Matlab 图像处理实验报告

xuan

一、基础知识

1. 使用 help images 阅读并大致了解图像处理工具箱中的函数基本功能

Imread 可读取图片各点的坐标并存储在 hall.mat 中，`imwrite` 将图像处理完的矩阵存入图片文件，`imshow` 可以查看图片，要查看多张图片可用 subplot，创建一个新的图窗窗口用 figure

2. 完成基本图像处理：以测试图像中心为圆心，长和宽中较小值的一半为半径画一个红色的圆；将测试图像涂成国际象棋状的“黑白格”的样子

首先定义图片宽高，半径和圆心的坐标点：

```
clear;close all;clc;
%read image
load('data/hall.mat');
[height,width,channel] = size(hall_color);%hall_color为unit8格式的原图
% define center and radius
center = [width+1,height+1]/2;%matlab中的下标从1开始，故中心坐标应各补上1/2。如1-2的中心为1.5而不是2/2=1
radius = min(width, height) / 2;
```

刚开始我想通过改变位于圆上的像素点的颜色来实现，但是卡在怎么选出位于圆上的像素点。后来和同学讨论后和查资料（https://zhuanlan.zhihu.com/p/126266520）后发现可以用 meshgrid 函数来实现红色实心圆的像素点选择。即先用 meshgrid 来建立坐标系，图片左上角为坐标（1,1）然后算出每个坐标点（像素点）距圆心的距离的矩阵 distance，再建立逻辑矩阵 circle_area=（distance<=radius）,即可选择出位于圆内的像素点。再改变这些点的三基色的强度即可。若只改变红色通道的值为 255，则圆比较透明，所以我把绿蓝色通道也设为 0,对比度更明显。

同理，我想到红色空心圆也可以这么做，只需限定 circle_area = ((distance <= 1.01*radius) & (distance >= 0.99*radius))即可。但后来学习 matlab 文档时看到 sub2ind 函数，又改进了自己的方法，用 sub2ind 函数来把坐标转换成索引可直接改变对应索引处的三通道亮度，显得更简洁。两种方法的结果是一样的。

（1）画红色实心圆

```
%draw filled circle
%构建标准方程和逻辑矩阵
img_fill_circle=hall_color;
[x, y] = meshgrid(1:width, 1:height);
distance = sqrt((x - center(1)).^2 + (y - center(2)).^2);
%circle_area = ((distance <= 1.01*radius) & (distance >= 0.99*radius));
circle_area = ((distance <= radius));
%构建三元组
R = hall_color(:,:,1);
G = hall_color(:,:,2);
B = hall_color(:,:,3);
%对于 RGB 三色不同色通道的操作实现红色实心圆
R(circle_area) = 255;
img_fill_circle(:,:,1) = R;
G(circle_area) = 0;
img_fill_circle(:,:,2) = G;
B(circle_area) = 0;
img_fill_circle(:,:,3) = B;
```

(2) 画红色空心圆:

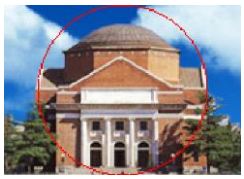
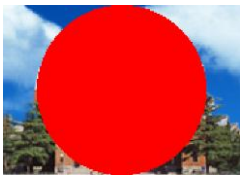
```
% draw empty circle
%构建参数方程（选取360个像素点）
img_empty_circle=hall_color;
angle = 0:359;
%不加max和min的话sub2ind会超范围,坐标得限制在1和宽/高之间
x = max(min(round(center(1) + radius * cos(angle)), width), 1);
y = max(min(round(center(2) + radius * sin(angle)), height), 1);
%构建三元组
R = hall_color(:,:,1);
G = hall_color(:,:,2);
B = hall_color(:,:,3);
%对于 RGB 三色不同色通道的操作实现红色空心圆
%因sub2ind是按列排序的，所以是y在前
R(sub2ind(size(img_empty_circle), y, x)) = 255;
img_empty_circle(:,:,1) = R;
G(sub2ind(size(img_empty_circle), y, x)) = 0;
img_empty_circle(:,:,2) = G;
B(sub2ind(size(img_empty_circle), y, x)) = 0;
img_empty_circle(:,:,3) = B;
```

非原创等级:改进参考

（3）画棋盘：用 meshgrid 函数实现应该是最简单的，因为 0 表示黑色，可先构建一个黑白棋盘（即 0 和 1 的矩阵），0 和原图像素点相乘仍为 0，即黑色，1 和原图像素点相乘为原图像素点

```
%draw grid
[x, y] = meshgrid(1:width, 1:height);
grid_size = 8; %一格的大小（width and height的最大公约数）
%构建黑白相间的棋盘
xgrid  = mod(floor((x-1) / grid_size), 2);
ygrid = mod(floor((y-1) / grid_size), 2);
grid_mask = xor(xgrid, ygrid);
img_grid = uint8(grid_mask) .* hall_color;%.*:整数只能与同类的整数或双精度标量值组合使用。
```

结果如下：


用 HuaweiImageViewer 查看上述 png 文件均达到了目标。分别为 result 文件夹中的 emptyCircle.png、fill_circle.png、grid.png

二、图像压缩编码

定义了 prepocess 函数用于将图像补足成宽高都是 8 的倍数的图像
1.可以在变换域进行。从图像矩阵中随机取得 8*8 的一小块矩阵记为 P,DCT 算子为 D，最终系数为 C。

$$C = D\left(P - \begin{bmatrix} 128 & \cdots & 128 \\ \vdots & \ddots & \vdots \\ 128 & \cdots & 128 \end{bmatrix}\right)D^T = DPD^T - D\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}128\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}^T D^T$$ 为空域的处理方法

$C = DPD^T - 128\begin{bmatrix} N & 0_{1\times(N-1)} \\ 0_{N-1\times 1} & 0_{(N-1)\times(N-1)} \end{bmatrix}$为频域的处理方法

代码如下：

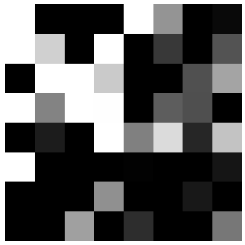
```
%读取height和width
img_width = 8;
[height, width] = size(hall_gray);
%选取随机的8*8矩阵块
piece_x = floor(rand * (height - img_width) + 1);
piece_y = floor(rand * (width - img_width) + 1);
rand_piece = double(hall_gray(piece_x:(piece_x + img_width - 1),piece_y:(piece_y + img_width - 1)));
%空域
piece_sd = rand_piece - 128;
dct_sd = dct2(piece_sd);%matlab的dct2(A)函数返回 A 的二维离散余弦变换
%频域
dct_fd = dct2(rand_piece);
dct_fd(1,1) = dct_fd(1,1) - 128 * img_width;
%算出二范数
disp(norm(dct_sd - dct_fd));
```

执行程序若干次，得到若干随机的图像块，得到的二范数均为 10^-13 量级，即空域和频域的图像块基本相同。由此可以认为两种方法得到的变换域矩阵是相等的。

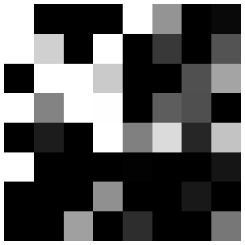
两个变换域矩阵的图像如下，结果保存在 result 文件夹的 preprocess_rand_piece.png、preprocess_dct_sd.png、preprocess_dct_fd.png

