MATLAB 高级编程与工程应用

图像处理大作业

BobAnkh

1. 基础知识

(1) 图像处理工具箱

已阅读并了解了 images 工具箱中相关的函数的基本功能

(2) 利用 matlab 提供的函数实现基本图像处理

a) 画红色的圆

根据图片信息确定圆心和半径，然后根据对于 RGB 三色不同色通道的操作实现透明红色圆和实心红色圆。相关代码如下：

```
% 绘制红色圆
hall_circle = hall_color;
[h, w, c] = size(hall_color);
center_x = (w + 1)/2;
center_y = (h + 1)/2;
r = min(h/2, w/2);
[x, y] = meshgrid(1:w, 1:h);
distance = sqrt((x - center_x).^2 + (y - center_y).^2);
circle_area = (distance <= r);

% 该圆透明红色
R = hall_color(:,:,1);
R(circle_area) = 255;
hall_circle(:,:,1) = R;
imwrite(hall_circle, 'hall_transparent_red_circle.jpg');
subplot(2, 2, 2);
imshow(hall_circle);
title("透明红色圆");

% 该圆实心红色
G = hall_color(:,:,2);
G(circle_area) = 0;
hall_circle(:,:,2) = G;
B = hall_color(:,:,3);
B(circle_area) = 0;
hall_circle(:,:,3) = B;
imwrite(hall_circle, 'hall_opaque_red_circle.jpg');
subplot(2, 2, 3);
```

```
imshow(hall_circle);
title("实心红色圆");
```

实际效果如下图（与棋盘格图像绘制在了一起）中所示，达到了预期效果。

b) 画黑白棋盘格

根据设定的格子长宽为图像每个像素点确定其在长宽两个维度上所属的格子标号，以其和的奇偶性判断其是否应该被涂黑。相关代码如下：

```
% 绘制棋盘格，block_w 为单个棋盘格横向宽度，block_h 为单个棋盘格纵向高度，可自由修改
hall_chess = hall_color;
block_w = 21;
block_h = 20;
x = ceil(x / block_w);
y = ceil(y / block_h);
chess_area = logical(mod(x + y +1, 2));
R = hall_color(:,:,1);
R(chess_area) = 0;
hall_chess(:,:,1) = R;
G = hall_color(:,:,2);
G(chess_area) = 0;
hall_chess(:,:,2) = G;
B = hall_color(:,:,3);
B(chess_area) = 0;
hall_chess(:,:,3) = B;
imwrite(hall_chess, 'hall_chess.jpg');
subplot(2, 2, 4);
imshow(hall_chess);
title("棋盘格");
```

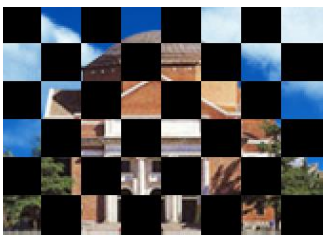
实现效果如下图中所示，达到了预期效果。

原始图像

透明红色圆

实心红色圆

棋盘格



2. 图像压缩编码

(1) 图像预处理

可以在变换域进行。记 DCT 算子为 D（D 矩阵大小为 N×N），最终变换得到的结果为 C，取出来进行变换的图像块为 P，则先将图像每个像素减去 128 再进行 DCT 变换的计算可记为：

$$C = D\left(P - \begin{bmatrix} 128 & \cdots & 128 \\ \vdots & \ddots & \vdots \\ 128 & \cdots & 128 \end{bmatrix}\right)D^T = DPD^T - \begin{bmatrix} 128N & 0_{1\times(N-1)} \\ 0_{(N-1)\times 1} & 0_{(N-1)\times(N-1)} \end{bmatrix}$$

可据此在变换域进行预处理（即将直接进行 DCT 变换结果的矩阵的第一个元素减去 128N）。将直接减去 128 的预处理方式的变换域结果和在变换域做预处理的结果分别打印出来并进行比较。相关代码如下：

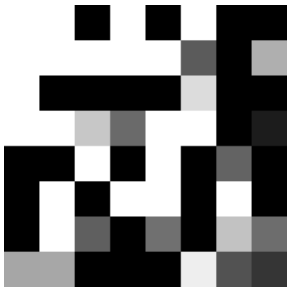
```
hall_select = double(hall_gray(1:8, 1:8));
hall_direct = hall_select - 128;
% 直接减去 128
hall_direct_dct = dct2(hall_direct);
figure;
subplot(2, 1, 1);
imshow(hall_direct_dct);
title("直接减去 128");
disp(hall_direct_dct);

% 在变换域进行
hall_trans = dct2(hall_select);
hall_trans(1, 1) = hall_trans(1, 1) - 128 * size(hall_select, 2);
subplot(2, 1, 2);
imshow(hall_trans);
title("在变换域操作");
disp(hall_trans);
disp(hall_trans - hall_direct_dct);
fprintf("差值最大
值:%.13f\n", max(max(abs(hall_trans - hall_direct_dct))));
```

命令行窗口输出差值绝对值最大值如下：

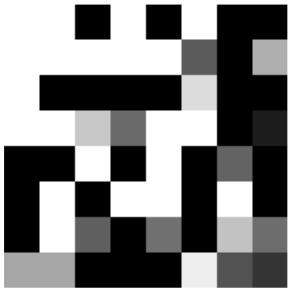
差值最大值:0.0000000000006

可以看到差值绝对值的最大值也小于10^{-12}，由此可以认为两种方法得到的变换域矩阵是相等的。绘制出两个变换域矩阵的图像：

直接减去128



在变换域操作



可以看到两种操作得到的变换域（即经过 DCT 变换后的）结果是一样的。

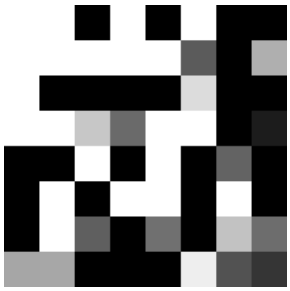
(2) 实现二维 DCT

根据公式实现了 DCT 变换，代码如下：

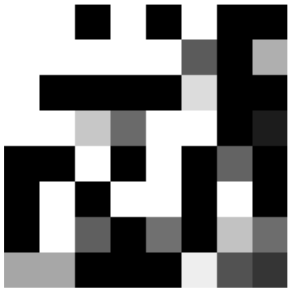
```
function C = mydct2(P)
    % 认为 P 是个方阵
    P = double(P);
    N = size(P, 1);
    D = cos((1:2:(2 * N - 1)) .* (0:1:(N - 1))' * pi / 2 / N);
    D(1, :) = sqrt(0.5);
    D = sqrt(2 / N) * D;
    C = D * P * D';
end
```

从原始图像中取出一块 8×8 大小的块，减去 128 后，分别用我自己实现的 mydct2 函数和 matlab 提供的 dct2 函数，做 DCT 变换，将两者的变换结果作差后取出误差绝对值的最大值，发现小于10^{-12}，由此可以认为两种变换的结果是一样的。将两种变换结果绘制成图像如下：

