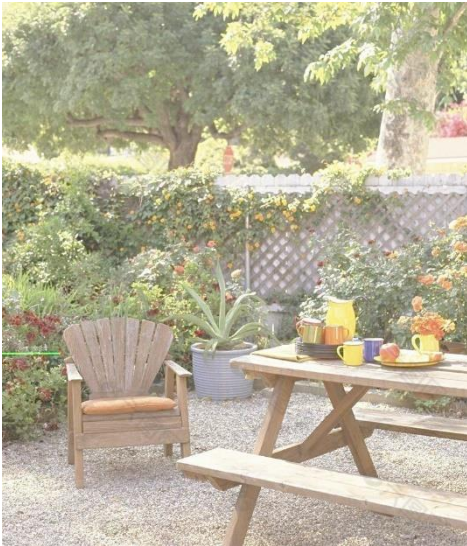


图 4-3 风景原始图像

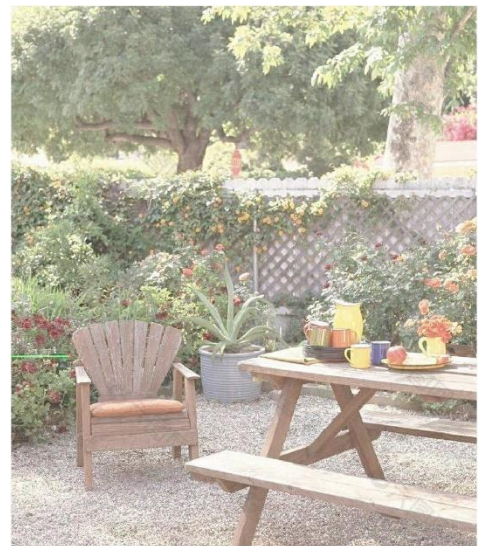


图 4-4 恢复的风景图像

算法在保存编码时计算了码流的长度，保存为 $LENGTH$ ，压缩比 $CR$ 计算方法如下，其中 $m, n$ 为图像的大小：

$$CR = \frac{m * n * 8 * 3}{LENGTH} \quad (4.1)$$

由于图像为三通道图像，为计算峰值信噪比 $PSNR$ ，本次实验将均方误差 $\overline{MSE}$ 认为是三通道的 $MSE$ 取均值，计算方法如下：

$$MSE_{r, g, b} = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I_{in}(i, j) - I_{out}(i, j)]^2 \quad (4.2)$$

$$PSNR = 10 \log_{10} \frac{MAX_I^2}{\overline{MSE}} \quad (4.3)$$

其中 $MAX_I$ 是图像可能的最大像素值，这里 $MAX_I = 255$ 。

Lena.bmp 的实验结果如下：

The Compression Rate CR = 41.7238  
The Peak Signal-to-Noise Ratio PSNR = 40.3864

