MATLAB 图像处理大作业

马嘉成 2021011966 无18

文件结构和重要文件的简要说明（具体功能描述及变量说明见附件的具体文件）

MATLABWorkspace *MATLAB的工作目录，存放所有涉及到的mlx、m文件和用于训练模型或人脸检测的bmp文件*

1. taskx_y_z.mlx 第x章，第y节，第z号任务
2. ACdecoder.m JPEG交流部分解码器
3. DCdecoder.m JPEG支流部分解码器
4. JPEGdecoder.m JPEG解码器，调用ACdecoder和DCdecoder
5. JPEGencoder.m JPEG编码器
6. DCTconceal1.m 信息隐藏方法一用到的编码器，用信息位逐一替换所有量化后DCT系数最低位，再进行熵编码
7. DCTretrieve1.m 信息隐藏方法一用到的信息提取模块
8. DCTconceal2.m 信息隐藏方法二用到的编码器，用信息位逐一替换每个8x8DCT系数块中(2:4,2:4)的部分，再进行熵编码
9. DCTretrieve2.m 信息隐藏方法二用到的信息提取模块
10. DCTconceal3.m 信息隐藏方法三用到的编码器，用[1 -1]替换zigzag扫描后最后一个非零值的后一位。若最后一位也非0，则直接替换该位
11. DCTretrieve3.m 信息隐藏方法三用到的信息提取模块
12. zigzag88_scan.m 对8x8的矩阵进行Zigzag扫描
13. i_zigzag88_scan.m 将长度为64的向量还原成Zigzag扫描之前的8x8矩阵形式
14. myDCT2.m 我自己的二维DCT变换函数
15. Dmtx.m 根据输入点数N生成对应的DCT算子**D**
16. ImgVecGet.m 计算传入图像在给定L下的特征向量
17. getFaceDist.m 获得Bhattacharyya度量方法下待测图像到标准人脸模型的距离
18. TrainModel.m 根据指定的文件目录和L训练人脸检测标准模型
19. FaceDetection.m 人脸检测器，根据给定的图片和L返回检测到的人脸在图片中的位置及范围
20. DFS.m 在FaceDetection模块中调用，用于对相邻的人脸块进行深度优先搜索，而后合并成完整人脸的范围
21. FrameAdder.m 根据FaceDetection的检测结果，在对应的位置为人脸加框
22. Faces 文件夹存放所有用于训练的bmp格式图片
23. TestImages 文件夹存储用于测试人脸检测算法的bmp格式图片

解答&遇到的问题及解决方案

Task 1.3.2 a

本题首先要获取图像的尺寸

```
% get size
[Height, Width] = size(hall_color, [1 2]);
```

而后以较短边的长度的一半作为半径

```
% get center
h_center = (Height+1)/2;
w_center = (Width+1)/2;
% get radius
r = min(h_center, w_center);
```

为了画圆方便，我获取了各像素点对应的坐标

```
% get index
[W, H] = meshgrid(1:Width, 1:Height);
```

提取符合画圆要求的像素点(到中心距离与目标半径相比偏差不超2%)

```
dist = sqrt((H-h_center).^2 + (W-w_center).^2); % Calculate the distance from
each pixel to the center
idx = abs(dist-r)/r < 0.02; % Extract the pixels to be colored
```

利用提取到的坐标idx，对各通道进行修改，最终显示出来

```
% extract channels
red_channel = hall_color(:, :, 1); % Red channel of the image
green_channel = hall_color(:, :, 2); % Green channel of the image
blue_channel = hall_color(:, :, 3); % Blue channel of the image

% draw circle
red_channel(idx) = 255; % Set the circle region to red
green_channel(idx) = 0; % Set the green channel to 0 for the circle region
blue_channel(idx) = 0; % Set the blue channel to 0 for the circle region

% merge
circ_hall = cat(3, red_channel, green_channel, blue_channel);

% Display the image with the red circle
imshow(circ_hall);
% imwrite(circ_hall,'circ_hall.bmp');
```

效果展示

Task 1.3.2 b

沿用上一问种用到的meshgrid

```
% get index
[W, H] = meshgrid(1:Width, 1:Height);
```

可以先将图像坐标切分为24*24的小方块，每块的横纵序号就由`floor((W-1)/24)`与`floor((H-1)/24)`获得。

根据序号奇偶组合可以将方格分为两类，将横纵奇偶不同的方格涂黑

```
w_flag = mod(floor((W-1)/24),2);
h_flag = mod(floor((H-1)/24),2);
mask = xor(w_flag, h_flag); % 若同奇偶则保留，不同则涂黑
```

写入通道并显示

```
% extract channels
red_channel = hall_color(:, :, 1); % Red channel of the image
green_channel = hall_color(:, :, 2); % Green channel of the image
blue_channel = hall_color(:, :, 3); % Blue channel of the image

% draw chessboard
red_channel(mask) = 0;
green_channel(mask) = 0;
blue_channel(mask) = 0;

% merge
chessboard_hall = cat(3, red_channel, green_channel, blue_channel);

% display
imshow(chessboard_hall);
% imwrite(chessboard_hall,'chessboard_hall.bmp','bmp');
```

效果展示

Task 2.4.1

根据DCT变换的线性性，一张图片*A*的DCT变换结果可以拆成一整张灰度为128的图片的DCT变换结果和原图每个像素点灰度减去128后的DCT变换结果的叠加。所以如果要在变换域进行这一操作，只需要创建一张相同大小，灰度均为128的图片*B*，对其进行DCT变换，得到一个DCT变换结果；而后在*A*的DCT变换结果中减去*B*的变换结果即可。下面验证这一结论

```
clear
clc

load('hall.mat');
double_hall_gray = double(hall_gray(1:8, 17:24)); %截取第一行第三块
[h, w] = size(double_hall_gray, [1 2]);
DCTorig = dct2(double_hall_gray-128); % 直接减128再变换

DCTtrans = dct2(double_hall_gray); % 先变换
DCT128 = dct2(zeros(h, w)+128); % 直流128的DCT变换
DCTsub = DCTtrans-DCT128; % 做差，得到频域处理

DCTdiff = DCTsub-DCTorig; % 比较
err = sum(DCTdiff.^2,'all')/h/w; %均方误差MSE
```

MSE = 1.3442e-28，两种方法的结果仅有数值计算的误差。

Task 2.4.2

按照公式

$$\sqrt{\frac{2}{N}}\begin{bmatrix} \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & \cdots & \sqrt{\frac{1}{2}} \\ \cos\frac{\pi}{2N} & \cos\frac{3\pi}{2N} & \cdots & \cos\frac{(2N-1)\pi}{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \cos\frac{(N-1)\pi}{2N} & \cos\frac{(N-1)3\pi}{2N} & \cdots & \cos\frac{(N-1)(2N-1)\pi}{2N} \end{bmatrix}$$

得到DCT算子**D**

```
% @Dmtx.m

function D = Dmtx(N)
% generates D matrix according to input dimension N
[D_temp_X, D_temp_Y] = meshgrid(1:2:(2*N-1),0:(N-1));
D_temp = D_temp_X.*D_temp_Y*pi()/2/N;
D = sqrt(2/N)*cos(D_temp);
D(1,:) = sqrt(1/N);
end
```

根据公式 $C = DPD^T$ 使用得到的DCT算子进行DCT变换

```
function C = myDCT2(img)
[h, w] = size(img,[1 2]);
if(h == w)
    D = Dmtx(h);
    C = D*img*D';
else
    C = Dmtx(h)*img*Dmtx(w)';
end
end
```