自行实现的DCT变换



MATLAB提供的DCT变换



可以看到两种 DCT 变换的结果是一样的

(3) DCT 系数矩阵部分列置零

为了方便预处理，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，并减去 128，相关代码如下：

```
function image_processed = preprocess(image)
%prepocess 预处理图像，将宽高补足成 8 的倍数并减去 128
%   image:原始图像
image_processed = int16(image);
[h, w] = size(image_processed);
a = mod(w, 8);
b = mod(h, 8);
if a ~= 0
    col_block = zeros(h, 8 - a);
    col = image_processed(:, w);
    for num=1:(8 - a)
        col_block(:, num) = col;
    end
    image_processed = [image_processed col_block];
end
[h, w] = size(image_processed);
if b ~= 0
```

```
    row_block = zeros(8 - b, w);
    row = image_processed(h, :);
    for num=1:(8 - b)
        row_block(num, :) = row;
    end
    image_processed = [image_processed; row_block];
end
image_processed = image_processed - 128;
end
```

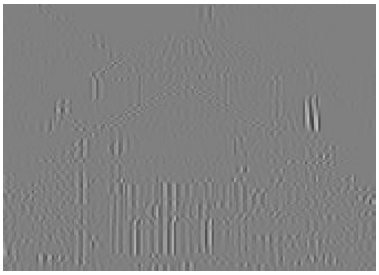
分别对 DCT 系数矩阵不处理、右侧 4 列置零、左侧 4 列置零，还原得到的图像如下：

未经任何处理的原图像

不置零变换的图像

右侧4列置零变换的图像

左侧4列置零变换的图像



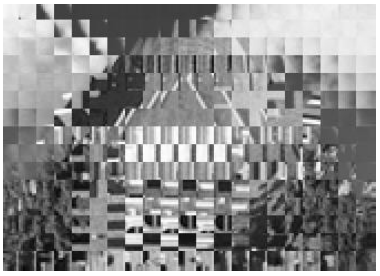
可以看到，不置零的图像和原图基本没有差别；而右侧 4 列置零的图像基本形貌仍然保留了下来，虽然与不置零以及原图相比起来，细节部分变模糊了（分辨率下降了）；而左侧 4 列置零的图像则基本已经难以分辨出原始图像了，只剩下一个大致轮廓了。这也是可以理解的，因为 DCT 系数矩阵左侧是直流和低频分量，而右侧是高频分量，低频分量是人认知图像的主要信息来源，高频分量则是明暗变化剧烈的边界，抹除低频分量就是剩下明暗变化剧烈的边界，而抹除高频分量就是模糊明暗变化剧烈的边界。

(4) DCT 系数矩阵做变换

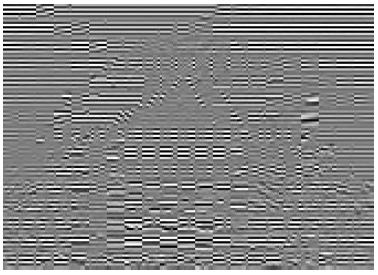
分别对每个 8×8 的系数矩阵做不同变换，然后还原得到如下结果：

未经任何处理的原图像

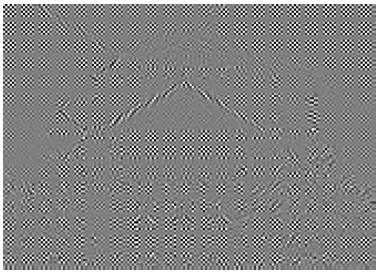
DCT系数转置的图像



DCT系数旋转90°的图像



DCT系数旋转180°的图像



这几种变换得到的图像与原图其实都有较大的差别，图像轮廓都还能勉强辨认出来。对于每个 8×8 块的系数矩阵转置再还原的结果其实就是讲原图的每一块进行转置后拼接在了一起，所以大体轮廓还在，但是细节难以辨识。而系数矩阵旋转 90° 是的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；而旋转 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。

(5) 差分编码

根据差分编码给出该系统的差分方程：

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

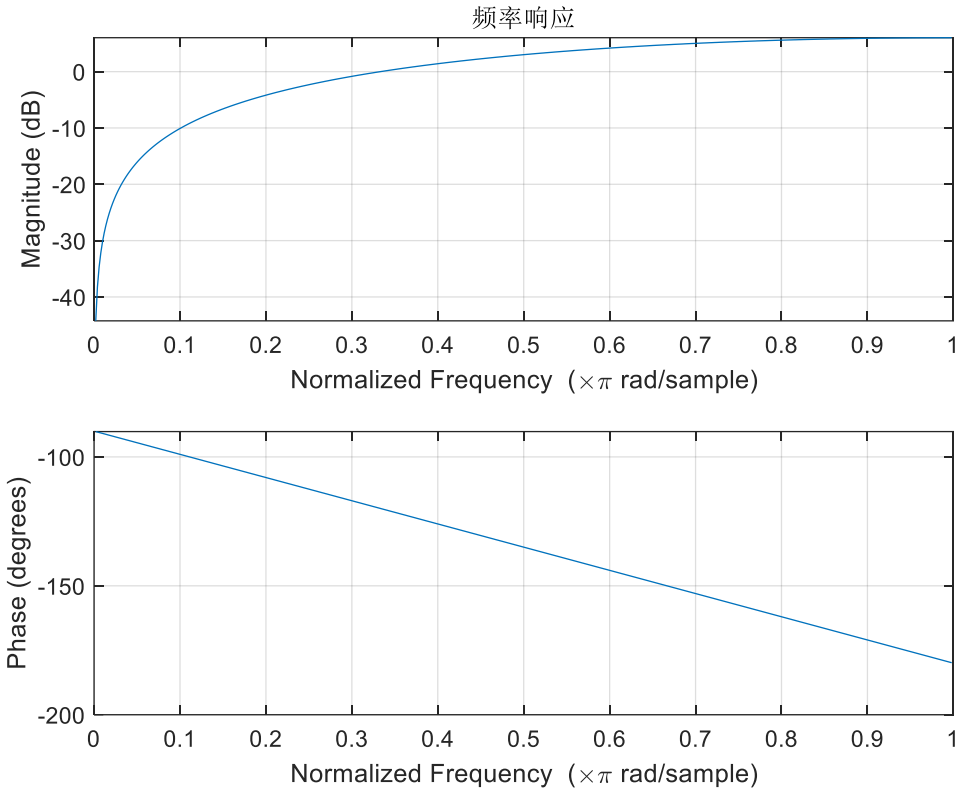
由此得到其系统函数为：

$$H(z) = z^{-1} - 1$$

则可用如下代码绘制出该系统的频率响应：

```
b = [-1, 1];
a = 1;
figure;
freqz(b, a);
title("频率响应");
```