随机8*8块

空域

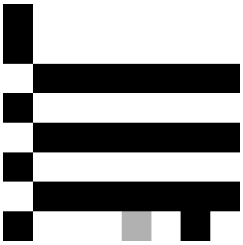


频域

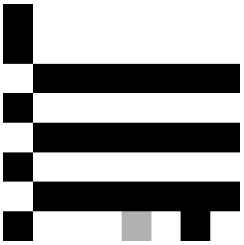


随机8*8块

空域



频域



2.实现二维 DCT

以下是自己编写的 mydct 函数

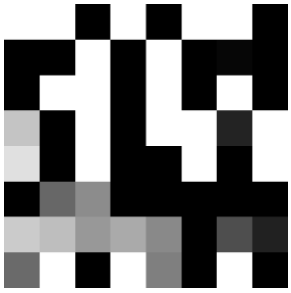
```
function C = mydct(P)
% 输入P需为方阵
[height,width] = size(P);
assert(height==width, "input isn't a square matrix");
C = double(P);
N = size(P, 1);%保证D和P大小相同
D = cos((1:2:(2 * N - 1)) .* (0:1:(N - 1))' * pi / 2 / N);%构建DCT矩阵
D(1, :) = sqrt(0.5);
D = sqrt(2 / N) * D;
C = D * C * D';
end
```

```
%读取height和width
img_width = 8;
[height, width] = size(hall_gray);
%选取随机的8*8矩阵块
piece_x = floor(rand * (height - img_width) + 1);
piece_y = floor(rand * (width - img_width) + 1);
rand_piece = double(hall_gray(piece_x:(piece_x + img_width - 1),piece_y:(piece_y + img_width - 1)));
% spatial domain
piece_sd = rand_piece - 128;
dct_sd = dct2(piece_sd);%matlab的dct2(A)函数返回 A 的二维离散余弦变换
dct_sd1 = mydct(piece_sd);
% frequency domain
dct_fd = dct2(rand_piece);
dct_fd(1,1) = dct_fd(1,1) - 128 * img_width;
dct_fd1 = mydct(rand_piece);
dct_fd1(1,1) = dct_fd1(1,1) - 128 * img_width;
%算出二范数
disp(norm(dct_sd - dct_fd));
disp(norm(dct_sd1 - dct_fd1));
```

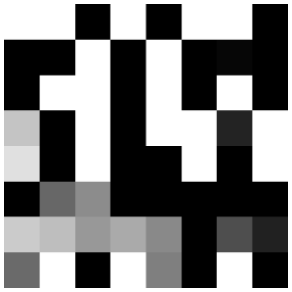
执行程序若干次，得到的二范数与 matlab 提供的 dct2 函数有误差，但还是在 10^-13 量级由此可以认为两种变换的结果是一样的。图像如下:

matlab提供的dct2空域

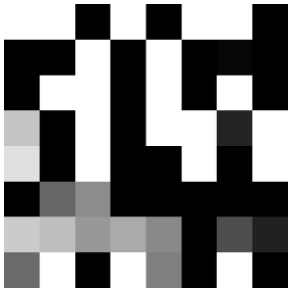
matlab提供的dct2频域



自行实现的mydct空域



自行实现的mydct频域



可以看到自行实现的 mydct 和 MATLAB 自带的库函数 dct2 变换结果基本一样

3.代码如下:

分别对图像的每一个 8*8 块做，可用分块处理函数 blockproc,这样相比双重循环的实现效率更高。

```
%hall_sub_select = hall_process(17:32, 41:56);
hall_sub = hall_process- 128;
[height, width] = size(hall_sub);
%不置零变换的图像
fun = @(block_struct) dct2(block_struct.data);
DCT = blockproc(hall_sub, [8 8], fun);
fun = @(block_struct) idct2(block_struct.data)+128;
D_nozero = blockproc(DCT, [8 8], fun);
%右侧四列置零变换的图像
fun = @(block_struct) dct2(block_struct.data);
DCT = blockproc(hall_sub, [8 8], fun);
for a = 0:(width / 8)-1
    DCT(:,5+8*a:8+8*a) = 0;
end
fun = @(block_struct) idct2(block_struct.data)+128;
D_rightzero = blockproc(DCT, [8 8], fun);
%左侧四列置零变换的图像
fun = @(block_struct) dct2(block_struct.data);
DCT = blockproc(hall_sub, [8 8], fun);
for a = 0:(width / 8)-1
    DCT(:,1+8*a:4+8*a) = 0;
end
fun = @(block_struct) idct2(block_struct.data)+128;
D_leftzero = blockproc(DCT, [8 8], fun);
```

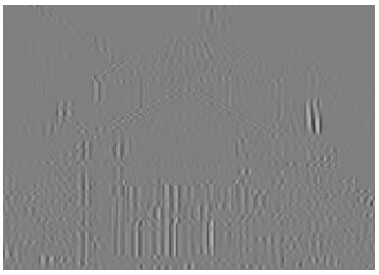
图像如下：左上为原图，右上为经过 dct 变换后又逆变换回来的图像
左下是右侧四列置零变换的图像，右下为左侧四列置零变换的图像

未经dct的原图像

不置零变换的图像

右侧四列置零变换的图像

左侧四列置零变换的图像



进一步提取横坐标 51-58，纵坐标 21-28 的区域进行观察，如下图

```
%展示图片
figure('Name', 'DCT_zero', 'NumberTitle', 'off');
subplot(2, 2, 2);
imshow(uint8(D_nozero(21:28,51:58)));
title("不置零变换的提取图像");
subplot(2, 2, 3);
imshow(uint8(D_rightzero(21:28,51:58)));
title("右侧四列置零变换的提取图像");
subplot(2, 2, 4);
imshow(uint8(D_leftzero(21:28,51:58)));
title("左侧四列置零变换的提取图像");
```

不置零变换的提取图像

右侧四列置零变换的提取图像

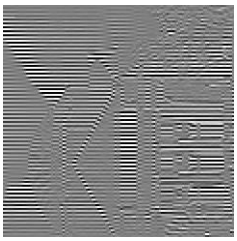
左侧四列置零变换的提取图像

可以看到，不置零的图像和原图基本相同；而右侧 4 列置零的图像基本形貌能分辨，但分辨率下降了；而左侧 4 列置零的图像则基本已经难以分辨出原始图像了，只剩下一点轮廓。从 DCT 变换的原理来看，DCT 矩阵的左上角为直流和低频分量，左下角为竖直高频和水平低频分量，右上角为竖直低频和水平高频分量，右下角为高频分量。低频分量是人认知图像的主要信息来源，高频分量则是明暗变化剧烈的边界。所以当左侧置零时，水平方向的低频分量被消去，只剩高频分量，在图片上就是剩下明暗变化剧烈的边界；当右侧置零时，水平方向的高频分量被消去，只剩低频分量，表现在测试图片 上就是明暗变化处模糊化。

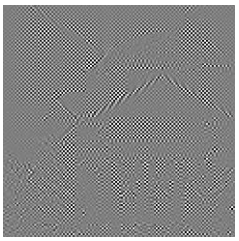
4.分别对全图和每一个 8*8 做不同的转置变换，然后逆变换得到结果如下：

D tran whole

D rot90 whole

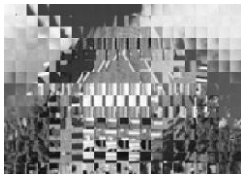


D rot180 whole

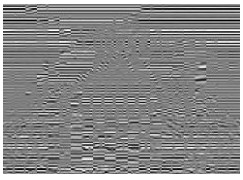


对全图的系数矩阵进行转置（左边图），即对系数矩阵 C 进行转置，相当于将水平和竖直方向的系数做了交换，逆变换得到的结果也就是由水平的变成了竖直的。即$\mathrm{P}^{\mathrm{T}} = (\mathrm{D}^{\mathrm{T}}\mathrm{CD})^{\mathrm{T}} = \mathrm{D}^{\mathrm{T}}\mathrm{CD}$。

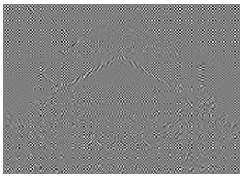
D tran every



D rot90 every



D rot180 every



对于每个 8×8 块的系数矩阵转置再还原其实就是把原图的每一块进行转置后拼接在了一

起，但每一块的位置不变，所以大体轮廓还在，整图也没有旋转，但是细节难以辨识。
系数矩阵旋转 90°把原系数矩阵中值较大的直流和低频区域旋转到了表征竖直方向的高频区，所以还原得到的图像有强烈的横条纹，对全图和分别对每一个 8*8 都是这样。
旋转 180°则是将原系数矩阵中值较大的直流和低频区域旋转到了同时表征竖直方向和水平方向的高频区，所以还原得到的图像有强烈的棋盘格黑白交替的表现，对全图和分别对每一个 8*8 都是这样。
代码如下：同理用 blockproc 函数比双重循环效率更高