以hall_gray为例进行测试

```
clear
clc

load('hall.mat');
double_hall_gray = double(hall_gray) - 128;
[h, w] = size(double_hall_gray, [1 2]);
DCTorig = dct2(double_hall_gray); % 自带DCT变换结果
DCTcust = myDCT2(double_hall_gray); % 我的DCT变换结果
DCTdiff = DCTcust-DCTorig; % 比较 % 做差
err = sum(DCTdiff.^2,'all')/h/w; % 均方误差
```

err(MSE) = 9.7110e-25，我实现的二维DCT变换与自带结果误差很小，功能的正确性得以验证

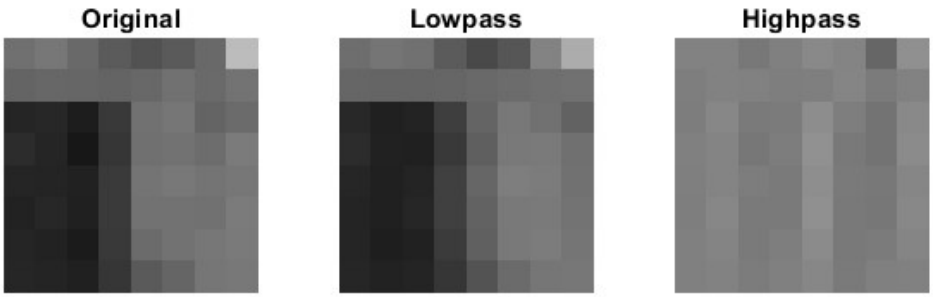
Task 2.4.3

DCT变换矩阵的左上代表低频部分，右下代表高频部分，左下代表图像的垂直高频和水平低频，右上是图像的垂直低频和水平高频部分。

如果将右边4列全部置0，则相当于去除了图像的高频部分和水平高频部分。图像水平方向的过渡会更加平滑，图像会呈现更明显（相对）的水平纹理和更平滑的整体过度。

如果将左边4列全部置0，则图像的高频和水平高频会占主导。可以看到较为明显的竖直条纹，水平纹理减弱。

以下是实际结果



与预测基本相符。第二张图是去掉右4列的结果，可以看出竖直纹理的减弱（中间两列尤其明显）。第三张是去掉左4列的结果，水平纹理明显减弱，基本仅剩下竖直条纹（水平方向的高频变化）

代码见`task2_4_3.mlx`

Task 2.4.4

若将DCT变换矩阵C取转置再做逆变换，得到的图像$P^{'}=D^TC^TD$。又因原图$P=D^TCD$，所以有

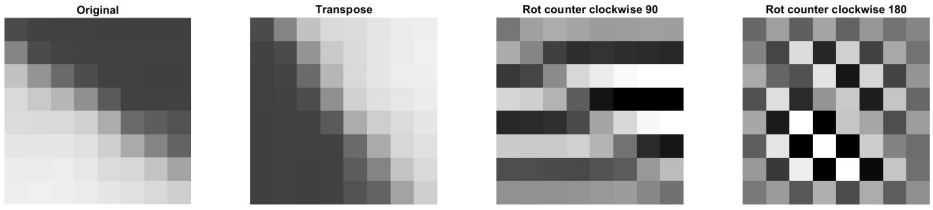
$$P^{'T}=(C^TD)^TD=D^TCD=P$$

即$P^{'}=P^T$，得到的新图像是原图的转置。

对于一张一般的图像，低频分量高于高频分量，这也是为什么JPEG压缩可以牺牲一部分高频但仍保持较好的画质。

如果将图像的DCT变换结果逆时针旋转90°，则较大的低频分量系数会被旋转到左下角，即竖直高频与水平低频部分。所以这时得到的新图像会呈现出较剧烈的竖直变化（水平纹理）。

如果将DCT变换结果旋转180°，则较大的系数会出现在右下角的高频区，不论横竖方向都会有较明显的变化，图像近乎马赛克。



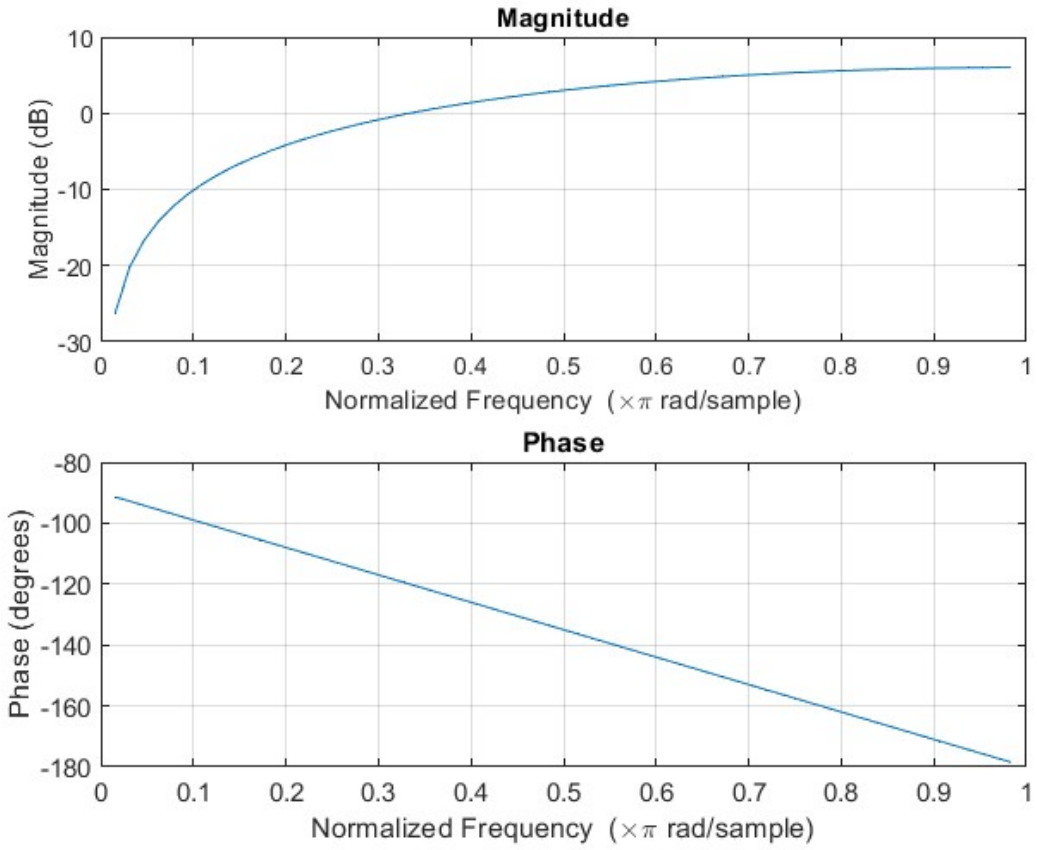
经检验，图二确实是原图的转置，图三对应DCT逆时针转90°，图四对应图像转180°，基本符合预期。

Task 2.4.5

使用 `freqz` 函数获得数字滤波器的频率响应

```
freqz([-1 1], 1, 64);
```

得以下结果



显然这是一个**高通**滤波器，可以推测DC系数**高频**分量更多

Task 2.4.6

预测误差取值为x
则Category值为$\lceil log_2(|x|+1) \rceil$，即用来表示x所需的二进制位数
数值较小的对应huffman编码也短，说明日常大多数图像各相邻区块间往往直流分量相差不大