频率响应如下：



由频率响应可知，这是一个高通滤波器。

DC 系数先进行差分编码再进行熵编码这说明 DC 系数的低频分量更多（所以希望用高通滤波器滤除）

(6) DC 预测误差的取值

根据表格，若记预测误差为 Error，则 Category 可由下式求得：

$$\text{Category} = \lceil log_2(|Error| + 1) \rceil$$

(7) Zig-Zag 扫描

针对本大作业独特情况，对于 8x8 分块采用更高效的索引法来实现 Zig-Zag 扫描，同时也给出更通用但是效率相对差一些的循环法 Zig-Zag 扫描。相关代码如下：

```
function vector = myzigzag(matrix)
%myzigzag 实现 zigzag 扫描
%    matrix:需要进行 zigzag 扫描的矩阵
[h, w] = size(matrix);
% 针对本大作业独特情况，8x8 分块采用更高效的索引法
if h == 8 && w == 8
    vector = reshape(matrix', 1, 64);
    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];
    vector = vector(index);
% 也给出更通用但是效率相对差一些的循环法
else
    row = 1;
    col = 1;
    pos = 1;
    % 方向，第一维表示纵向，第二维表示横向
    direction = [-1, 1];
    vector = zeros(1, h * w);
    while( row <= h && col <= w)
        vector(pos) = matrix(row, col);
        if isequal(direction, [-1, 1]) && row == 1 && col ~= w
            direction = [0, 1];
            col = col + 1;
        elseif isequal(direction, [-1, 1]) && col == w
            direction = [1, 0];
            row = row + 1;
        elseif isequal(direction, [-1, 1]) && row ~= 1 && col ~= w
            row = row - 1;
            col = col + 1;
        elseif isequal(direction, [1, -1]) && row ~= h && col == 1
            direction = [1, 0];
            row = row + 1;
        elseif isequal(direction, [1, -1]) && row == h
            direction = [0, 1];
            col = col + 1;
        elseif isequal(direction, [1, -1]) && col ~= 1 && row ~= h
            row = row + 1;
            col = col - 1;
        elseif isequal(direction, [1, 0]) && col == 1
            direction = [-1, 1];
            row = row - 1;
            col = col + 1;
        elseif isequal(direction, [0, 1]) && row == 1
            direction = [1, -1];
            row = row + 1;
            col = col - 1;
        elseif isequal(direction, [1, 0]) && col == w
            direction = [1, -1];
```

```
            row = row + 1;
            col = col - 1;
        elseif isequal(direction, [0, 1]) && row == h
            direction = [-1, 1];
            row = row - 1;
            col = col + 1;
        end
        pos = pos + 1;
    end
end
end
```

(8)量化

封装了一个函数 quantify，用于实现分块、DCT、量化、zigzag 扫描功能。相关代码如下：

```
function output = quantify(image, QTAB)
%quantify 实现分块、DCT、量化、zigzag 功能
%   image:已经经过预处理的图像
%   QTAB:量化系数
P = double(image);
[h, w] = size(P);
output = zeros(64, h * w / 64);
for a = 1:(h / 8)
    for b = 1:(w / 8)
        output(:,(w / 8) * (a - 1) + b) = myzigzag(round(dct2(P((8 *
a - 7):(8 * a), (8 * b - 7):(8 * b))) ./ QTAB)).';
    end
end
```

(9)JPEG 编码

封装了一个函数 JpegEncode，用于进行 Jpeg 编码，根据 DC 系数码本和 AC 系数码本以及量化系数，将原始未经任何处理的图片转化成 DC 码流、AC 码流和宽高，由 hw4_2_9.m 来调用，并将相关数据存入 jpegcodes.mat 中。使用前一问撰写的量化函数将原始图像处理成经过量化后的数据，再分别编写了生成 DC 系数和 AC 系数的子函数，用于生成两个系数。均是采用循环的方法，对于每一个数据进行相应处理。因为一开始并不知道转换出来的系数会有多长，所以采用了一维数组不断扩展的方式来拼接系数，虽然会有一定的性能损失，但是未曾找到更好的方法。相关代码如下：

```
function [DC, AC, height, width] = JpegEncode(image, QTAB, DCTAB, ACT
AB)
%JpegEncode JPEG 编码函数
```

```
%   image:未经任何处理的原始图片
%   QTAB:量化系数
%   DCTAB:DC 系数码本
%   ACTAB:AC 系数码本

[height, width] = size(image);

% 图像预处理与量化
img = preprocess(image);
img_quantified = quantify(img, QTAB);

% DC 系数
dc_coeff = img_quantified(1, :);
% DC 差分编码
dc_diff = [dc_coeff(1), -diff(dc_coeff)];
% DC Huffman 编码
DC = DCHuffman(dc_diff, DCTAB);

% AC Huffman 编码
AC = ACHuffman(img_quantified(2:end, :), ACTAB);

end

function DC = DCHuffman(dc_diff, DCTAB)
    category = ceil(log2(abs(dc_diff)+1));
    DCList = [];
    for a = 1:length(category)
       % category 的 Huffman 编码
       category_huff =  DCTAB(category(a) + 1, 2: (DCTAB(category(a)
+ 1, 1) + 1));
       % magnitude 的编码
       if(dc_diff(a) > 0)
           bin = split(dec2bin(dc_diff(a)), '');
           magnitude = str2double(bin(2:end - 1)).';
       elseif(dc_diff(a) == 0)
           magnitude = [];
       else
           bin = split(dec2bin(-dc_diff(a)), '');
           magnitude = ~str2double(bin(2:end - 1)).';
       end
       DCList = [DCList, category_huff, magnitude];
    end
    DC = num2str(DCList, '%d');
end
```

```
function AC = ACHuffman(ac_coeff, ACTAB)
    ZRL = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1];
    EOB = [1, 0, 1, 0];
    AClist = [];
    for a = 1:size(ac_coeff, 2)
       ac = ac_coeff(:, a).';
       % 非零值的 index
       index = find(ac);
       if(isempty(index))
           AClist = [AClist, EOB];
           continue;
       end
       % 每个非零值对应的游程
       ac_run = [index(1), diff(index)] - 1;
       for b = 1:length(index)
           while(ac_run(b) >= 16)
               AClist = [AClist, ZRL];
               ac_run(b) = ac_run(b) - 16;
           end
           amp = ac(index(b));
           ac_size = ceil(log2(abs(amp)+1));
           % run/size 的 Huffman 编码
           number = find(ACTAB(:,1) == ac_run(b) & ACTAB(:,2) == ac_s
ize);
           rs_huff = ACTAB(number, 4: (ACTAB(number, 3) + 3));
            % amplitude 的编码
           if(amp > 0)
               bin = split(dec2bin(amp), '');
               amplitude = str2double(bin(2:end - 1)).';
           elseif(amp == 0)
               amplitude = [];
           else
               bin = split(dec2bin(-amp), '');
               amplitude = ~str2double(bin(2:end - 1)).';
           end
           AClist = [AClist, rs_huff, amplitude];
       end
       AClist = [AClist, EOB];
    end
    AC = num2str(AClist, '%d');
end
```

hw4_2_9.m 会在命令行窗口输出 DC 码流和 AC 码流的长度，以及图像的原始宽度高度，输出如下：

```
>> hw4_2_9
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) 计算压缩比

根据 hw4_2_9.m 的输出，压缩比为 6.4247，也可以根据 DC 码流和 AC 码流的长度，以及图像的原始宽度高度来进行手动计算：

$$\frac{120 * 168 * 8}{2031 + 23072} = 6.4247$$

(11) JPEG 解码

解码主要是编码的逆过程，同样封装了一个函数 JpegDecode，用于进行 Jpeg 解码，编写了一个反量化函数，用于将量化结果还原回原始图像，反量化函数如下：