图 4-5 lena 图像实验结果

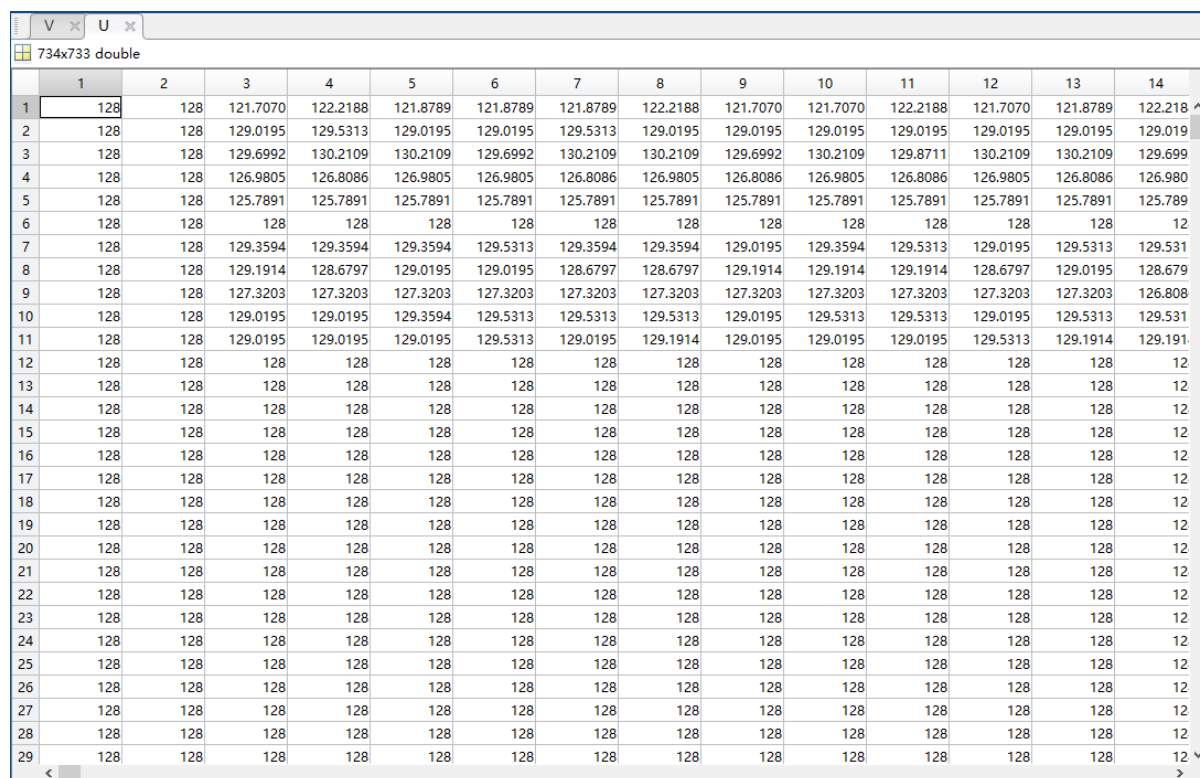
风景.bmp 的实验结果如下：

The Compression Rate CR = 16.3643  
The Peak Signal-to-Noise Ratio PSNR = 28.1882

图 4-6 风景图像实验结果

实验恢复的图像和原图像几乎没有肉眼可见的差别。Lena 图像的压缩比很高，通过观察图像像素值可以发现，lena 测试图像绝大多数像素位置三通道的值是相同的，因此在转为 YUV 图像后，U 分量和 V 分量几乎全为 128，只有少量不为 128，对这样一个内部元素均相同的  $8 \times 8$  矩阵进行离散余弦变换，其结果是仅最左上角的位置不为 0，其余位置均为 0，这对于压缩的帮助是十分巨大的。而风景图像是普通的彩色图像，所以其压缩比也就没有 lena 图像那么高。

图像的失真由两方面造成，一是对 U、V 分量的采样，二是量化过程中引起的误差，这些误差是不可避免的。



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	128	128	121.7070	122.2188	121.8789	121.8789	121.8789	122.2188	121.7070	121.7070	122.2188	121.7070	121.8789	122.2188
2	128	128	129.0195	129.5313	129.0195	129.0195	129.5313	129.0195	129.0195	129.0195	129.0195	129.0195	129.0195	129.0195
3	128	128	129.6992	130.2109	130.2109	129.6992	130.2109	130.2109	129.6992	130.2109	129.8711	130.2109	130.2109	129.6992
4	128	128	126.9805	126.8086	126.9805	126.9805	126.8086	126.9805	126.8086	126.9805	126.8086	126.9805	126.8086	126.9805
5	128	128	125.7891	125.7891	125.7891	125.7891	125.7891	125.7891	125.7891	125.7891	125.7891	125.7891	125.7891	125.7891
6	128	128	128	128	128	128	128	128	128	128	128	128	128	128
7	128	128	129.3594	129.3594	129.3594	129.5313	129.3594	129.3594	129.0195	129.3594	129.5313	129.0195	129.5313	129.5313
8	128	128	129.1914	128.6797	129.0195	129.0195	128.6797	128.6797	129.1914	129.1914	129.1914	128.6797	129.0195	128.6797
9	128	128	127.3203	127.3203	127.3203	127.3203	127.3203	127.3203	127.3203	127.3203	127.3203	127.3203	127.3203	126.8086
10	128	128	129.0195	129.0195	129.3594	129.5313	129.5313	129.5313	129.0195	129.5313	129.5313	129.0195	129.5313	129.5313
11	128	128	129.0195	129.0195	129.0195	129.5313	129.0195	129.1914	129.0195	129.0195	129.0195	129.5313	129.1914	129.1914
12	128	128	128	128	128	128	128	128	128	128	128	128	128	128
13	128	128	128	128	128	128	128	128	128	128	128	128	128	128
14	128	128	128	128	128	128	128	128	128	128	128	128	128	128
15	128	128	128	128	128	128	128	128	128	128	128	128	128	128
16	128	128	128	128	128	128	128	128	128	128	128	128	128	128
17	128	128	128	128	128	128	128	128	128	128	128	128	128	128
18	128	128	128	128	128	128	128	128	128	128	128	128	128	128
19	128	128	128	128	128	128	128	128	128	128	128	128	128	128
20	128	128	128	128	128	128	128	128	128	128	128	128	128	128
21	128	128	128	128	128	128	128	128	128	128	128	128	128	128
22	128	128	128	128	128	128	128	128	128	128	128	128	128	128
23	128	128	128	128	128	128	128	128	128	128	128	128	128	128
24	128	128	128	128	128	128	128	128	128	128	128	128	128	128
25	128	128	128	128	128	128	128	128	128	128	128	128	128	128
26	128	128	128	128	128	128	128	128	128	128	128	128	128	128
27	128	128	128	128	128	128	128	128	128	128	128	128	128	128
28	128	128	128	128	128	128	128	128	128	128	128	128	128	128
29	128	128	128	128	128	128	128	128	128	128	128	128	128	128