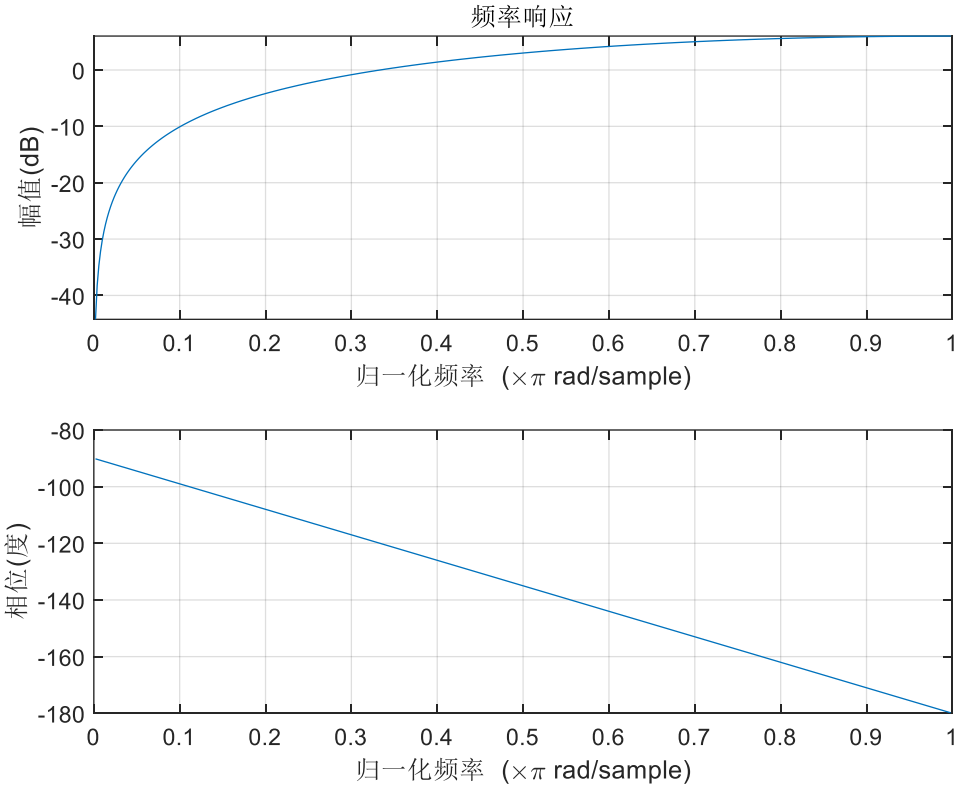
```
%对每一个 8 × 8 的系数矩阵进行转置
fun = @(block_struct) dct2(block_struct.data);
DCT = blockproc(hall_sub, [8 8], fun);
fun = @(block_struct) idct2(transpose(block_struct.data))+128;
D_tran_every = blockproc(DCT, [8 8], fun);
%对每一个 8 × 8 的系数矩阵逆时针旋转90
fun = @(block_struct) dct2(block_struct.data);
DCT = blockproc(hall_sub, [8 8], fun);
fun = @(block_struct) idct2(rot90(block_struct.data))+128;
D_rot90_every = blockproc(DCT, [8 8], fun);
%对每一个 8 × 8 的系数矩阵逆时针旋转180
fun = @(block_struct) dct2(block_struct.data);
DCT = blockproc(hall_sub, [8 8], fun);
fun = @(block_struct) idct2(rot90(block_struct.data,2))+128;
D_rot180_every = blockproc(DCT, [8 8], fun);
%对全图的系数矩阵进行转置
edge = min(width,height);
C = dct2(hall_sub(1:edge,1:edge));
D_tran_whole = idct2(C')+128;
%对全图的系数矩阵逆时针旋转90
D_rot90_whole = idct2(rot90(C))+128;
%对全图的系数矩阵逆时针旋转180
D_rot180_whole = idct2(rot90(C,2))+128;
```

5.该差分编码系统的差分方程如下：

$$y(n) = x(n-1) - x(n)$$

n=1 时 c^D(n)=-c~D(n)，为线性时不变系统
做 Z 变换得到传递函数$H(z) = z^{-1} - 1$
用 freqz 函数得到频率响应如下

```
clear;close all;clc;
b = [-1,1];%差分方程右侧系数
a = [1,0];%差分方程左侧系数
figure;
freqz(b,a),title('频率响应');
```



上图的频率响应说明它是一个高通滤波器。

DC 系数先进行差分编码再进行熵编码，说明 DC 系数的低频分量更多，所以才要用高通滤波器滤除。

（6）根据表格，若记预测误差为 Error，则 Category 可由下式求得：

$$\text{Category} = \lceil \log2(|\text{Error}| + 1) \rceil$$

Error 为预测误差

（7）我知道的方法有索引法和循环法。

循环法可以对任意形状的矩阵使用，以 O(n^2) 的复杂度进行一轮 Zig-Zag 扫描，效率相对差。而索引法更高效，但要牺牲空间来换时间。而本次实验 Zig-Zag 扫描全是 8*8 矩阵，需要的空间不大，使用索引法更高效。即事先定义一个索引数组作为表，然后使用查表的方法进行扫描即可。函数如下

```
function vector = myzig_zag(matrix)
%对输入的8*8矩阵实现 zigzag 扫描
% matrix:需要进行 zigzag 扫描的矩阵，必须是8*8矩阵
index = [1, 2, 9, 17, 10, 3, 4, 11, 18, 25, 33, 26, 19, 12, 5, 6,...
13, 20, 27, 34, 41, 49, 42, 35, 28, 21, 14, 7, 8, 15, 22, 29,...
36, 43, 50, 57, 58, 51, 44, 37, 30, 23, 16, 24, 31, 38, 45, 52,...
59, 60, 53, 46, 39, 32, 40, 47, 54, 61, 62, 55, 48, 56, 63, 64];
matrix = matrix';
vector = matrix(index);
end
```

（8）为方便后面题目的调用，直接把预处理（将宽高补足成 8 的倍数）、分块、DCT 和量化，结果的矩阵 result 每一列为一个块的 DCT 系数 Zig-Zag 扫描后形成的列矢量，第一行为各个块的 DC 系数。

刚开始想手动实现分块，后来看到 https://blog.csdn.net/sktzsktz/article/details/85542112 介绍的 blockproc 函数，仿照示例构造了分块量化和 zigzag 的函数 dct_quant_zig，再用 block_struct 进行分块函数变换。

实现的思想是按照分块方法，利用 blockproc 函数分块，处理顺序是从左上角开始，先行后列依次编码各块；再对每一块的 8*8 矩阵进行 DCT，量化、zig-zag 扫描，最终会把 8*8 矩阵变成 64*1 的列向量；即 blockproc 函数的输出结果 process 每一列有多个 64*1 的列向量（一个块的 DCT 系数的 zig-zag 扫描），最后利用双循环重新排列即可。如下图

```
function result = block_dct_quant_zig(hall_gray,QTAB)
%对原始图像进行预处理（将宽高补足成8的倍数）、-128、分块、DCT和量化，结果的矩阵
%每一列为一个块的DCT系数Zig-Zag扫描后形成的列矢量，第一行为各个块的DC系数
%hall_gray 输入的原始图像
%QTAB 量化步长
%result 输出的结果矩阵
img = preprocess(hall_gray) - 128;
fun = @(block_struct) dct_quant_zig(block_struct.data, QTAB);
[height, width] = size(hall_gray);
process = blockproc(img, [8 8], fun);%960*21
result = zeros(64, height*width/64);%先排完21
for a = 0:height*8/64-1%共15
    for b = 1:width/8
        result(:,b+21*a) = process(1+64*a:64+64*a,b);%64*315
    end
end
end

function output = dct_quant_zig(input, QTAB)
    C = dct2(input);
    C_quant = round(C ./ QTAB);
    output = myzig_zag(C_quant);%64*1的列向量
end
```