```
function image = iquantify(image_quantified, QTAB, height, width)
%iquantify 实现逆 zigzag、反量化、IDCT 和拼接功能
%   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);
for a = 1:h
    for b = 1:w
        img((8 * a - 7):(8 * a), (8 * b - 7):(8 * b)) = idct2(myizigz
ag(image_quantified(:,w  * (a - 1) + b)) .* QTAB);
    end
end
image = img(1:height, 1:width);
```

在 JPEG 解码函数中，也撰写了两个子函数分别用于 DC 系数解码和 AC 系数解码，其中 DC 系数考察到其 Huffman 编码独特的形式（码长 2 的仅有一个，码长超过 3 的第一个 0 都在第三位之后），据此可以无需根据 DCTAB 码表就将 DC 系数解码出。而对于 AC 系数，由于其复杂性，所以只能够根据 ACTAB 码表遍历寻找对应的 run 和 size，从而解出 AC 系数。相关代码如下：

```
function image = JpegDecode(DC, AC, height, width, QTAB, ACTAB)
%JpegDecode
```

```
%    DC: DC 码流
%    AC: AC 码流
%    height: 原始图像高度
%    width: 原始图像宽度
%    QTAB: 量化系数
%    ACTAB: AC 系数码本

col = ceil(height / 8) * ceil(width / 8);
img_quantified = zeros(64, col);
% 解码出 DC 系数
dc_coeff = DCdecode(DC, col);
img_quantified(1,:) = dc_coeff;
% 解码出 AC 系数
ac_coeff = ACdecode(AC, ACTAB, col);
img_quantified(2:64,:) = ac_coeff;
% 反量化
image = iquantify(img_quantified, QTAB, height, width);
image = uint8(image + 128);
end

function dc_coeff = DCdecode(DC, col)

    % 从码流转换成数组
    a = str2double(split(DC, '')).';
    DCList = a(2:end-1);
    dc_coeff = zeros(1, col);
    for b = 1:col
        if(all(DCList(1:2) == 0))
            DCList(1:2) = [];
            % 此时对应 dc_coeff(b)已经是 0 了，无需额外操作
        else
            % 寻找第一个 0 的位置
            index = find(~DCList, 1);
            % 计算 category
            if(index <= 3)
                category = bin2dec(num2str(DCList(1:3), '%d')) - 1;
                DCList(1:3) = [];
            else
                category = index + 2;
                DCList(1:index) = [];
            end
            magnitude = DCList(1:category);
            if(magnitude(1) == 1)
                dc_coeff(b) = bin2dec(num2str(magnitude, '%d'));
```

```
            else
                dc_coeff(b) = -bin2dec(num2str(~magnitude, '%d'));
            end
            DCList(1:category) = [];
        end
    end
    for c = 2:col
        dc_coeff(c) = dc_coeff(c - 1) - dc_coeff(c);
    end
end



function ac_coeff = ACdecode(AC, ACTAB, col)
    ZRL = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1];
    EOB = [1, 0, 1, 0];
    % 从码流转换成数组
    a = str2double(split(AC, '')).';
    ACList = a(2:end-1);
    ac_coeff = zeros(63, col);
    b = 1;
    index = 0;
    while(b <= col)
        if(all(ACList(1:4) == EOB))
            b = b + 1;
            index = 0;
            ACList(1:4) = [];
        elseif(length(ACList) > 11 && all(ACList(1:11) == ZRL))
            index = index + 16;
            ACList(1:11) = [];
        else
            % 取 run 和 size
            for c = 1:size(ACTAB, 1)
                L = ACTAB(c, 3);
                if(L > length(ACList))
                    continue;
                end
                if(all(ACList(1:L) == ACTAB(c, 4:L + 3)))
                    ac_run = ACTAB(c, 1);
                    ac_size = ACTAB(c, 2);
                    ACList(1:L) = [];
                    break;
                end
            end
            index = index + ac_run + 1;
```

```
            amplitude = ACList(1:ac_size);
            if(amplitude(1) == 1)
                ac_coeff(index, b) = bin2dec(num2str(amplitude, '%d')
);
            else
                ac_coeff(index, b) = -
bin2dec(num2str(~amplitude, '%d'));
            end
            ACList(1:ac_size) = [];
        end
    end
end
```

在 hw2_2_11.m 中将其解码出，得到还原图片，并将其存为 hall_jpeg_decode.jpeg，图像如下：

原始图像

解码还原图像

可以看出，主体上没什么区别，不过在细节之处，解码还原的图像会变得有些模糊，比如树叶、屋顶的纹理，而且在一些明暗变化很大的交界处，可以明显看到出现了一些噪点，图片质量在主观上感觉比原始图片略有下降，但是变化都是比较细微的（不仔细看差别并不是很明显），基本上完全可以接受。在 hw2_2_11.m 中将 PSNR 算出，输出到命令行窗口中，结果如下：

```
>> hw4_2_11
PSNR = 34.8926dB
```

可以看到失真较小，编解码效果比较好，带来的图像质量损失较小

(12) 量化步长减小为一半

将量化步长减小一半，重新进行 JPEG 编码和解码，得到的图像如下：

原始图像

解码还原图像

可以看到，解码还原得到的图像和原始图像可以说基本上是一样的，难以看出什么差异。

在代码中计算出压缩比和 PSNR，输出到命令行窗口如下：

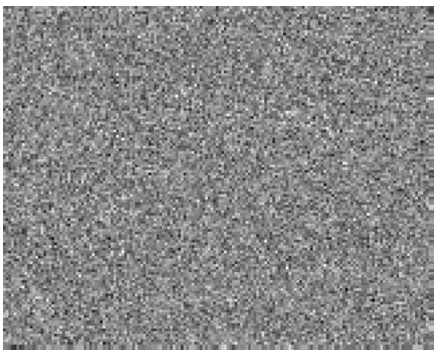
```
>> hw4_2_12
DC码流长度:2410, AC码流长度:34164, 高度:120, 宽度:168
压缩比为: 4.4097
PSNR = 37.3208dB
```

由此可以看到，PSNR 增大了，这说明失真进一步减小，图像质量有所提高，这与直观观察的感受是一致的，而压缩比也降低了，这也就说明，图像质量是以压缩比为代价的——量化系数减半会使得 AC 和 DC 系数变长，从而使得压缩比变小。在实际应用中，采用标准量化步长也是一个比较合适的中庸选择，同时也可以根据实际情况自己选择合适的量化系数来达到需要的目的。

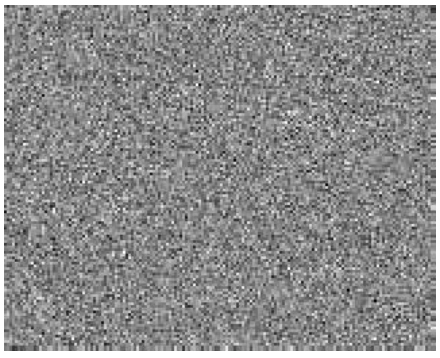
(13) 雪花图像

将雪花图像进行编码解码，得到如下图像：

原始图像



解码还原图像



可以看到解码还原的图像仍然是雪花图像，但是与原始的图像存在不少细节上的差异。

在代码中计算出压缩比和 PSNR，输出到命令行窗口如下：