Task 2.4.7

由于只用处理8x8矩阵得Zigzag遍历，所以直接将元素序号按照对8x8矩阵Zigzag扫描的顺序写成一向量形式即可

```
function outMtx = zigzag88_scan(inMtx)
    outMtx = inMtx([ 1, 9, 2, 3,10,17,25,18, ...
                    11, 4, 5,12,19,26,33,41, ...
                    34,27,20,13, 6, 7,14,21, ...
                    28,35,42,49,57,50,43,36, ...
                    29,22,15, 8,16,23,30,37, ...
                    44,51,58,59,52,45,38,31, ...
                    24,32,39,46,53,60,61,54, ...
                    47,40,48,55,62,63,56,64]);
end
```

对应的，将向量按照逆Zigzag恢复成矩阵也可以靠直接排列下标实现

```
function mtxOut = i_zigzag88_scan(mtxIn)
    mtxOut = mtxIn([
          [ 1, 3, 4,10,11,21,22,36];
          [ 2, 5, 9,12,20,23,35,37];
          [ 6, 8,13,19,24,34,38,49];
          [ 7,14,18,25,33,39,48,50];
          [15,17,26,32,40,47,51,58];
          [16,27,31,41,46,52,57,59];
          [28,30,42,45,53,56,60,63];
          [29,43,44,54,55,61,62,64]
          ]');
end
```

Task 2.4.8

```
clear
clc

load('JpegCoeff.mat');
load('hall.mat');
```

对原图进行分块DCT变换

```
dcthall = blockproc(double(hall_gray)-128,[8 8],@(mat)(dct2(mat.data)));
```

量化

```
roundhall = blockproc(dcthall,[8 8],@(mat)(round(mat.data./QTAB)));
```

对每个8x8块进行Zigzag扫描，

```
zigzaghall = blockproc(roundhall,[8 8],@(mat)(zigzag88_scan(mat.data)));
```

将扫描后的结果合成题目要求的矩阵形式

```
[h, w] = size(zigzaghall,[1 2]);
rslt = [];
for i = 1:h
    for j = 1:64:w
        rslt = [rslt,zigzaghall(i,j:j+63)'];
    end
end
```

Task 2.4.9

首先，同上2.4.8，通过分块DCT变换、量化和分块Zigzag扫描，我得到了有64行的一个矩阵

```
[imgH, imgW] = size(inMat,[1 2]);
load('JpegCoeff.mat','ACTAB','DCTAB','QTAB');
dctMat = blockproc(double(inMat) - 128,[8 8],@(mat)(dct2(mat.data))); % DCT变换
roundMat = blockproc(dctMat,[8 8],@(mat)(round(mat.data./QTAB))); % 量化
zigzagMat = blockproc(roundMat,[8 8],@(mat)(zigzag88_scan(mat.data))); % Zigzag扫
描
[h, w] = size(zigzagMat,[1 2]);
rslt = [];
for i = 1:h % row1(col1 2 3 ......) row2 ......
    for j = 1:64:w
        rslt = [rslt,zigzagMat(i,j:j+63)'];
    end
end
```

分离出DC部分和AC部分

```
DCarray = rslt(1,:);
ACarray = rslt(2:end,:);
```

生成DC码流(重要代码的解释以注释形式给出，下同)

```
% generate DC stream
DCdiff = [2*DCarray(1),DCarray(1:end-1)] - DCarray; % 先做差分
DCcat = min(ceil(log2(abs(DCdiff)+1)),11); % 通过2.4.6中的公式计算各位的Category
DCstream = [];
for i = 1:length(DCcat)
    lenHuffman = DCTAB(DCcat(i)+1,1); % 通过Category定位Huffman编码所在行，提取霍夫曼
码长
    if DCcat(i) ~= 0
        binTemp = dec2bin(abs(DCdiff(i))) - '0'; % 获取二进制码
        if DCdiff(i)<0
            binTemp = ~binTemp; % 如果是负数，按位取反
        end
    else
        binTemp = []; % 如果Category==0，只用写霍夫曼码字，不用写二进制数
```

```
    end
    DCstream = [DCstream,DCTAB(DCcat(i)+1,2:(1+lenHuffman)),binTemp]; % 写入现有码流
end
```

生成AC码流

```
% generate AC stream
ACstream = [];
for j = 1:length(DCarray) % 从左向右逐列编码
    zeroCount = 0; % '0'计数器
    for i = 1:63 % 从上到下编码
        ACnum = ACarray(i,j); % 当前要处理的AC系数
        if ACnum ~= 0
            ZRLcount = floor(zeroCount/16); % how many ZRLs should be added
            Run = mod(zeroCount,16); %run
            Size = ceil(log2(abs(ACnum)+1)); %size
            if(ACnum > 0)
                Amp = dec2bin(ACnum)-'0';
            else
                Amp = ~(dec2bin(-ACnum)-'0');
            end % Amplitude
            ACTABidx = Run*10 + Size; % 计算Run/Size对应的行索引
            ACstream = [ACstream,repmat([1,1,1,1,1,1,1,1,1,0,0,1],[1,ZRLcount]),...
                        ACTAB(ACTABidx,4:(3+ACTAB(ACTABidx,3))),Amp]; % 向原码流后插入ZRL、Huffman码和Amplitude
            zeroCount = 0;
        else
            zeroCount = zeroCount + 1; % 如果当前项为0，计数，不写码流
        end
    end
    ACstream = [ACstream,[1,0,1,0]]; % 最后插入EOB标志块编码结束
end
```

以上代码在`JpegEncoder.m`模块中，在task2_4_9中调用之，得到的码流保存在`jpegcodes.mat`中。以下是调用代码

```
clear
clc

load('hall.mat');
[ACstream,DCstream,h,w] = JpegEncoder(hall_gray);
save('jpegcodes.mat',"DCstream","ACstream","h","w",'-mat');
```

Task 2.4.10

```
clear
clc

load('jpegcodes.mat');
compressrate = 8*h*w/(length(ACstream)+length(DCstream));
```

压缩后AC码流长len_ACstream=23072bits

DC码流长len_DCstream=2031bits

压缩编码码流总长度len_compressed = len_ACstream + len_DCstream = 25103bits

原图像大小size_hall_gray = 120 * 168 * 8 bits = 161280bits

压缩比为size_hall_gray / len_compressed = 6.4247

Task 2.4.11

JPEG解码的DC部分封装为`DCdecoder.m`

```
function DCarray = DCdecoder(DCstream,blockamount)
    load("JpegCoeff.mat","DCTAB");

    DCarray = zeros(1,blockamount); % init DCarray for i_zigzag

    % decode DC
    currDC_idx = 1; % init pointer currDC_idx作为指针指示当前已经解码完的最后一位的下一位
    for j = 1:blockamount
        len_append = 0; % number of bits after currDC_idx pointer 从当前指针又往后看了多少位
        category_idx = 0:(size(DCTAB,1)-1); % 与candidates的行绑定，记录对应行的category
```