对测试图像进行分块、DCT、量化等，结果如下

```
clear;close all;clc;
load('data/JpegCoeff.mat');
load('data/hall.mat');
result = block_dct_quant_zig(hall_gray,QTAB);
```

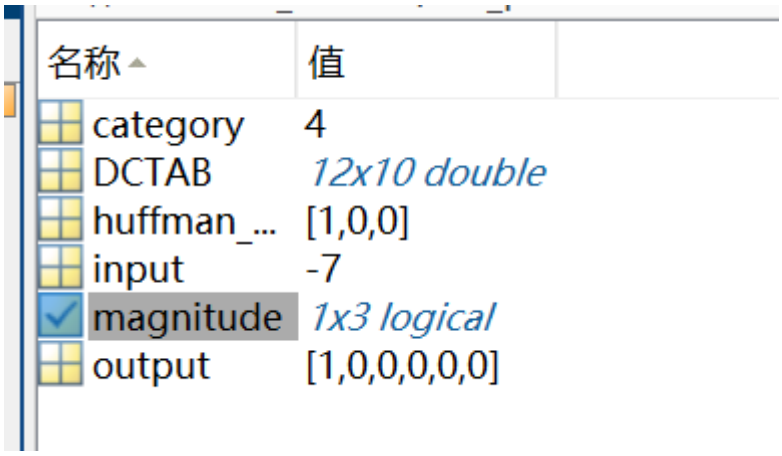
结果 result 见 result 文件夹的 result.mat
非原创等级：借鉴参考
（9）按照定义设计函数即可
输出 DC 系数的码流、AC 系数的码流、图像高度和图像宽度，写入 jpegcodes.mat 文件
把各个图像块的量化后的直流分量表示为 1 个矢量$\tilde{c}_D$
差分运算得到$\hat{c}_D(\mathrm{n}) = \tilde{c}_D(\mathrm{n}-1) - \tilde{c}_D(\mathrm{n})$
然后分别对每个系数计算 category、huffman_code 和 DC_magnitude，代码如下：

```
%计算DC码流
DC_coef = coef(1, :);%把各个图像块的量化后的直流分量表示为1个矢量c~D
DC_coef = [DC_coef(1), DC_coef(1:end - 1)- DC_coef(2:end)] ;%差分运算得到c^D
dc_block_process = @(data) DC_process(data, DCTAB);
DC = arrayfun(dc_block_process, DC_coef, 'UniformOutput', false);%Uniform 输出中不能存在非标量值
DC = cell2mat(DC);%把cell转换成mat,计算出DC码流
```

```
function bin_vec = dec2bin_j(dec_int)%把十进制转换成二进制（负数表示为1补码）
    if(dec_int > 0)
        bin_vec = split(dec2bin(dec_int), '');%是str格式的二进制,且首尾为空
        bin_vec = str2double(bin_vec(2:end - 1))';
    elseif(dec_int == 0)
        bin_vec = [];
    else
        bin_vec = split(dec2bin(abs(dec_int)), '');
        bin_vec = ~str2double(bin_vec(2:end - 1)).';
    end
end

function output = DC_process(input, DCTAB)%对每个DC系数计算category、huffman_code和DC_magnitude
    category = ceil(log2(abs(input)+1)) + 1;%Category = ⌈log2(|Error| + 1)⌉
    huffman_code = DCTAB(category,2:DCTAB(category, 1) + 1);%找到category对应的huffman_code
    DC_magnitude = dec2bin_j(input);%预测误差的二进制表示（负数为1的补码）
    output = [huffman_code, DC_magnitude];
end
```

Magnitude 为预测误差的二进制形式，且若预测误差为负数需表示为 1 的补码（即反码）如 -7 对应二进制 111, 对应 1 补码为 000; 对应 huffman_code 为 100 所以 output 为[1,0,0,0,0,0]



计算 AC 码流代码如下:

```
%计算AC码流
AC_coef = coef(2:end, :);%Zig-Zag扫描形成列矢量去掉DC系数 63*315
ZRL = [1 1 1 1 1 1 1 1 0 0 1];%16个连零编码为1111-1111-001
EOB = [1 0 1 0];
AC = [];
for i = 1:blocks
    block = AC_coef(:, i);%63*1 一个AC系数块
    while(true)
        % find 第一个非零系数
        %c_ind表示这个系数在block(不断缩短）的第几个位置，c_mag即为这个系数的值
        [c_ind, ~, c_mag] = find(block, 1);
        if(isempty(c_ind))
            break;%block已经没有非零系数
        end
        c_run = c_ind - 1;%c_run的值表示这个系数前面有几个零
        while(c_run > 15)%当run>15时，插入ZRL表示16个连零
            c_run = c_run - 16;%减16后剩下的零也要参与huffman的计算
            AC = [AC, ZRL];
        end
        c_size = ceil(log2(abs(c_mag)+1));%计算size(即DC的Category)
        huffman = ACTAB(c_run * 10 + c_size,4:ACTAB(c_run * 10 + c_size, 3) + 3);%计算对应的huffman
        AC_magnitude = dec2bin_j(c_mag);%计算magnitude
        AC = [AC, huffman, AC_magnitude];%一个AC系数块算完，还要连上之前的AC系数的AC码流
        block(1:c_ind) = [];%没循环一次，block都删掉第一个非零系数及其之前的零
    end
    AC = [AC, EOB];%计算出AC，加上结束符
end
end
```

要注意每次计算每个 AC 系数矩阵对应的 AC 码流都要连上之前的 AC 系数的 AC 码流和 EOB

最后把结果存进 jpegcodes.mat

```
clear;close all;clc;
load('data/JpegCoeff.mat');
load('data/hall.mat');
[DC, AC, height, width] = JPEG_encode(hall_gray, QTAB, DCTAB, ACTAB);
save('jpegcodes.mat', 'DC', 'AC', 'height', 'width');
fprintf("DC码流长度:%d, AC码流长度:%d, 高度:%d, 宽度:%d\n", size(DC, 2), size(AC, 2), height, width);
%压缩比=输入文件长度/输出码流长度
fprintf("压缩比为: %5.4f\n", height * width * 8 / (size(DC, 2) + size(AC, 2)));
```

```
令行窗口
DC码流长度:2031, AC码流长度:23072, 高度:120, 宽度:168
压缩比为: 6.4247
```

（10）如上图，计算得压缩比为 6.4247，即压缩比 = $\frac{\text{输入文件长度}}{\textit{输出码流长度}}$ = $\frac{8\ (unit8)*120\ (高度)*168(宽度)}{2031+23072} = 6.4247$

(11)按照解码的流程逐步实现即可

逆 zig-zag:

```
function matrix = myizigzag(vector)
%myzigzag 实现逆zigzag扫描
%vector:用于逆向zigzag扫描得到矩阵的输入一维向量
index = [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];
    matrix = reshape(vector(index), 8, 8).';
end
```

逆 zig-zag、反量化、IDCT、逆分块和+128

```
function image = iquantify(image_quantified, QTAB, height, width)
%iquantify 实现逆zigzag、反量化、IDCT、逆分块和+128功能
%   image_quantified:图像的量化矩阵
%   QTAB:量化系数
%   height: 原始图像高度
%   width: 原始图像宽度
h = ceil(height / 8);
w = ceil(width / 8);
img = zeros(8 * h, 8 * w);
%实现逆zigzag、反量化、IDCT
for a = 1:h
    for b = 1:w
        %注意重建时的块要一一对应
        img((8 * a - 7):(8 * a), (8 * b - 7):(8 * b)) = idct2(myizigzag(image_quantified(:,w  * (a - 1) + b)
    end
end
%舍去多余，+128
image = img(1:height, 1:width);
image = uint8(image + 128);
end
```