```
>> hw4_2_13
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
```

可以看到，PSNR 比起之前略有下降，但是压缩比下降的就比较多了，我认为应该是雪花图像相对接近随机图像，而且高频分量比较多，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常规图片的，是考虑将高频分量较多地滤除，高频分量量化系数较大，所以得到这样的一个结果。

3. 信息隐藏

(1) 空域隐藏和提取方法

随机生成一个 01 序列，并补 0 到与图片数据点个数相同的长度。嵌入隐藏信息后，进行 JPEG 编码和解码，相关代码如下：

```
clear all;
close all;

load hall.mat;
load JpegCoeff.mat;
image = double(hall_gray);
height = size(image, 1);
width = size(image , 2);
max_len = height * width;


% 随机生成一个需要隐藏的序列
seq_len = randi(max_len);
hide_info = randi([0, 1], 1, seq_len);
hide_seq = dec2bin([hide_info, zeros(1, max_len - seq_len)]);
% 隐藏信息
P = dec2bin(image.');
P(:, end) = hide_seq;
% 还原回图片
img = reshape(bin2dec(P), width, height).';
C = dec2bin(img.');
% 将隐藏信息抽取出来
extract_info = bin2dec(C(1:seq_len, end)).';
correct = length(find(extract_info==hide_info));
correct_rate = correct / seq_len;
fprintf("只隐藏信息的正确率: %f\n", correct_rate);
```

```
figure;
subplot(1, 3, 1);
imshow(hall_gray);
title("原图片");
subplot(1, 3, 2);
imshow(uint8(img));
title("隐藏信息后的图片");

% JPEG 编码和解码
[DC, AC, height, width] = JpegEncode(img, QTAB, DCTAB, ACTAB);
img = JpegDecode(DC, AC, height, width, QTAB, ACTAB);
subplot(1, 3, 3);
imshow(uint8(img));
title("JPEG 编码解码后的图片");
C = dec2bin(img.');
% 将隐藏信息抽取出来
extract_info = bin2dec(C(1:seq_len, end)).';
correct = length(find(extract_info==hide_info));
correct_rate = correct / seq_len;
fprintf("JPEG 编码解码后的正确率: %f\n", correct_rate);
```

得到图像如下：

原图片

隐藏信息后的图片

JPEG编码解码后的图片

可以看到图像与之前的编解码结果没有什么明显差异，隐藏信息后直接将信息抽出也是100%正确率的，但是编码解码后将信息抽取出来，计算正确率并输出至命令行窗口，得到如下结果：

```
>> hw4_3_1
只隐藏信息的正确率: 1.000000
JPEG编码解码后的正确率: 0.491692
```

因为信息是随机生成的，所以我又多试验了几次，发现基本上正确率都在 50%上下波动，所以可以基本认为解码后信息的正确率在 50%左右，这个正确率是很低的，不能够接受的，基本上可以认为又随机生成了一个信息段，也说明这种方法抗 JPEG 编码的能力是很弱的，并不适于当前互联网环境下使用。

(2) 变换域隐藏和提取方法

注：变换域隐藏方法的隐藏信息由于是我随机生成的，同时因为不同方法对于隐藏信息的长度限制不同，所以每种方法都是独立随机生成隐藏信息的，故而只能大略地进行互相比较

a) 第一种隐藏方法

利用如下代码段生成随机信息：

```
% 生成随机信息
h = size(img_quantified, 1);
w = size(img_quantified, 2);
max_len = h * w;
seq_len = randi(max_len);
hide_info = randi([0, 1], 1, seq_len);
hide_seq = [hide_info, zeros(1, max_len - seq_len)];
```

利用如下代码段隐藏信息：

```
% 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).';
```

利用如下代码段将隐藏信息抽出

```
% 将隐藏信息抽取出来
extract_seq = mod(reshape(img_quantified.', 1, []), 2);
extract_info = extract_seq(1:seq_len);
```

将添加了隐藏信息的量化数据进行 JPEG 编码和解码，得到图像如下：

并且将编解码前后的正确率计算出，与压缩比和 PSNR 一同输出到命令行窗口：

```
>> hw4_3_2_1
方法1只隐藏信息的正确率：1.000000
方法1压缩比为：3.7149
方法1在JPEG编解码后的正确率：1.000000
方法1的PSNR = 28.7627dB
```

可以看到，该方法由于是在量化之后再变换域做的操作，所以所有隐藏信息都被保留下来了，即正确率均为 100%，同时可以看到，图像的质量和压缩比明显下降（这或许与隐藏信息量较多也有一定关系），人眼看起来有一些类似棋盘格样子的感觉，所以可见其嵌密的隐蔽性并不好，容易被发现。

b) 第二种隐藏方法

利用如下代码段生成随机信息：

```
% 生成随机信息
h = size(img_quantified, 1);
w = size(img_quantified, 2);
max_len = h * w;
% 生成随机起始点
start = randi(max_len);
% 根据起始点生成长度
seq_len = randi(max_len - start);
hide_info = randi([0, 1], 1, seq_len);
```

利用如下代码段隐藏信息：

```
% 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 + seq_len - 1) =  bitshift(bitshift(hidden(start:
 start + seq_len - 1), -1, 'int64'), 1, 'int64')+ hide_info;
img_quantified = reshape(hidden, [], 64).';
```

利用如下代码段将隐藏信息抽出

```
% 将隐藏信息抽取出来
extract_seq = reshape(img_quantified.', 1, []);
extract_info = mod(extract_seq(start: start + seq_len - 1), 2);
```

将添加了隐藏信息的量化数据进行 JPEG 编码和解码，得到图像如下：

并且将编解码前后的正确率计算出，与压缩比和 PSNR 一同输出到命令行窗口：

```
>> hw4_3_2_2
方法2只隐藏信息的正确率：1.000000
方法2压缩比为：6.2529
方法2在JPEG编解码后的正确率：1.000000
方法2的PSNR = 34.1487dB
```

可以看到，该方法由于是在量化之后再变换域做的操作，所以所有隐藏信息都被保留下来了，即正确率均为 100%，同时可以看到，图像的质量和压缩下降得较少（这或许也与隐藏信息量相对较少一些有关），人眼看起来就是比原始图像部分细节有些模糊，边界上有明显的渐变，也就是说，如果粗看可能隐蔽性还是较好的，但是如果查看细节就有可能发现。压缩比和 PSNR 也和此前未隐藏信息的结果类似。本方法的优点就在于信息容量和隐藏区间比较自由，比较适合灵活多变的场景。

c) 第三种隐藏方法

利用如下代码段生成随机信息：