每次开始解码一个新的DC系数时，candidates复制整个DCTAB，而后每加入一位，就根据当前前缀码排除candidates中不可能的选项，这样candidates的行数会不断减少。又因为Huffman编码是前缀码，只要编码不出错，一个前缀有且只有一个对应的选项是正确的，而且仅当在码流中取完整个Huffman码字才会得到唯一解，不会提前。所以当candidates的范围逐渐缩小最终到**唯一一行**时，搜索结束，该行对应的Huffman码就是码流中的Huffman码字，从而我们可以看category_idx的对应行得到category，从而进一步解码出magnitude。

```
        candidates = DCTAB; % possible matches
        while size(candidates,1) > 1 % how many possible matches remain? done when only one left
            prefix = DCstream(currDC_idx:currDC_idx+len_append); % current prefix
            choose_idx = []; % choose which rows will remain
            filter = candidates(:,2:2+len_append) == prefix; % which of the candidates match the current prefix?
            for curr_row = 1:size(filter,1)
                if filter(curr_row,:)
                    choose_idx = [choose_idx,curr_row];
                end
            end
            candidates = candidates(choose_idx,:); % extract the rows that still match
            category_idx = category_idx(choose_idx);
            len_append = len_append + 1;
        end
        % should have found one and only one match
        DCcat = category_idx;
        currDC_idx = currDC_idx + len_append; % no need to +1, since an additional 1 has been added in line 25
        if DCcat == 0
```

```
            DCarray(j) = 0; % 如果category为0，则码流后面没有记录magnitude
        else
            mag_bin = DCstream(currDC_idx:currDC_idx + DCcat - 1);
            currDC_idx = currDC_idx + DCcat;
            if mag_bin(1) == 0
                mag_dec = -bin2dec(char(~mag_bin + '0'));
            else
                mag_dec = bin2dec(char(mag_bin + '0'));
            end
            DCarray(j) = mag_dec;
        end
    end
    for i = 2:length(DCarray)
        DCarray(i) = DCarray(i-1)-DCarray(i); % 逆差分过程得到原始DC向量
    end
end
```

JPEG解码的AC部分封装为`ACDecoder.m`。与DC解码类似，AC解码也使用了逐渐缩小候选范围的方式找Huffman码，只是这里将ZRL和EOB也并入了ACTAB的末尾以使程序更加简单

```
function ACarray = ACdecoder(ACstream,blockamount)
    load("JpegCoeff.mat","ACTAB");
    ACarray = zeros(63,blockamount); % init ACarray for i_zigzag
    ZRL = [0,0,0,1,1,1,1,1,1,1,1,0,0,1,0,0,0,0,0];
    EOB = [0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0];
    ACTAB = [ACTAB;ZRL;EOB];

    % decode AC
    currAC_idx = 1;
    for j = 1:blockamount
        decoded_amount = 0; % number of decoded numbers in a column 用来记录该列已经解码的系数数量，达到63说明该列解码完毕
        while decoded_amount < 63
            len_append = 1; % since the shortest huffman code has a length of 2 这里由于Huffman码字2位起步，所以附加长度直接从1开始，写DC部分时没意识到这一点
            candidates = ACTAB;
            while size(candidates,1) > 1 % 同上面DC的道理，利用Huffman前缀码的性质可以如此判断是否识别到一个完整的Huffman码字
                prefix = ACstream(currAC_idx: currAC_idx+len_append);
                % match_condition = candidates(:,4:4+len_append) == prefix;
                choose_idx = [];
                for x = 1:size(candidates,1)
                    if isequal(prefix,candidates(x,4:4+len_append))
                        choose_idx = [choose_idx,x];
                    end
                end
                candidates = candidates(choose_idx,:);
                len_append = len_append+1;
            end
            if isequal(candidates,ZRL)
                decoded_amount = decoded_amount + 16; % 16 zeros
                currAC_idx = currAC_idx + 11; % len huffman ZRL = 11
            elseif isequal(candidates,EOB)
                currAC_idx = currAC_idx + 4; % len huffman EOB = 4
                break
```

```
            else
                % Run = candidates(1);
                % Size = candidates(2);
                % HuffmanLen = candidates(3);
                currAC_idx = currAC_idx + candidates(3); % skip Huffman code
                decoded_amount = decoded_amount + candidates(1); % zeros before
num
                binAmp = ACstream(currAC_idx : currAC_idx + candidates(2) - 1); %
获取Amplitude的二进制码
                if binAmp(1) == 0 % get amplitude
                    Amplitude = -bin2dec(char(~binAmp + '0'));
                else
                    Amplitude = bin2dec(char(binAmp + '0'));
                end
                currAC_idx = currAC_idx + candidates(2);
                decoded_amount = decoded_amount + 1;
                ACarray(decoded_amount,j) = Amplitude;
            end
        end
        if decoded_amount == 63
            currAC_idx = currAC_idx + 4; % if dosen't end with a zero, currAC_idx
should be added with an additional len(EOB)
        end
    end
end
```

一开始忘了讨论不以0结尾的情况，但由于多数照片AC码流都会以0结尾，所以没发现，后来解码雪花时发现有bug才意识到这个问题。

完成这两部分后使用`JPEGDecoder.m`调用它们，分别获得AC码流和DC码流，合并后做反Zigzag，反量化，最后做反DCT得到解码后的最终图像。代码如下

```
function Image = JpegDecoder(DCstream,ACstream,img_h,img_w)
    blockamountW = ceil(img_w/8); % 横向块数
    blockamountH = ceil(img_h/8); % 纵向块数
    blockamount = blockamountH * blockamountW; % 总块数
    DCarray = DCdecoder(DCstream,blockamount); % DC解码
    ACarray = ACdecoder(ACstream,blockamount); % AC解码
    arrayFull = [DCarray;ACarray]; % 拼接成完整的DC、AC矩阵
    iZigzagMtx = zeros(blockamountH,blockamountW);
    for y = 1:blockamountH % 反Zigzag
        startY = (y-1) * 8 + 1;
        for x = 1:blockamountW
            startX = (x-1) * 8 + 1;
            iZigzagMtx(startY:startY + 7,startX:startX + 7) =
i_zigzag88_scan(arrayFull(:,(y-1) * blockamountW + x));
        end
    end
    load('JpegCoeff.mat','QTAB');
    deRoundMat = blockproc(iZigzagMtx,[8 8],@(mat)(mat.data.*QTAB)); % 反量化
    iDCTmtx = blockproc(deRoundMat,[8 8],@(mat)(idct2(mat.data))); % 逆DCT
    Image = uint8(iDCTmtx + 128); % 得到最终图像
end
```

最后完成Task2.4.11内容，先解码之前编码的图片

```
load("jpegcodes.mat");
JpegDecodeImage = JpegDecoder(DCstream,ACstream,h,w);
```

原图与解码结果对比如下

先从主观来看，二者总体差异不明显。但可以在白云或礼堂屋顶和天空过渡处附近看到一些不平滑的噪声，拱门的顶也变得略粗糙，可见还是稍微有些失真的，这主要来源于对高频分量的压制，也有一小部分来源于数值计算误差。

根据公式，峰值信噪比$PSNR = 10 * lg(\frac{255^2}{MSE})$

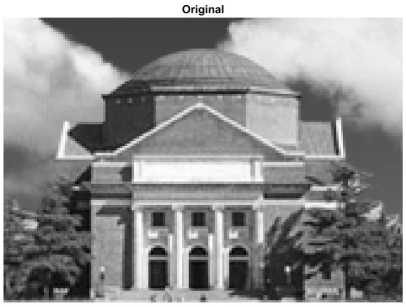
```
MSE = sum((JpegDecodeImage-hall_gray).^2,'all')/h/w;
PSNR = 10 * log10(255^2/MSE);
```