DC 码流还原为 DC 系数:

```
function dc_coeff = DCdecode(DCList, col)
%函数功能：把DC码流还原为DC系数
 %DCList是1*2031行向量
%col: DC系数的个数 即有图片分为col个8*8矩阵块
    dc_coeff = zeros(1, col);
    for b = 1:col
        if(all(DCList(1:2) == 0))
            %对应Huffman=00,catagory=0,预测误差c^D为0，DC码流为00
            DCList(1:2) = [];
            % 因dc_coeff初始化为全0,此时对应dc_coeff(b)已经是0
        else
            %寻找Huffman_code第一个0的位置
            index = find(~DCList, 1);
            % 计算category
            if(index <= 3)%对应category为1到5，Huffman_code固定为3bit
                %Huffman_code为category的二进制表示
                category = bin2dec(num2str(DCList(1:3), '%d')) - 1;
                DCList(1:3) = [];%用完的DC码流就删除掉
            else%对应catagory为6到11；有关系category = index + 2;
                category = index + 2;
                DCList(1:index) = [];
            end
            %根据category和DC码流计算magnitude,category即为magnitude的bit数
            magnitude = DCList(1:category);
            %根据magnitude还原出DC系数（c^D)
            if(magnitude(1) == 1)
                %首bit是1说明是正数，直接把二进制转成十进制
                dc_coeff(b) = bin2dec(num2str(magnitude, '%d'));
            else
                %首bit是0说明是负数(1补码),先逐bit取反，转成十进制后还要加个负号
                dc_coeff(b) = -bin2dec(num2str(~magnitude, '%d'));
            end
```

比较难的是 AC 码流还原为 AC 系数，需要根据码长逐个匹配 Huffman_code

```
ac_coeff = zeros(63, col);
b = 1;pos = 0;
while(b <= col)
    %AC系数矩阵的非零系数被转换完后，就被删除掉
    if(all(AcList(1:4) == EOB))%结束符说明一个AC系数矩阵已经转换完，b++
        b = b + 1;
        pos = 0;
        AcList(1:4) = [];
    elseif(length(AcList) > 11 && all(AcList(1:11) == ZRL))%判断遇到了16个连零
        pos = pos + 16;
        AcList(1:11) = [];
    else
        %取run和size，size即对应Amplitude的bit数，即Huffman_code后面的size比特即为AC非零系数的二进制
        %ACTAB第一列是Run,第二列是size,第三列是码长，以后则是Huffman_code
        for c = 1:size(ACTAB, 1)%按表2.3从上到下的顺序一个个匹配Run/size
            L = ACTAB(c, 3);%码长
            %若码长比AcList剩下的长度还长，肯定不是该Huffman_code,也是防止索引超出数组元素的数目
            if(L > length(AcList))
                continue;
            end
            if(all(AcList(1:L) == ACTAB(c, 4:L + 3)))
                ac_run = ACTAB(c, 1);%表示非零系数前面有几个零
                ac_size = ACTAB(c, 2);
                AcList(1:L) = [];
                break;
            end
        end
        pos = pos + ac_run + 1;%表示非零系数在zig_zag后的系数向量中的位置
        %同DC系数，把amplitude表示的二进制转换为十进制的AC系数
        amplitude = AcList(1:ac_size);
        if(amplitude(1) == 1)
            ac_coeff(pos, b) = bin2dec(num2str(amplitude, '%d'));
        else
```

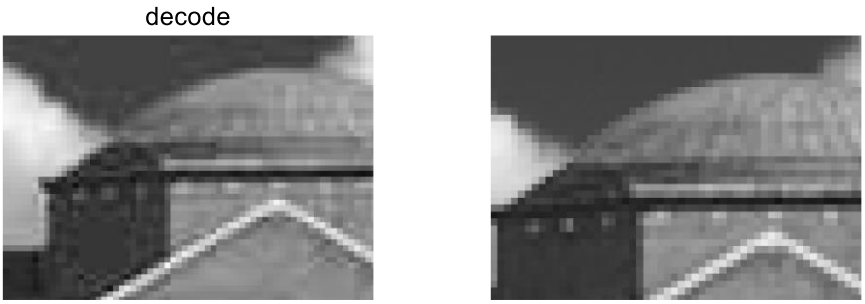
decode

PSNR 评价：如下图，PSNR=34.8926.，失真较小，编解码效果比较好，图像质量损失较小。
主观评价：对比原图，经过编解码的图片基本与原图相同，但变得模糊了些，多出了一些噪声，也隐隐有了分块的边界。

```
clear;close all;clc;
load('data/JpegCoeff.mat');
load('data/hall.mat');
jpegcode = load('jpegcodes.mat');
img = JPEG_decode(jpegcode.DC, jpegcode.AC, jpegcode.height, jpegcode.width, QTAB, ACTAB);
%客观评价
PSNR = 10 * log10(255^2 / mse(img, hall_gray));
fprintf("PSNR = %5.4fdB\n", PSNR);
%展示图片
figure;
subplot(1,2,1);imshow(img);title('decode');
imwrite(img, 'results/2_11_decode.png');
subplot(1,2,2);imshow(hall_gray);title('原始图像');
imwrite(hall_gray, 'results/2_11_origin.png');
```

命令行窗口

```
PSNR = 34.8926dB
```



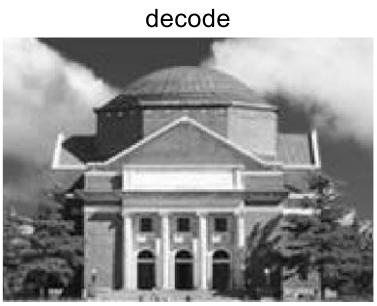
放大观察，明暗交界如天空与云交界处，多出了许多噪点。

（12）量化步长减少为原来一半后，重新编解码，结果如下

```
clear;close all;clc;
load('data/JpegCoeff.mat');
load('data/hall.mat');
QTAB = QTAB/2;%量化步长减少为原来的一半
[DC, AC, height, width] = JPEG_encode(hall_gray, QTAB, DCTAB, ACTAB);
img = JPEG_decode(DC, AC, height, width, QTAB, ACTAB);
fprintf("DC码流长度:%d, AC码流长度:%d, 高度:%d, 宽度:%d\n", size(DC, 2), size(AC, 2), height, width);
%压缩比=输入文件长度/输出码流长度
fprintf("压缩比为: %5.4f\n", height * width * 8 / (size(DC, 2) + size(AC, 2)));
%客观评价
PSNR = 10 * log10(255^2 / mse(img, hall_gray));
fprintf("PSNR = %5.4fdB\n", PSNR);
%展示图片
figure;
subplot(1,2,1);imshow(img);title('decode');
imwrite(img, 'results/2_12_decode.png');
subplot(1,2,2);imshow(hall_gray);title('原始图像');
imwrite(hall_gray, 'results/2_12_origin.png');
```

命令行窗口

```
DC码流长度:2410, AC码流长度:34164, 高度:120, 宽度:168
压缩比为: 4.4097
PSNR = 37.3208dB
```