```
% 生成随机信息
h = size(img_quantified, 1);
w = size(img_quantified, 2);
max_len = h * w;
seq_len = randi(w);
hide_info = randi([0, 1], 1, seq_len);
hide_info = hide_info - ~hide_info;
```

利用如下代码段隐藏信息：

```
% 将信息隐藏入 img_quantified
for b = 1:seq_len
```

```
    index = find(img_quantified(:, b));
    if isempty(index)
        img_quantified(1, b) = hide_info(b);
    elseif index(end) == 64
        img_quantified(64, b) = hide_info(b);
    else
        img_quantified(index(end) + 1, b) = hide_info(b);
    end
end
```

利用如下代码段将隐藏信息抽出

```
% 将隐藏信息抽取出来
extract_info = zeros(1, seq_len);
for b = 1:seq_len
    index = find(img_quantified(:, b));
    extract_info(b) = img_quantified(index(end), b);
end
```

将添加了隐藏信息的量化数据进行 JPEG 编码和解码，得到图像如下：

并且将编解码前后的正确率计算出，与压缩比和 PSNR 一同输出到命令行窗口：

```
>> hw4_3_2_3
方法3只隐藏信息的正确率：1.000000
方法3压缩比为：6.2239
方法3在JPEG编解码后的正确率：1.000000
方法3的PSNR = 33.5524dB
```

可以看到，该方法由于是在量化之后再变换域做的操作，所以所有隐藏信息都被保留下来了，即正确率均为 100%，同时可以看到，图像的质量和压缩下降得较少（这或许也与隐

藏信息量相对较少一些有关），本方法的隐蔽性相对较好，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。不过，本方法最大的缺点就是最大信息容量比较前两种方法要低不少。

4. 人脸检测

(1) 人脸标准 v

a) 虽然样本人脸大小不一，但是并不需要将图像调整为相同大小，因为采用的是基于颜色直方图的人脸检测方法，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。

b) 直观上我们可以认识到，不同 L 会影响到颜色空间聚集区域的数量，换言之，就是会影响到人脸标准的颜色分辨率。L 越大，颜色划分越细致，得到的人脸标准也越长，也相对越精确。因而 L 小的人脸标准 v 可以由 L 大的人脸标准 v 得到，即加和部分颜色索引点对应的值。取 L 对应的人脸标准 v 中特定位置的 8 个点可以得到 L-1 对应的人脸标准 v 中对应的 1 个点。例如：L=3 的人脸标准中的第 1 个点的值，可以由 L=4 的人脸标准中的第 1,2,17,18,257,258,273,274 个点的值求和得到。

封装了一个函数用于将图片根据 L 值转换成不同长度的颜色向量：

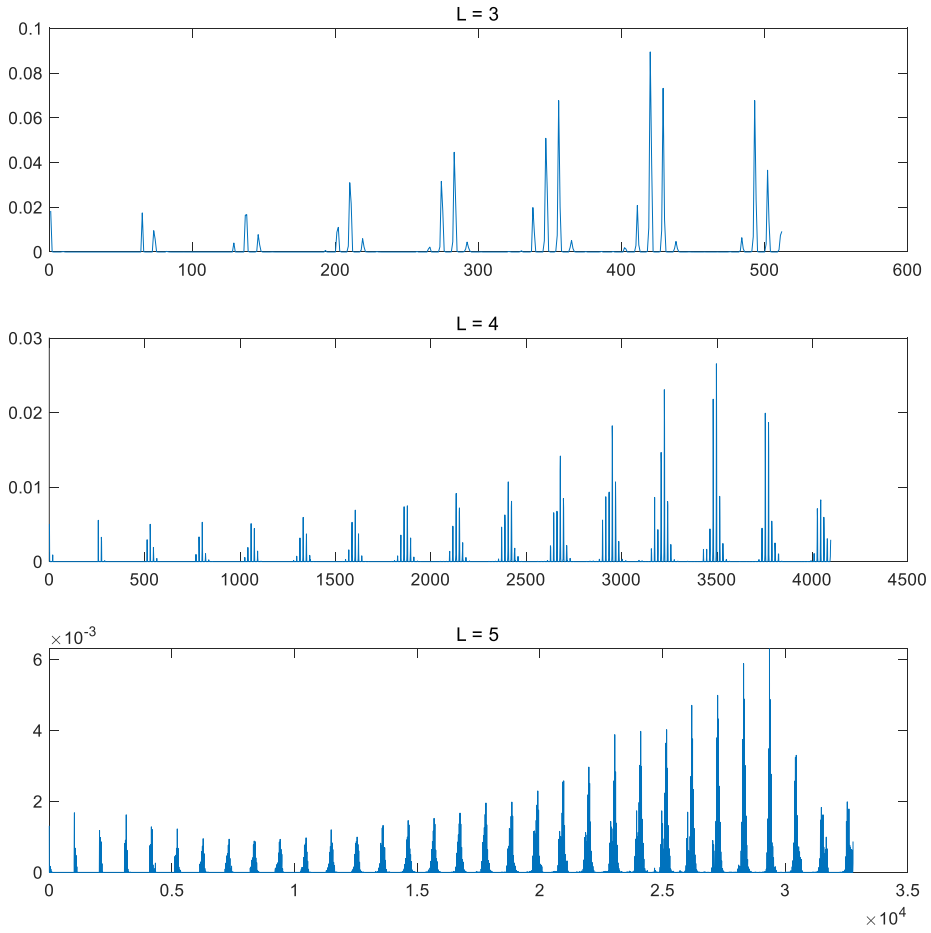
```
function color_vec = img2color_vec(img, L)
%将图像根据 L 转变成颜色比例向量的函数，L 就是选取二进制高 L 位
%    img:图像
%    L:选取二进制高 L 位
    color_vec = zeros(2^(3 * L), 1);
    % 抽取前 L 位，因为后续还需要向左移位，所以延展位数
    image = double(bitshift(img, -(8 - L)));
    color_list = bitshift(image(:, :, 1), 2 * L) + bitshift(image(:,
:, 2), L) + image(:, :, 3);
    % 展平成 1 维数组，方便后续计算
    color_list = color_list(:);
    for color=1:length(color_list)
        color_vec(color_list(color) + 1) = color_vec(color_list(color
) + 1) + 1;
    end
    % 转换成比例
    color_vec = color_vec / length(color_list);
end
```

封装了一个函数用于根据人脸图片训练集生成人脸模板：

```
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 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 = '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]);
        face_template = face_template + img2color_vec(image, L);
    end
    face_template = face_template / (length(img_list) - 2);
end
```

分别令 L 为 3,4,5，得到不同的人脸模板结果，绘图如下：



可以看到，确实是 L 越大的模板长度越长，同时分辨率越高

(2) 实现人脸检测

因为考虑到后续仍需多次使用人脸检测功能，所以同样封装为一个函数方便复用。该函数需要的参数为：待测图像矩阵，L 的值，探测窗的大小（根据不同的识别情况和实际图片的大小需要调整不同大小的探测窗），距离判别阈值（据此判断是否需要将某个探测区域纳入考虑）。

本函数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，若存在则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。相关代码如下：