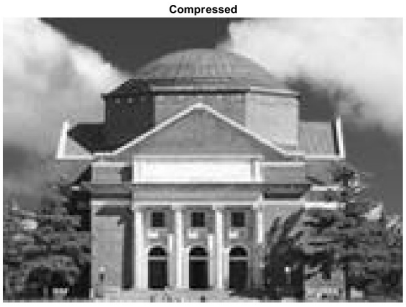
得到PSNR=34.8926，大于30dB，数值较大，失真较小

Task 2.4.12

量化步长缩小为原来一半，只需要对编解码器加入`QTAB = QTAB/2`即可。

得到的结果如下





肉眼观察与原步长解码的结果差别不大，天空与云交界处仍有高频噪声，屋顶纹理较原图更模糊，效果主观上没有实质性改善。

计算PSNR

```
MSE = sum((Image_Decoded-hall_gray).^2,'all')/h/w;
PSNR = 10 * log10(255^2/MSE);
```

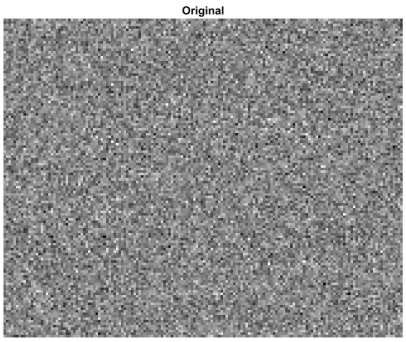
PSNR = 37.2983，比标准步长的34.8926更高，失真更小，但提升不大
计算压缩比

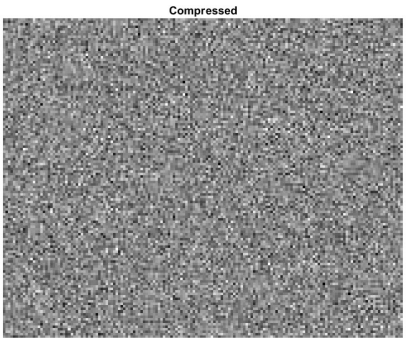
```
compressrate = 8*h*w/(length(ACstream)+length(DCstream));
```

compressrate = 4.4097，压缩比比标准步长之下的6.4247明显更小
这说明实际使用中选用更小的量化步长可能对失真的抑制效果有限，但却严重牺牲了压缩性能，所以原步长在压缩比和抑制失真之间取得了比较好的平衡。

Task 2.4.13

对`snow.mat`进行类似的编解码，得到与原图的对比如下





计算PSNR

```
MSE = sum((Image_Decoded-snow).^2,'all')/h/w;
PSNR = 10 * log10(255^2/MSE);
```

PSNR = 29.5614
由于原图高频分量很多，量化时主要对高频分量进行了压缩，所以从PSNR看图像失真较大。但实际上由于原图杂乱无章，凭肉眼不是很容易分辨两图的差异。
计算压缩比

```
compressrate = 8*h*w/(length(ACstream)+length(DCstream));
```

compressrate = 3.6450
由于高频分量较多，长游程项更多，所以huffman码长较长，且对AC部分编码时EOB出现较晚，所以AC码流显著变长，压缩比较低。

Task 3.4.1

先生成随机的测试数据并对其进行隐藏

```
info = uint8(randi([0 1],h,w)); % 生成测试信息
imageConceal = bitset(hall_gray,1,info); % 空域隐藏
```

经过JPEG编解码得到空域图片

可见信息隐藏对图像的影响较小，不易被发现。

接下来提取信息并计算还原率

```
infoRetrieve = bitget(ImageRslt,1); % 恢复信息
ErrorMtx = bitxor(info,infoRetrieve); % 按位异或得错误
fidelity = 1-sum(ErrorMtx,"all")/h/w; % 还原率
```

我先后测试了五次，得到数据如下

fidelity = 0.5005, 0.5011, 0.4953, 0.5004, 0.4936

还原率在0.5上下，与再随机生成一个还原后得结果计算还原率无异，几乎无法还原任何有用信息

Task 3.4.2.1

用信息位逐一替换所有量化后DCT系数最低位，再进行熵编码。沿用原编码器，但是在量化后插入如下语句（完整代码见DCTconceal1.m）

```
% 信息隐藏部分
concealMat = bitset(int32(roundMat),1,srcInfo);
```

其中`srcInfo`是待隐藏的信息

因为信息是按照8x8的分块DCT系数进行隐藏的，所以在信息恢复部分先对DC与AC分别解码，而后通过反Zigzag扫描将变换域图像恢复成8x8的分块矩阵形式，而后提取对应位的信息（完整代码见DCTretrieve1.m）

```
DCarray = DCdecoder(DCstream,blockamount);
ACarray = ACdecoder(ACstream,blockamount);
arrayFull = [DCarray;ACarray];
iZigzagMtx = zeros(blockamountH,blockamountW);
```

```
for y = 1:blockamountH
    startY = (y-1) * 8 + 1;
    for x = 1:blockamountW
        startX = (x-1) * 8 + 1;
        iZigzagMtx(startY:startY + 7,startX:startX + 7) =
i_zigzag88_scan(arrayFull(:,(y-1) * blockamountW + x));
    end
end
% 信息恢复部分
infoRtv = bitget(int32(iZigzagMtx),1);
% 信息恢复结束
```

信息隐藏后的图像如下

最直观的影响就是高频分量多了，出现了类似马赛克一样的图案。这是由于替换所有DCT系数的最低位时增加了整张图片的高频分量，例

如将原来右下角的0换成了1。另外，经过多次测试，得到PSNR和compressrate的大致范围

PSNR = 28.1473, 28.1461, 28.1667, 28.1531, 28.1576

compressrate = 2.8616, 2.8567, 2.8580, 2.8751, 2.8730

可以明显看出PSNR较低，在28.14左右浮动，图像失真较大。嵌密方法的隐蔽性较差。

而且压缩率也大幅下降，仅为2.86左右，这是因为替换了高频分量的最低位后Zigzag扫描结果每一列的最后一个非0值出现地很晚，而且连续的0也变少了，中间会掺杂较多的非0值，这些导致AC码长较长。

评估信息还原的准确率

```
ErrorMtx = bitxor(info,infoRtv); % 按位异或得错误
fidelity = 1-sum(ErrorMtx,"all")/h/w; % 还原率
```

fidelity = 1, 1, 1, 1, 1

因为量化后嵌入信息避免了除法运算与舍入误差对信息位的影响，这种方法抗JPEG压缩的能力很好，能完全准确地恢复信息。而且这种方法可携带的信息密度大，平均1bit/pixel