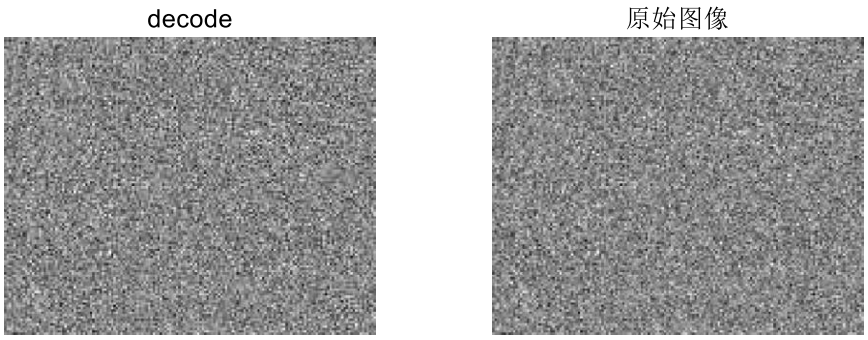
如上图，编码得到的 DC 码流长度 2410，AC 码流长度:34164，高度:120，宽度:168；
压缩比为:4.4097；相比前面的结果，压缩比增加，因为量化步长减半导致 DC 与 AC 系数相对较大，所以编码变长。
PSNR 为: 37.3208，结果比前面的大，说明量化步长减少后编解码效果更好，图像质量损失更小。
主观评价：对比原图，量化步长减少后重新编解码的图片噪声情况有所改善。
（13）代码如下

```
clear;close all;clc;
load('data/JpegCoeff.mat');
load('data/snow.mat');
[DC, AC, height, width] = JPEG_encode(snow, QTAB, DCTAB, ACTAB);
img = JPEG_decode(DC, AC, height, width, QTAB, ACTAB);
fprintf("DC码流长度:%d, AC码流长度:%d, 高度:%d, 宽度:%d\n", size(DC, 2), size(AC, 2), height, width);
%压缩比=输入文件长度/输出码流长度
fprintf("压缩比为: %5.4f\n", height * width * 8 / (size(DC, 2) + size(AC, 2)));
%客观评价
PSNR = 10 * log10(255^2 / mse(img, snow));
fprintf("PSNR = %5.4fdB\n", PSNR);
%展示图片
figure;
subplot(1,2,1);imshow(img);title('decode');
imwrite(img, 'results/2_13_decode.png');
subplot(1,2,2);imshow(snow);title('原始图像');
imwrite(snow, 'results/2_13_origin.png');
```

命令行窗口

```
DC码流长度:1403, AC码流长度:43546, 高度:128, 宽度:160
压缩比为: 3.6450
PSNR = 29.5614dB
```

如上图，将雪花图像进行编解码，DC 码流长度 1403，AC 码流长度:43546，高度:128，宽度:160；压缩比为: 3.6450；PSNR = 29.5614dB。得到如下图像



可以看到编解码的图像与原始图像基本相同，但在细节上有不少差异
PSNR 和压缩比比之前少了很多，而 AC 码流比之前增加很多，我认为应该是雪花图像相对接近随机图像，高频分量比较多，所以 AC 码流较长，同时本次大作业使用的量化系数应该是针对一般图片的，是考虑将一般的图像中不太重要的高频分量较多地滤除。而雪花图接近随机的矩阵，量化表不适合对其进行处理，故最终压缩比相对较低。

三、信息隐藏

（1）空域隐藏代码如下：使用 rand 和 ceil 生成一个给定长度（img_len/2）的随机 01 序列。通过二进制转换，将图片第一个到第隐藏信息长度个像素亮度分量最低比特位替换为该随机序列。一是不经过编解码直接还原成图片取出隐藏信息与原有序列对比，二是经过 JPEG 编解码后与原有序列对比，分别记录正确率。

```
img_len = height * width;%图片大小
% 任意生成一个需要隐藏的01序列
hid_len = img_len/2;%定义并记录隐藏信息的长度
hide_info = ceil(rand(hid_len,1)-0.5);%生成长度为hid_len的随机01序列
% 隐藏信息
%图片的每一个像素转成二进制是8bit，在像素亮度分量最低比特位放隐藏信息
P = dec2bin(image.');%把图片的像素转换成二进制：img_len*8
P(1:hid_len, end) = string(hide_info);%最低位替换为要隐藏的信息
% 不编解码直接还原回图片
img = reshape(bin2dec(P), width, height).';
% 将隐藏信息抽取出来
C = dec2bin(img.');%把图片的像素转换成二进制：img_len*8
extract_info = bin2dec(C(1:hid_len, end));%从像素亮度分量最低比特位提取隐藏信息
correct = length(find(extract_info==hide_info));%计算和原始隐藏信息相同的个数
correct_rate = correct / hid_len;
fprintf("只隐藏信息的正确率：%f\n", correct_rate);
% JPEG编码和解码
[DC, AC, height, width] = JPEG_encode(img, QTAB, DCTAB, ACTAB);
img = JPEG_decode(DC, AC, height, width, QTAB, ACTAB);
% 将隐藏信息抽取出来
C = dec2bin(img.');%把图片的像素转换成二进制：img_len*8
extract_info = bin2dec(C(1:hid_len, end));%从像素亮度分量最低比特位提取隐藏信息
correct = length(find(extract_info==hide_info));%计算和原始隐藏信息相同的个数
correct_rate = correct / hid_len;
fprintf("JPEG编码解码后的正确率：%f\n", correct_rate);
```

命令行窗口

```
只隐藏信息的正确率：1.000000
JPEG编码解码后的正确率：0.495437
```

原图片　　隐藏信息后的图片　　隐藏信息后编解码的图片



可以看到隐藏信息后不编解码和 JPEG 编解码后的图像与原图片基本相同，隐藏信息后直接将信息抽出是正确率是 1，但是经编解码后将信息抽取出来，正确率为 0.5 左右，因为信息是随机生成的，又试了几次随机生成的序列都如此，且改变序列长度 hid_len 到 100 或 1000 正确率都无明显变化。这个正确率是很低的，肯定是不能接受的，也说明这种方法抗 JPEG 编码的能力弱，不适合使用

（2）变换域信息隐藏

为方便起见，把 JPEG_encode 和 JPEG_decode 函数重写了，其中把计算 DC、AC 码流；量化、反量化；还原 DC、AC 系数；zig_zag、反 zig_zag 等操作都封装成了函数

第一种方法：

原图片

隐藏信息后的编解码图片

```
clear;close all;clc;
load('data/hall.mat');
load('data/JpegCoeff.mat');
image = double(hall_gray);
[height, width] = size(image);
img_len = height * width;
% 图像预处理（将宽高补足成8的倍数）、-128、分块、DCT和量化
img_quantified = block_dct_quant_zig(image,QTAB);
% 生成随机信息
hid_len = randi(img_len);%定义并记录隐藏信息的长度
hide_info = ceil(rand(hid_len,1)-0.5).';%生成长度为hid_len的随机01序列
hide_seq = [hide_info, zeros(1, img_len - hid_len)];
% 将信息隐藏入img_quantified
hidden = reshape(bitshift(bitshift(img_quantified.', -1, 'int64'), 1, 'int64'), 1, []);
img_quantified = reshape(hidden + hide_seq, [], 64).';
%JPEG编码
blocks = size(img_quantified, 2);
%计算DC码流
DC_coef = img_quantified(1, :);%把各个图像块的量化后的直流分量表示为1个矢量c^D
DC_coef = [DC_coef(1), DC_coef(1:end - 1)- DC_coef(2:end)] ;%差分运算得到c^D
dc_block_process = @(data) DC_process(data, DCTAB);
DC = arrayfun(dc_block_process, DC_coef, 'UniformOutput', false);%Uniform 输出中不能存在非标量值
DC = cell2mat(DC);%把cell转换成mat,计算出DC码流
%计算AC码流
AC_coef = img_quantified(2:end, :);%Zig-Zag扫描形成列矢量去掉DC系数 63*315
AC = AC_process(AC_coef, ACTAB, blocks);
% JPEG解码
col = ceil(height / 8) * ceil(width / 8);
img_quantified = zeros(64, col);
% 解码出DC系数
```