```
function test_image = face_detect(test_img,L, detect_window, epsilon)
%face_detect 人脸识别用函数，输出加识别框后的图片
%   test_img:需要被人脸检测的图像矩阵
%   L:选取二进制高 L 位
%   detect_window:探测窗大小
%   epsilon:不同的距离判别阈值

% 得到人脸标准
face_template = color_histogram(L);

% 识别图像
% 读取待测图像
test_image = test_img;
[h_test, w_test, ~] = size(test_image);
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_image(h_select:(h_selec
t + detect_window(1) - 1), w_select:(w_select + detect_window(2) - 1)
, :), L);
        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 + 2 * h_dup, 1 + 2 * w_dup)
                boxes(h_select, w_select) = d;
            else
                % 识别框去重
                [dup_row, dup_col, dup_d] = find(boxes((h_select - h_
dup):(h_select + h_dup),(w_select - w_dup):(w_select + w_dup)));
                for num=1:length(dup_row)
                    if d < dup_d(num)
                        boxes(dup_row(num) - 1 + h_select - h_dup,dup
_col(num) - 1 + w_select - w_dup) = 0;
                        boxes(h_select, w_select) = d;
                    else
                        boxes(h_select, w_select) = 0;
                        break
                    end
                end
            end
        end
    end
end

% 画红框
[row, col] = find(boxes);
red_horizontal = 255 * ones(1, detect_window(2));
red_horizontal(:, :, 2) = zeros(1, detect_window(2));
red_horizontal(:, :, 3) = zeros(1, detect_window(2));
red_vertical = 255 * ones(detect_window(1), 1);
red_vertical(:, :, 2) = zeros(detect_window(1), 1);
red_vertical(:, :, 3) = zeros(detect_window(1), 1);
for num=1:length(row)
    test_image(row(num), col(num):(col(num) + detect_window(2) - 1),
:) = red_horizontal;
```

```
    test_image(row(num) + detect_window(1) - 1, col(num):(col(num) +
detect_window(2) - 1), :) = red_horizontal;
    test_image(row(num):(row(num) + detect_window(1) - 1), col(num),
:) = red_vertical;
    test_image(row(num):(row(num) + detect_window(1) - 1), col(num) +
 detect_window(2) - 1, :) = red_vertical;
end
end

% 计算区域特征与人脸模板的相近程度（即距离）
function dis = distance(test, template)
    dis = 1 - sum(sqrt(test .* template));
end
```

在网上随便寻找了一张足球比赛的合照进行人脸检测，根据图片大小选择了[45, 30]大小的识别框，分别令 L 为 3,4,5，各选择距离判别阈值为 0.3, 0.4,0.6（该参数也是经过几次简单的尝试后，发现相对效果较好而确定的），得到不同的人脸检测结果，保存并绘图如下：

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 取各值时，除了一位肤色较深的球员的人脸无法识别出来以外，其他的人脸均能够正常识别，总体效果都是非常好的，识别也较为精准，根据图框位置直观上感觉 L=5 的检测结果最好。而关于肤色较深球员的人脸无法得到识别的问题，我认为这也是颜色直方图方法的局限性，由于训练集中缺少肤色较深人脸的样本，故难以将其检测出也是可以理解的。

(3)进行不同处理后进行人脸检测

由上题可以看到 L=3 的情况效果已经很好了，为了方便，本题就选择 L=3 的情况来进行，同时距离标准阈值也不改动，仍为 0.30。分别对图像做如下操作（同时对探测窗也做相应变换）后，观察人脸检测的结果。

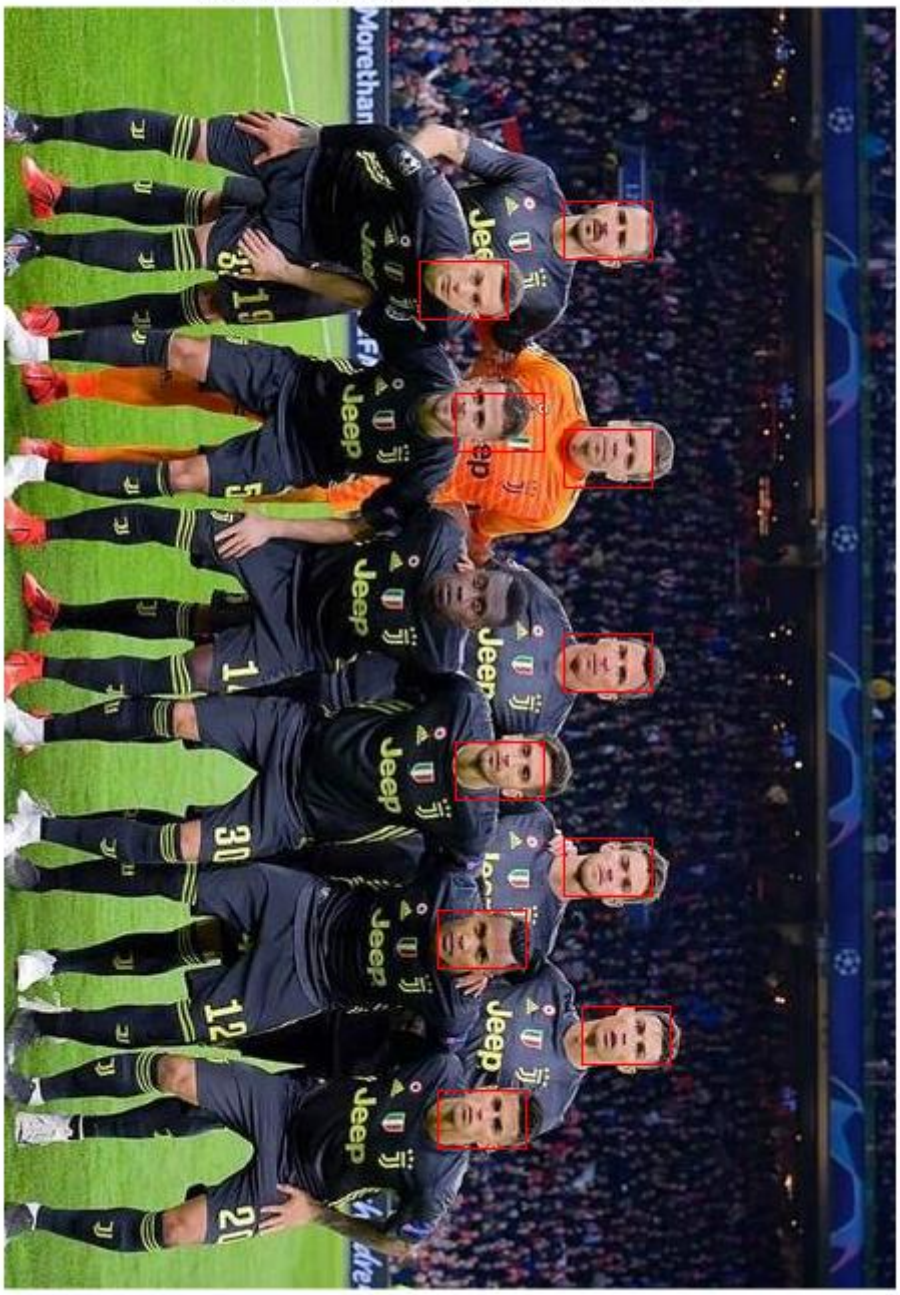
a) 顺时针旋转 90°(imrotate)

用 imrotate 函数将图像顺时针旋转 90°，同时将探测窗的宽高也交换，相关代码如下：