Task 3.4.2.2

用信息替换掉若干量化后的DCT系数最低位。我选择替换掉8x8矩阵中(2:4,2:4)的区域的最低位，这是因为(1,1)是量化后的DC系数，如果将其最低位替换掉，编码，再进行解码，经过反量化，会放大对DC系数造成的影响，使相邻块间的亮度不连续。所以为了嵌密的隐蔽性，我不对DC系数进行操作。又因为在上一个方法中对高频分量的更改会严重影响压缩比和隐蔽性；而且左上角的量化步长较小，对量化后的系数做更改对反量化后的值影响较小，所以我选择只对左上角的DCT系数进行操作。信息隐藏的部分代码如下，完整代码见`DCTconceal2.m`

```
% 信息隐藏部分
concealMat = int32(roundMat);
blockamountW = ceil(imgW/8);
blockamountH = ceil(imgH/8);
for i = 1:blockamountH
    yStart = (i-1)*8+1; % 该8x8方阵的起始Y坐标
    infoYstart = (i-1)*3+1; % 该8x8方阵要隐藏的信息的起始位置Y坐标
    for j = 1:blockamountW
        xStart = (j-1)*8+1; % 该8x8方阵的起始X坐标
        infoXstart = (j-1)*3+1; % 该8x8方阵要隐藏的信息的起始位置X坐标
        concealMat(yStart+1:yStart+3,xStart+1:xStart+3) =
bitset(concealMat(yStart+1:yStart+3,xStart+1:xStart+3),1,srcInfo(infoYstart:infoY
start+2,infoXstart:infoXstart+2));
    end
end
% 信息隐藏结束
```

得到如下图像

肉眼可见仍对画质造成了一定的影响，但相比方法1要好一些。

多次测试，得到PSNR和compress rate

PSNR = 33.2751, 33.2779, 33.3021, 33.2821, 33.3764

compressrate = 5.8102, 5.8096, 5.7969, 5.7875, 5.8276

PSNR较高，在33.3左右浮动，说明这种信息隐藏方法对图像的失真影响较小，隐蔽性较好。

而且compress rate较高，在5.8左右浮动。这主要是因为我只在低频部分做了替换，没有修改高频分量。由于低频部分一般非0值比较多，而且我只修改了最低位，所以对Run和Size的影响都较小，从压缩编码的角度与直接压缩差距较小。

同上一种方法，由于在量化后嵌入信息，这种方式的抗JPEG压缩能力很好，几次的还原率都是100%

fidelity = 1, 1, 1, 1, 1

另外，这种方法较上一个方法牺牲了携带信息的密度，平均0.14bit/pixel

Task 3.4.2.3

用[1 -1]替换zigzag扫描后最后一个非零值的后一位。若最后一位也非0，则直接替换该位。

信息隐藏的部分代码如下，在Zigzag后的矩阵中寻找最后一个非0值，在其后面嵌入信息。如果Zigzag序列以非0值结束，则直接替换序列的最后一个元素。完整代码见`DCTconceal3`

```
% 信息隐藏开始
for i = 1:blockamount
    last_notzero = find(rslt(:,i),1,'last'); % 寻找最后一个非0值位置
    if last_notzero == 64
        rslt(64,i) = info(i);
    else
        rslt(last_notzero+1,i) = info(i);
    end
end
% 信息隐藏结束
```

得到的信息隐藏图像如下

肉眼观感比之前的两种方法都好。但由于最后一位非0值可能处于高频位置，在其后插入信息容易引入水平或竖直的高频分量，部分块呈现出了竖条纹或横条纹，影响其隐蔽性。而且从解码器的角度看，解完熵编码就会发现以这种方式隐藏信息的图片其Zigzag序列的结尾永远是1或-1，而且可能在稀疏的高频部分经常有连续两个非0值，且后一个不是1就是-1，很有其特点。如果有针对性地去检查则较易漏出马脚。所以其真实隐蔽性存疑。

PSNR和compress rate情况如下

PSNR = 33.1693, 33.2170, 33.2128, 33.1888, 33.2069

compressrate = 6.1916, 6.1916, 6.1916, 6.1916, 6.1916

PSNR与方法2接近，图像失真较小。

由于固定在最后一个非0值后或在末尾嵌入信息，而且嵌入的信息不是1就是-1，所以不论信息内容如何，对整体的Run和Size的影响是一样的，压缩率不受信息内容影响。

压缩率在三种方法中最高，接近直接压缩的压缩率。这也是因为引入的非0值少，不破坏EOB前0值的连续情况，很多0都可以表示在ZRL和Run中，对EOB的位置影响也很小，所以AC码流长度不会延长很多。

同前两种方法，fidelity = 1, 1, 1, 1, 1，抗JPEG压缩能力很好。但缺点是能嵌入的信息密度更低，只有0.015bit/pixel。

Task 3.4.3 (选做)

之前的信息隐藏方式1、2可以嵌入大量信息，但是从视觉上很容易被发现，隐蔽性不好。方式3则会在量化后的序列中留下比较明显的痕迹，而且引入水平或竖直的高频分量，容易被视觉检查发现。针对这两者的缺点，我提出自己的信息隐藏方法。虽然这种方法携带信息的密度和方法三一样，为0.015bit/pixel，不是很高，但是隐蔽性更强。

原理是用每个8x8量化后DCT系数块的绝对值之和的奇偶表示1bit信息。具体说来，要嵌入的信息是一个由0,1组成的向量，向量长度和将图片切割成8x8的块得到的块数一致。从上到下，从左到右，将系数块按顺序编号，如此和信息向量的元素一一对应。如果某8x8图像块的量化值的绝对值之和为偶，则其表示0，反之如果为奇，则表示1。提取信息时只需要对该块的所有系数求绝对值后求和，判断其奇偶，就可以恢复该bit信息。在`DCTretrieve4.m`中，由这行代码实现

```
infoRtv(i) = mod(sum(abs(arrayFull(:,i)),"all"),2);
```

如果要嵌入信息，则比较该块绝对值之和的奇偶性与该bit信息是否对应，如果绝对值之和为奇而该bit信息就应当是1，或者绝对值之和为偶而该bit信息就应当是0，则不用改变任何系数。如果不一样，则对该块`(2,2)`位置的数据加1或减1，这样便可反转该块绝对值之和的奇偶性，使其与对应的信息一致。部分实现代码如下，详见`DCTconceal4.m`

```
% 信息隐藏开始
for i = 1:blockamount
    temp = mod(sum(abs(rslt(:,i)),"all"),2);
    if temp ~= info(i)
        rslt(5,i) = rslt(5,i) - 1; % zigzag后(2,2)元素的位置
    end
end
% 信息隐藏结束
```

这样做的好处之一是`(2,2)`元素对应的量化步长仅为12，对量化后的系数加减1不会对该分量的幅度产生很大的影响，比较隐蔽。另外，由于该元素本身是低频分量，稍作改变引起的差异在视觉上没有高频分量那么明显，也加强了其隐蔽性。即便是希望通过量化后的序列找出隐藏信息的痕迹，也没有方法3那么容易被发现，因为我不用任何单独的元素或单独的位表示信息。另一个好处是由于修改低频部分几乎不影响Run与Size，所以对码长基本没有影响，这样就保证了图片较高的压缩率。再者，由于它也是在DCT域做信息隐藏，所以有着很好的抗JPEG压缩能力。最后，这种方法很**灵活**，除了通过mod2判断奇偶之外，还**可以衍生出使用mod3、mod4等方法的变种**，在尽量保持图片质量的同时**提高单像素能携带的比特数**，提升信息的传递效率。

实测得到如下对比效果，左侧是不隐藏信息直接进行JPEG压缩得到的，右边是利用我的方法隐藏了信息的图像

二者之间几乎没有任何明显的区别

PSNR = 34.8004, 34.7905, 34,8042, 34.7862, 34.8115

PSNR在34.8上下浮动，和直接压缩的34.8926相比，二者差距很小，和前面的三种方法相比，有较大的优势。

compress rate = 6.3901, 6.3990, 6.3942, 6.3995, 6.39957

接近直接压缩的6.4247，较之前面的三种方法，压缩率有较大优势。

fidelity = 1, 1, 1, 1, 1

抗JPEG压缩能力强

这种方法的缺点是如果只用简单的奇偶携带信息，则只能实现0.015bit/pixel的信息嵌入密度。但如果使用mod3乃至mod4，再将对元素的修改操作比较均匀地分摊给更多低频元素，就可以在尽量保持图像质量的同时提高信息的嵌入密度。