命令行窗口

```
方法1压缩比为: 3.5676
方法1在JPEG编解码后的正确率: 1.000000
方法1的PSNR = 28.6246dB
```

由上图，方法 1 所有隐藏信息都被保留下来了，即正确率为 100%。
而图像的质量和压缩比明显下降（这与其隐藏信息的大小较大也有关系），人眼看起来有一些棋盘的感觉，所以可见其嵌密的隐蔽性并不好，易被发现。
第二种方法:

```
clear;close all;clc;
load('data/hall.mat');
load('data/JpegCoeff.mat');
image = double(hall_gray);
[height, width] = size(image);
img_len = height * width;
% 图像预处理（将宽高补足成8的倍数）、-128、分块、DCT和量化
img_quantified = block_dct_quant_zig(image,QTAB);
% 生成随机信息
% 生成随机起始点
start = randi(img_len);
% 根据起始点生成长度
hid_len = randi(img_len-start);%定义并记录隐藏信息的长度
hide_info = ceil(rand(hid_len,1)-0.5).';%生成长度为hid_len的随机01序列

% 将信息隐藏入img_quantified, 根据start和seq_len变更中间的若干系数
hidden = reshape(img_quantified.', 1, []);
hidden(start: start + hid_len - 1) =  bitshift(bitshift(hidden(start: start + hid_len - 1), -1, 'int64'), 1,
img_quantified = reshape(hidden, [], 64).';
%JPEG编码
blocks = size(img_quantified, 2);
%计算DC码流
DC_coef = img_quantified(1, :);%把各个图像块的量化后的直流分量表示为1个矢量c~D
DC_coef = [DC_coef(1), DC_coef(1:end - 1)- DC_coef(2:end)] ;%差分运算得到c^D
dc_block_process = @(data) DC_process(data, DCTAB);
DC = arrayfun(dc_block_process, DC_coef, 'UniformOutput', false);%Uniform 输出中不能存在非标量值
DC = cell2mat(DC);%把cell转换成mat,计算出DC码流
%计算AC码流
AC_coef = img_quantified(2:end, :);%Zig-Zag扫描形成列矢量去掉DC系数 63*315
```

令行窗口

```
方法2压缩比为: 5.9566
方法2在JPEG编解码后的正确率: 1.000000
方法2的PSNR = 33.2989dB
```

原图片

隐藏信息后的编解码图片

第三种方法:

```
img_quantified = block_dct_quant_zig(image,QTAB);
% 生成随机信息
[qh,qw] = size(img_quantified);
hid_len = randi(qw);%定义并记录隐藏信息的长度
hide_info = ceil(rand(hid_len,1)-0.5).';%生成长度为hid_len的随机01序列
hide_info = hide_info - ~hide_info;%把01序列改成1 -1序列
% 将信息隐藏入img_quantified
for b = 1:hid_len
    index = find(img_quantified(:, b));%找所有非零系数的位置
    if isempty(index)
    %没有非零系数，就替换掉第一个系数
        img_quantified(1, b) = hide_info(b);
    elseif index(end) == 64
    %找到的最后一个非零系数是最后一个系数，即最后一个系数不为0
        img_quantified(64, b) = hide_info(b);
    else
    %追加在最后一个非零系数的后面
        img_quantified(index(end) + 1, b) = hide_info(b);
    end
end
% JPEG编码
```

```
51      % 将隐藏信息抽取出来
52      % 将隐藏信息抽取出来
53 —    extract_info = zeros(1, hid_len);
54 —    for b = 1:hid_len
55 —        index = find(img_quantified(:, b));
56 —        extract_info(b) = img_quantified(index(end), b);
57 —    end
```

命令行窗口

```
方法3压缩比为: 6.2983
方法3在JPEG编解码后的正确率: 1.000000
方法3的PSNR = 34.1510dB
```

原图片

隐藏信息后的编解码图片

可以看到，方法 2 和 3 也实现了正确率为 1，且因为添加信息较少，基本不会影响高频系数，故压缩比与未添加隐藏信息时差不多。同时看出 PSNR 比方法 1 大，即图像的质量和压缩下降得较少（这也与方法 2 和方法 3 的隐藏信息量少有关），方法 2 和方法 3 的隐蔽性相对较好，单凭肉眼基本上看不出图像与正常编解码图像之间的差异。压缩比和 PSNR 也和此前未隐藏信息的结果类似。
但方法 2 的最大信息容量比方法 1 少（和 start 有关），而方法 3 则更少（一个块只能隐藏 1 位信息）。
为方便对比，在隐藏信息大小相同（168）时执行:
方法 1 压缩比为: 6.1959
方法 1 在 JPEG 编解码后的正确率: 1.000000
方法 1 的 PSNR = 32.4381dB
方法 2 压缩比为: 6.2119
方法 2 在 JPEG 编解码后的正确率: 1.000000
方法 2 的 PSNR = 34.0739dB
方法 3 压缩比为: 6.2983
方法 3 在 JPEG 编解码后的正确率: 1.000000
方法 3 的 PSNR = 34.1510dB
可见在压缩比上方法 3 略大于方法 2 略大于方法 1，方法 1PSNR 最大，其次是方法 3，最后是方法 2

四、人脸识别

（1）（a）在生成颜色分布直方图时利用了所有像素，选取的特征是整个样本图像各颜色占据其图像大小的比例，得到的结果与图像大小无关。故不需要调整图片大小，可以直接用于