```
test_img_rotate = imrotate(test_img, -90);
detect_window_rotate = fliplr(detect_window);
```

检测结果如下：

顺时针旋转90°的人脸检测结果



可以看到，检测效果是良好的，和原图没有差别，第(2)题中能够检测出来的人脸在本变换下依然能够检测出来。这也是因为旋转对于像素数值可以说是没有本质的变化。

b) 高度不变宽度拉伸为原来的两倍(imresize)

用 imresize 函数将图像保持高度不变而宽度拉伸为原来的两倍，同时将探测窗的高度保持不变而宽度也变为原先的两倍，相关代码如下：

```
test_img_resize = imresize(test_img, [size(test_img, 1), size(test_im
g, 2) * 2]);
detect_window_resize = [detect_window(1), detect_window(2) * 2];
```

检测结果如下：

高度不变宽度拉伸的人脸检测结果

可以看到，检测效果是良好的，和原图没有差别，第(2)题中能够检测出来的人脸在本变换下依然能够检测出来。说明本人脸检测算法应对拉伸的效果较为良好。

c)　适当改变颜色(imadjust)

用 imadjust 函数将调整图像，但调整的幅度不太大，同时将探测窗的宽高均保持不变，相关代码如下：

```
test_img_adjust = imadjust(test_img, [0.15, 0.15, 0; 0.90, 0.90, 1]);
detect_window_adjust = detect_window;
```

检测结果如下：

适当改变颜色的人脸检测结果

可以看到，在这样适度的颜色调整下，检测效果尚可，和第(2)题中能够检测出来的人脸相比，少了一张可以检测出的人脸。这样的结果也是可以想象的，因为颜色改变就改变了本身的颜色向量，与模板的匹配程度自然就不同了，而且可以预料如果颜色改变幅度更大的情

况下可能检测效果更差，这也是颜色直方图算法的局限性——其对颜色变化比较敏感。

(4) 人脸样本训练标准选取

可以从上述情况中看出，颜色直方图的方法有其局限性，其很大程度受到训练集颜色分布的影响，比如本作业提供的样本中不包含深色皮肤人群的人脸，所以对于我寻找的足球比赛合照中的深色皮肤的人脸无法检测出，同时，该方法对于颜色变化较为敏感。对此，我认为，如果能够重新选取人脸样本训练标准，首先需要扩充训练集，需要包含更多类型的样本，同时应当对样本进行聚类（即应该使用多个人脸模板），否则如果将各种肤色的人脸混合成一个人脸模板可能会导致匹配效果较差。同时应该考虑提取其他特征配合匹配，比如面部的边缘特征

5. 文件附录清单

hw4_1_2.m	第一部分第(2)问的主调程序
hw4_2_1.m	第二部分第(1)问的主调程序
hw4_2_2.m	第二部分第(2)问的主调程序
hw4_2_3.m	第二部分第(3)问的主调程序
hw4_2_4.m	第二部分第(4)问的主调程序
hw4_2_5.m	第二部分第(5)问的主调程序
hw4_2_8.m	第二部分第(8)问的主调程序
hw4_2_9.m	第二部分第(9)问的主调程序
hw4_2_11.m	第二部分第(11)问的主调程序
hw4_2_12.m	第二部分第(12)问的主调程序
hw4_2_13.m	第二部分第(13)问的主调程序
hw4_3_1.m	第三部分第(1)问的主调程序
hw4_3_2_1.m	第三部分第(2)问中第一种方法的主调程序
hw4_3_2_2.m	第三部分第(2)问中第二种方法的主调程序
hw4_3_2_3.m	第三部分第(2)问中第三种方法的主调程序
hw4_4_1.m	第四部分第(1)问的主调程序
hw4_4_2.m	第四部分第(2)问的主调程序
hw4_4_3.m	第四部分第(3)问的主调程序
mydct2.m	自己写的 DCT 变换函数
myidct2.m	自己写的 DCT 逆变换函数
myzigzag.m	自己写的用于 zigzag 扫描的函数
myizigzag.m	自己写的用于逆 zigzag 扫描的函数

quantify.m	自己写的用于将原始图片处理得到量化后结果的函数
iquantify.m	自己写的用于将量化后结果还原会原始图片的函数
JpegDecode.m	自己写的用于 JPEG 解码的函数
JpegEncode.m	自己写的用于 JPEG 编码的函数
preprocess.m	自己写的预处理函数，主要将宽高延拓为 8 的倍数并减去 128
img2color_vec.m	第四部分需要用到的转换成颜色比例向量的函数
color_histogram.m	第四部分需要用到的生成人脸模板的函数
face_detect.m	第四部分需要用到的人脸检测的函数
hall.mat	大作业提供的 hall 图片的数据文件
snow.mat	大作业提供的雪花图片的数据文件
JpegCoeff.mat	大作业提供的包含各种参数的数据文件
jpegcodes.mat	第二部分第(9)问 JPEG 编码时存入相关数据供解码用的数据文件
Faces 文件夹	大作业提供的人脸样本训练集的文件夹
football.jpeg	我挑选的用于人脸检测的图片
save_pictures 文件夹	本文件夹中包含各题生成的图片： hall_color.jpg 原始的 hall 的有色图像（由 hw4_1_2.m 生成）； hall_transparent_red_circle.jpg 加上透明红色圆的 hall 图像（由 hw4_1_2.m 生成）； hall_opaque_red_circle.jpg 加上实心红色圆的 hall 图像（由 hw4_1_2.m 生成）； hall_chess.jpg 变成棋盘格形式的 hall 图像（由 hw4_1_2.m 生成）； hall_jpeg_decode.jpeg 由 JPEG 解码得到的还原图像（由 hw4_2_11.m 生成）； face_detect_result_L3.jpg 当 L=3 时的人脸检测结果（由 hw4_4_2.m 生成）； face_detect_result_L4.jpg 当 L=4 时的人脸检测结果（由 hw4_4_2.m 生成）； face_detect_result_L5.jpg 当 L=5 时的人脸检测结果（由 hw4_4_2.m 生成）； face_detect_adjust.jpg 顺时针旋转 90° 的人脸检测结果（由 hw4_4_3.m 生成）； face_detect_resize.jpg 宽度拉伸的人脸检测结果（由 hw4_4_3.m 生成）； face_detect_rotate.jpg 适当颜色调整的人脸检测结果（由 hw4_4_3.m 生成）；