Task 4.3.1

获取图片的特征向量这一功能由以下`ImgVecGet.m`函数实现

```
function ImgVec = ImgVecGet(Img,L)
% 计算传入图像在给定L下的特征向量
    ImgVec = zeros(1,8^L);
```

由L决定要保留多少位，剩下的低位由移位去除

```
    RedIdx = int32(bitsrl(Img(:,:,1),8-L)); % 要保留的红色位，通过移位操作去除多余的位，
下同
    GreenIdx = int32(bitsrl(Img(:,:,2),8-L)); % 要保留的绿色位
    BlueIdx = int32(bitsrl(Img(:,:,3),8-L)); % 要保留的蓝色位
    Idx = bitsll(RedIdx,2*L) + bitsll(GreenIdx,L) + BlueIdx + 1; % Matlab array
starts from 1 计算出各颜色对应的下标
    for i = 1:size(Idx,1)
        for j = Idx(i,:)
        ImgVec(j) = ImgVec(j) + 1; % 统计落在各下标内的颜色们出现了多少次
        end
    end
    ImgVec = ImgVec/size(Img,1)/size(Img,2); % 归一化
end
```

归一化后得到某张照片颜色的分布情况

为了批量处理，我编写了`TrainModel.m`模块用于训练模型。

我把用于训练的文件放入列表，依次计算每个图片的特征向量并累加之，最后求平均，即得到**“平均人脸”**的颜色分布向量

```
for i = 1:size(samplelist,1)
    nameTmp = samplelist(i).name;
    currImg = imread(strcat(directory,'\',nameTmp),'bmp');
    array = array + ImgVecGet(currImg,L);
    clearvars currImg;
end
array = array/size(samplelist,1);
```

训练得到L=3、L=4以及L=5时的模型`v3`,`v4`,`v5`，存储在`FacialRecogModels.mat`中。

a) 并不需要调整人脸大小，因为本识别方法依赖颜色分布，并不提取图像的边缘形态等信息，所以单纯改变图像大小，不改变颜色占比对本训练方法理论上并没有影响。

b) 其实L=X时就是把L=X+1时的8种颜色合并为了一种颜色。对某L=X时的颜色，这8种颜色的二进制序号(从0开始计算)可以由向L=X时某颜色的二进制序号(也从0开始计算)分段后在每段后面补0或1得到，就像下面的程序一样。所以如果对L=X时的所有颜色，我们把其对应的L=X+1时的8种颜色找出来，将8种颜色出现频率相加，理论上应该得到与L=X时训练出的向量完全一致的向量。下面编程验证L=3,4向量之间的关系，L=4,5时同理。

```
test1 = zeros(1,512);
for i = 0:511
    bintmp = dec2bin(i,9);
    seg1 = bintmp(1:3);
    seg2 = bintmp(4:6);
    seg3 = bintmp(7:9);
    idx1 = bin2dec([seg1,'0',seg2,'0',seg3,'0'])+1;
    idx2 = bin2dec([seg1,'0',seg2,'0',seg3,'1'])+1;
    idx3 = bin2dec([seg1,'0',seg2,'1',seg3,'0'])+1;
    idx4 = bin2dec([seg1,'0',seg2,'1',seg3,'1'])+1;
    idx5 = bin2dec([seg1,'1',seg2,'0',seg3,'0'])+1;
    idx6 = bin2dec([seg1,'1',seg2,'0',seg3,'1'])+1;
    idx7 = bin2dec([seg1,'1',seg2,'1',seg3,'0'])+1;
```

```
    idx8 = bin2dec([seg1,'1',seg2,'1',seg3,'1'])+1;
    test1(i+1) = sum(v4([idx1,idx2,idx3,idx4,idx5,idx6,idx7,idx8])); % 将v4中在v3中被合并的颜色的各自占比相加，得到的比例就是合并后v3中对应颜色的占比
end
MSE1 = sum((test1-v3).^2);
```

计算$MSE = 2.4534 * 10^{-33}$，除数值计算误差外两结果完全一样，理论正确。

Task 4.3.2

要检测人脸，我的思路如下：

首先，人脸整体以肉色为主，颜色较为集中。如果将人脸的一部分截取出来，计算它与标准人脸之间的距离，应该不会很远。所以可以先将整张图切成较小的块，对每个块计算距离，这样就可以大致区分出有人脸的块和没有人脸的块。而后将相邻的符合要求的块组合在一起，得出其矩形外包络。这些矩形框起来的地方就是人脸的大致范围。过滤掉过小的矩形，因为那些可能是误判。之后，微调矩形的四条边，使得矩形围出的区域和标准人脸的距离尽可能近，得到最终的人脸识别结果。

计算距离的公式由$d = 1 - \sum_n \sqrt{u_n v_n}$给出，具体实现在`getFaceDist.m`中

分块计算距离时要适当粗略，因为不能保证人脸的一部分和整张人脸的颜色分布是一样的。所以我首先用L=3粗筛，而且距离的阈值也设置地比较宽松

```
% @FaceDetection.m
threshold = 0.6;

load("FacialRecogModels.mat",'v3');
v = v3;

for x = 1:xBlockNum
    startX = (x-1)*blocksize+1;
    for y = 1:yBlockNum
        startY = (y-1)*blocksize+1;
        if getFaceDist(testImg(startY:startY+blocksize-1,startX:startX+blocksize-1,:),v,3) < threshold
            evalMtx(y,x) = 1; % 如果该块距离小于阈值，说明有可能是人脸的一部分，在evalMtx中做记录
        end
    end
end
```

现在已经大致获取了含有人脸的块信息，接下来需要将相邻块的信息汇总起来得到初步的人脸范围，为此我使用了DFS算法，从人脸的一块开始，搜寻周围相邻的人脸块，最终将连通区域找全。

```
global visited
visited = zeros(yBlockNum,xBlockNum);
margins = [];
for x = 1:xBlockNum
    for y = 1:yBlockNum
        if evalMtx(y,x) && visited(y,x) == 0
            [upTmp,downTmp,leftTmp,rightTmp] = DFS(x,y,xBlockNum,yBlockNum); % 深
度优先搜索将零碎的块拼起来
            if downTmp-upTmp > minFaceBlock && rightTmp-leftTmp > minFaceBlock %
过滤掉过小的识别结果
                margins = [margins;(upTmp-1)*blocksize+1,(downTmp-1)*blocksize,
(leftTmp-1)*blocksize+1,(rightTmp-1)*blocksize]; % 记录边缘下标
            end
        end
    end
end
```