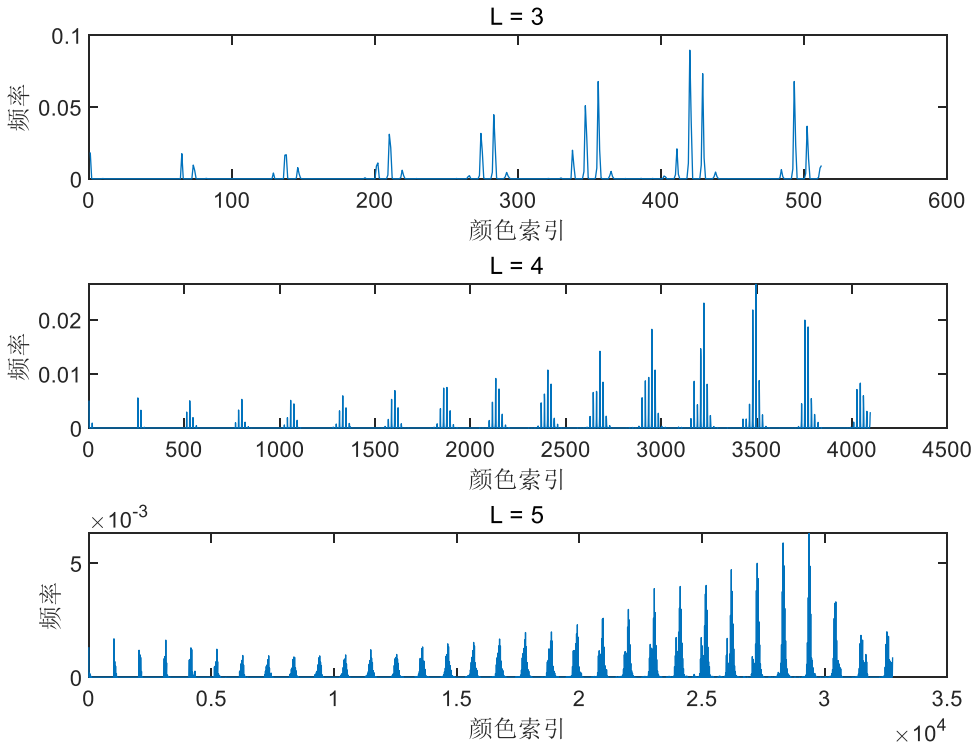
训练和检测。

(b) 用 dir 函数可大批量读取一个文件夹内的所有图片信息

根据 u(R)和 v 的定义，先封装一个生成一张图的颜色比例向量 u(R)的函数 img2color_vec，再封装一个生成训练集内所有图片的人脸标准 v 的函数 color_histogram，再依次对 L=3,4,5 求 v 即可

```
function color_vec = img2color_vec(img, L)
%将图像根据L转变成颜色比例向量v的函数，L就是选取二进制高L位
%   img:图像
%   L:选取二进制高L位
    color_vec = zeros(2^(3 * L), 1);
    % 抽取前L位，因为后续还需要向左移位，所以延展位数
    %image大小即图片大小
    image = double(bitshift(img, -(8 - L)));
    color_list = bitshift(image(:, :, 1), 2 * L) + bitshift(image(:, :, 2), L) + image(:, :, 3);
    % 展平成1维数组，方便后续计算
    color_list = color_list(:);%每一个像素的（R<<2L)+(G<<L)+(B)
    %计算dirac(P_xy-c(n))
    %color_vec(4)对应n=3,color_vec(4)=sum(color_list==3)
    %即n=color_list(xiangsu),应该放到color_vec(n+1),因length(color_list)<<n，用像素来循环效率远高于n循环
    for xiangsu=1:length(color_list)
        n = color_list(xiangsu);
        color_vec(n + 1) = color_vec(n + 1) + 1;
    end
    % 转换成比例
    color_vec = color_vec / length(color_list);%sum(color_vec)=1,color_list表示像素点
end
```

```
function face_template = color_histogram(L)
%从训练集中根据L得到人脸模板，需将样本置于同目录的data/Faces文件夹下
% L:选取二进制高L位
    face_template = zeros(2^(3 * L), 1);
    % 将图片读取进来并得到人脸标准
    img_folder = 'data/Faces/';
    img_list = dir(img_folder);
    % img_list前两个是"."和"..", 所以计数从3开始
    for img_name=3:length(img_list)
        image = imread([img_folder, img_list(img_name).name]);
        %所有图片的颜色向量u(R)累加，512*1
        face_template = face_template + img2color_vec(image, L);
    end
    % img_list前两个是"."和"..", 所以除以长度-2
    face_template = face_template / (length(img_list) - 2);
end
```



从上图可看出，L 越大，颜色划分越细密，所以人脸的颜色分辨率就越高，判断的精度越大。

（2）把人脸识别封装为函数 humanface_check 方便调用。实现原理是先根据相关参数加载待测图像并计算部分后续需要使用的参数，然后使用两重循环用探测窗遍历待测图像，计算探测窗区域的颜色向量与人脸模板的距离，根据距离判别阈值决定是否纳入考虑。纳入考虑的则考察其周围是否已存在识别框，若存在还要根据距离大小进行去重操作。完成探测后，根据探测结果为待测图像加上红框后返回加框图像。

设计时遇到了四个难点：

一是如何遍历图像去找每一个区域的颜色向量来和人脸模板的颜色向量算距离，在 matlab 论坛上看到可以通过滑动识别窗口的方式来实现（借鉴参考了 https://www.ilovematlab.com/thread-490354-1-1.html）；

二是要调试好探测窗口隔多少距离移动检测一次，过大框会偏移（原因是遗漏掉一些区域），过小执行速度慢；

三是要防止识别框的移动超过了图像的边界，我的设计是[h_select,w_select]表示框的左上角。以宽的移动 w_select 为例，w_select 的移动是每次增加 w_step 的，范围是从 1 到图像宽度 width,在最后一次移动中若 w_select+识别框的宽度>图像宽度，就得把 w_select 往左移到临界点而不能直接加 w_step;

四是识别框去重的问题，刚开始没意识到要去重，每次出来的图像都有重叠的识别框。然后参考了 https://blog.csdn.net/qq_33801763/article/details/77184657 中的分类器排序，先创建一个代表像素点的矩阵 dis_record,把每个识别框算出来的距离储存在这个识别框的左上角的像素点对应的 dis_record 的点上，为了方便运用 find 函数（找非零值）且考虑到一般识别框不可能与人脸模板完全一样即距离不可能为 0，所以初始化 dis_record 全 0。然后每次循环时找到识别框重叠范围内所有识别框的距离的最小值，再与该识别框比较，若该识别框的距离较小，则重叠范围内的所有识别框在 dis_record 对应点的距离置为 0，否则把该识别框对应 dis_record 的点置为 0；但因为没想到怎么直接取最小值，所以还是又用了个循环来

实现，也是导致速度较慢的原因。然后取 dis_record 的非零值代表的识别框画红框即可。
非原创等级：借鉴参考
相关代码如下：

```
% 读取待测图像
[h_test, w_test, ~] = size(test_img);
boxes = zeros(h_test, w_test);
% 计算探测窗口在两个方向上滑动的步长
h_step = floor(detect_window(1) / 5);
w_step = floor(detect_window(2) / 5);
% 识别框去重范围
h_dup = floor(0.8 * detect_window(1));
w_dup = floor(0.8 * detect_window(2));
% 探测窗口开始扫描待测图像
for w=1:w_step:w_test
    % 防止识别框超过待测图像边界
    if w > w_test - detect_window(2)
        w_select = w_test - detect_window(2) + 1;
    else
        w_select = w;
    end
    for h=1:h_step:h_test
        % 防止识别框超过待测图像边界
        if h > h_test - detect_window(1)
            h_select = h_test - detect_window(1) + 1;
        else
            h_select = h;
        end
        % 获得探测窗口中的颜色比例矢量并计算与人脸模板之间的距离
        detect_color_vec = img2color_vec(test_img(h_select:(h_select + detect_window(1) - 1), w_select:(w_se
        d = distance(detect_color_vec, face_template);
        % 根据阈值判断是否需要纳入考虑
        if d < epsilon
            if boxes((h_select - h_dup):(h_select + h_dup),(w_select - w_dup):(w_select + w_dup)) == zeros(1
                boxes(h_select, w_select) = d;
```

原始待测图像

L=3的人脸检测结果

L=4的人脸检测结果

L=5的人脸检测结果

可以看到，L 取 3,4,5 时，除了肤色过黑或过白的人脸无法识别出来以外（这是颜色直方图方法的局限性，由于训练集中缺少相关样本，故难以将其检测出），其他的人脸均能够正常识别，总体效果都是非常好的。直观感觉 L=5 的检测结果最好，检测出来的图框最精确。同时，因为手挨得过近导致范围和颜色与人脸相近，所以把手也判成了人脸，这点是因为选取的图片不太适合，可忽略。

同时，设计的算法还有计算速度慢、图框偏移等缺陷，暂时没想到完善的方法。

（3）在上一节中，L=3 的效果已经不错，下面仍然使用 L=3 进行实验。使用以下函数对图像进行调整和检测：

```
%处理图像，操作框也要随着变化
test_img1 = imrotate(test_img, -90);
detect_window1 = fliplr(detect_window);%顺时针旋转90度
test_img2 = imresize(test_img, [size(test_img, 1), size(test_img, 2) * 2]);
detect_window2 = [detect_window(1), detect_window(2) * 2];%宽度拉伸2倍
test_img3 = imadjust(test_img, [0.15 0.15 0; 0.9 0.9 1],[]);%改变颜色
% 分别令L为3,4,5，得到不同的人脸检测结果
test_image1 = humanface_check(test_img1, 3, detect_window1, 0.30);
test_image2 = humanface_check(test_img2, 3, detect_window2, 0.30);
test_image3 = humanface_check(test_img3, 3, detect_window, 0.30);
test_image4 = humanface_check(test_img, 3, detect_window, 0.30);
```

原始L=3检测结果

旋转90度的检测结果

宽度拉伸两倍的检测结果

改变颜色的检测结果

从颜色分布直方图检测法的原理来看，旋转必然不会影响检测结果（因为脸部颜色直方图），而拉伸可能会由于插值方式的不同带来些微的影响，颜色调整则必然会带来影响。从上图也可以看到，旋转和拉伸后的结果与（2）一样。

与之相比改变颜色则少检测出四个人。应该是颜色的些微变化导致球员的脸部颜色直方图发生移位，导致最后检测结果出现变化。

（4）从前面的实验结果来看，这种基于颜色分布直方图的方法虽然有受旋转、拉伸影响小的优点，但是非常受训练样本的影响，导致本实验中没有检测出肤色过黑或者过白的球员。如需进行更为准确的人脸检测，应当考虑针对不同人种训练不同的人脸标准，并在应用时将检测区域与所有样本进行比对，使得不同人种的脸部都能被识别出来。但随着样本的增多，误判的概率会有所上升，应考虑进一步限制阈值以保证准确性。且由于颜色分布直方图的原理仅局限于颜色，不能捕获边缘特征。实际应用时应考虑结合颜色和边缘特征来训练和设定阈值，这样精确度会更高。