图 4-7 U 分量

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	128	128	122.2305	122.1484	121.7188	121.7188	121.7188	122.1484	122.2305	122.2305	122.1484	122.2305	121.7188	122.148
2	128	128	129.2891	129.2070	129.2891	129.2891	129.2070	129.2891	129.2891	129.2891	129.2891	129.2891	129.2891	129.289
3	128	128	130.1484	130.0664	130.0664	130.1484	130.0664	130.0664	130.1484	130.0664	129.6367	130.0664	130.0664	130.148
4	128	128	126.7109	127.2227	126.7109	126.7109	127.2227	126.7109	127.2227	126.7109	127.2227	126.7109	127.2227	126.710
5	128	128	125.9336	125.9336	125.9336	125.9336	125.9336	125.9336	125.9336	125.9336	125.9336	125.9336	125.9336	125.933
6	128	128	128	128	128	128	128	128	128	128	128	128	128	12
7	128	128	129.7188	129.7188	129.7188	129.2070	129.7188	129.7188	129.2891	129.7188	129.2070	129.2891	129.2070	129.207
8	128	128	128.7773	128.8594	129.2891	129.2891	128.8594	128.8594	128.7773	128.7773	128.7773	128.8594	129.2891	128.859
9	128	128	127.1406	127.1406	127.1406	127.1406	127.1406	127.1406	127.1406	127.1406	127.1406	127.1406	127.1406	127.222
10	128	128	129.2891	129.2891	129.7188	129.2070	129.2070	129.2070	129.2891	129.2070	129.2891	129.2070	129.2891	129.207
11	128	128	129.2891	129.2891	129.2891	129.2070	129.2891	128.7773	129.2891	129.2891	129.2891	129.2070	128.7773	128.777
12	128	128	128	128	128	128	128	128	128	128	128	128	128	12
13	128	128	128	128	128	128	128	128	128	128	128	128	128	12
14	128	128	128	128	128	128	128	128	128	128	128	128	128	12
15	128	128	128	128	128	128	128	128	128	128	128	128	128	12
16	128	128	128	128	128	128	128	128	128	128	128	128	128	12
17	128	128	128	128	128	128	128	128	128	128	128	128	128	12
18	128	128	128	128	128	128	128	128	128	128	128	128	128	12
19	128	128	128	128	128	128	128	128	128	128	128	128	128	12
20	128	128	128	128	128	128	128	128	128	128	128	128	128	12
21	128	128	128	128	128	128	128	128	128	128	128	128	128	12
22	128	128	128	128	128	128	128	128	128	128	128	128	128	12
23	128	128	128	128	128	128	128	128	128	128	128	128	128	12
24	128	128	128	128	128	128	128	128	128	128	128	128	128	12
25	128	128	128	128	128	128	128	128	128	128	128	128	128	12
26	128	128	128	128	128	128	128	128	128	128	128	128	128	12
27	128	128	128	128	128	128	128	128	128	128	128	128	128	12
28	128	128	128	128	128	128	128	128	128	128	128	128	128	12
29	128	128	128	128	128	128	128	128	128	128	128	128	128	12

图 4-8 V 分量

## 5. 遇到的主要问题及解决方法

### 1) 恢复的图像上存在一条光带。



图 4-9 恢复图像存在光带

这是在分块补齐时候产生的错误，对于 Y 分量应该补 0，U、V 分量应该补 128，出现光带是因为给 Y、U、V 均补 0 造成的，因此增加函数的参数，用来表示本次补 0 还是补 128，便可解决问题。

### 2) 编码过程中尾码的问题

---

在编码时，尾码分为正负两种情况，若要编码的数为正数，其尾码就是数的二进制表达，若要编码的数为负数，其尾码是数的二进制数的反码，然而 MATLAB 中函数 `dec2bin()` 的结果为字符型，取反操作为按位异或 1，MATLAB 中函数 `bitxor()` 要求操作数均为数字不能为字符，因此需要用原数字求补码，再将结果转为二进制保存为字符型。

对于正数，直接保存其二进制数的结果；对于负数，首先去掉负号，再按位异或运算，最后保存异或结果的二进制数。