现在，我已经初步获得了人脸的范围，接下来就是微调边框位置，使得框住的部分距离最小。我选择了贪心算法，将上下左右边框微调后的距离算出来，看哪个最小就调哪个。

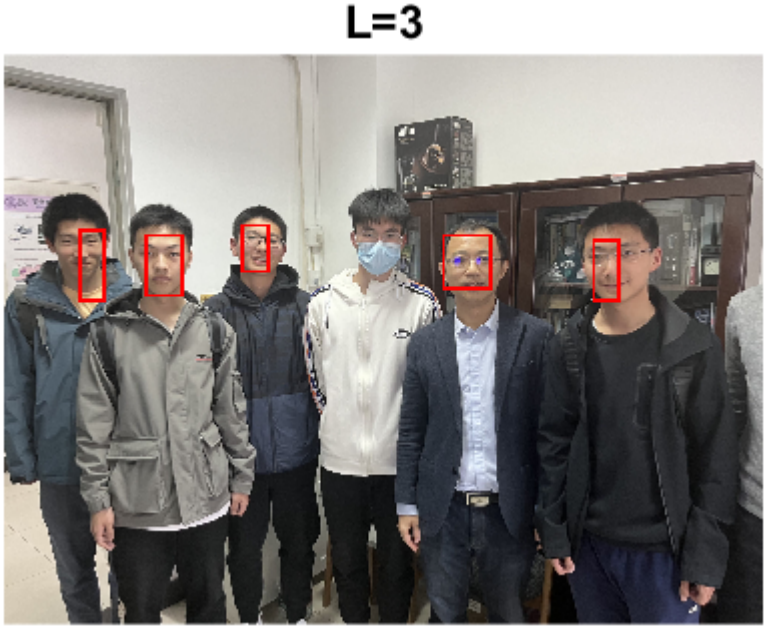
```
plusStep = 8; % 向外延申的步长
subStep = 9; %向内缩的步长，与前者互质，可以多次调整达到最优
for i = 1:size(margins,1) % refinement
    up = margins(i,1); % 上边框y值
    down = margins(i,2); % 下边框y值
    left = margins(i,3); % 左边框x值
    right = margins(i,4); % 右边框x值
    currDist = getFaceDist(testImg(up:down,left:right,:),v,L); %初步得到的人脸距离

    % up refinement
    distTmp = currDist; % distTmp存储试探性微调后当前的最优距离

    upPlusDist = getFaceDist(testImg(max(up-plusStep,1):down,left:right,:),v,L);
% 向外延申一个步长后的新距离
    upSubDist = getFaceDist(testImg(min(up+subStep,down):down,left:right,:),v,L);
% 向内缩一个步长后的新距离
    downPlusDist =
getFaceDist(testImg(up:min(down+plusStep,h),left:right,:),v,L);
    downSubDist = getFaceDist(testImg(up:max(down-subStep,up),left:right,:),v,L);
    leftPlusDist = getFaceDist(testImg(up:down,max(left-
plusStep,1):right,:),v,L);
    leftSubDist =
getFaceDist(testImg(up:down,min(left+subStep,right):right,:),v,L);
    rightPlusDist =
getFaceDist(testImg(up:down,left:min(right+plusStep,w),:),v,L);
    rightSubDist = getFaceDist(testImg(up:down,left:max(right-
subStep,left),:),v,L);
    minDistTmp =
min([distTmp,upPlusDist,upSubDist,downPlusDist,downSubDist,leftPlusDist,leftSubDi
st,rightPlusDist,rightSubDist]); % 当前距离、延申后距离和收缩后距离的最小值
    while minDistTmp ~= distTmp % 如果当前并非最优
        if upPlusDist == minDistTmp % 如果向外伸展更优
            distTmp = upPlusDist; % 更新当前距离
            up = up-plusStep; % 更新上边框y值
```

```
        elseif upSubDist == minDistTmp% 如果向内缩更优
            distTmp = upSubDist; % 更新当前距离
            up = up+subStep; % 更新上边框y值
        elseif downPlusDist == minDistTmp
            distTmp = downPlusDist;
            down = down + plusStep;
        elseif downSubDist == minDistTmp
            distTmp = downSubDist;
            down = down - subStep;
        elseif leftPlusDist == minDistTmp
            distTmp = leftPlusDist;
            left = left - plusStep;
        elseif leftSubDist == minDistTmp
            distTmp = leftSubDist;
            left = left + subStep;
        elseif rightPlusDist == minDistTmp
            distTmp = rightPlusDist;
            left = left + plusStep;
        else
            distTmp = rightSubDist;
            right = right - subStep;
        end
        upPlusDist = getFaceDist(testImg(max(up-
plusStep,1):down,left:right,:),v,L); % 向外延申一个步长后的新距离
        upSubDist =
getFaceDist(testImg(min(up+subStep,down):down,left:right,:),v,L); % 向内缩一个步长后
的新距离
        downPlusDist =
getFaceDist(testImg(up:min(down+plusStep,h),left:right,:),v,L);
        downSubDist = getFaceDist(testImg(up:max(down-
subStep,up),left:right,:),v,L);
        leftPlusDist = getFaceDist(testImg(up:down,max(left-
plusStep,1):right,:),v,L);
        leftSubDist =
getFaceDist(testImg(up:down,min(left+subStep,right):right,:),v,L);
        rightPlusDist =
getFaceDist(testImg(up:down,left:min(right+plusStep,w),:),v,L);
        rightSubDist = getFaceDist(testImg(up:down,left:max(right-
subStep,left),:),v,L);
        minDistTmp =
min([distTmp,upPlusDist,upSubDist,downPlusDist,downSubDist,leftPlusDist,leftSubDi
st,rightPlusDist,rightSubDist]); % 当前距离、延申后距离和收缩后距离的最小值
    end
    margins(i,1) = up;
    margins(i,2) = down;
    margins(i,3) = left;
    margins(i,4) = right;
end
```

使用此方法，取L=3，得到以下结果



除了中间的人脸，其余人脸均被识别出来，实现了人脸检测的功能

在不微调、L=3、L=4，L=5的时候得到了以下结果对比

观察到随着L的增大，有的人脸检测更加准确，范围更加合理，也有的人脸检测准确度下降。准确度变化的不稳定主要是由算法造成。因为我首先由初步的筛选得到人脸的大致范围，这相当于确定了一个初值。而后我在初始范围的基础上进行贪心算法。类似最速下降法可能会受到初值影响，这种贪心算法的具体操作也会受到初值影响，即使是不同的微调步长都会对结果产生影响，而且L选取的不同也会改变对距离的计算结果，影响算法的判断。

至于右3人脸始终没有被识别出来，这是因为我设定了最小人脸大小的阈值。如果阈值太小，由于颜色相近，手等裸露的其他肢体部位也容易被识别为人脸。初筛得到的右3人脸的范围和手的大小相近，为了不引起误判，只能保持此阈值。至于为什么右3人脸的初筛范围较小，其一是因为他戴了口罩，遮挡住了一部分脸。另外，我发现这张照片的高光和阴影也对识别结果造成了较大影响。比如L=3时最终识别结果明显倾向于保留人脸高光的部分，而暗部经常被丢掉。这可能是由于暗部的颜色分布距离标准人脸比较远。注意到右3人脸大部分都在阴影内，所以初筛时被识别出来的块数就比较少，自然也就更容易被过滤掉。

Task 4.3.3

- 顺时针旋转90°

由于我初筛人脸是切割成方块，所以旋转90°对初筛时切割结果的影响很小。外加上旋转90°本身不改变每个方块内的颜色分布，所以对通过颜色识别人脸的方法的检测结果基本没有影响。最后，由于我的算法在微调边缘时将四边的调整结果同时考虑选取最优调整，所以上下左右边框是平等的，不会因为旋转而让算法偏爱调整某一个方向而忽略另一个。综上，理论上旋转90°对最终的检测结果也几乎没有影响。事实上最后的结果也和之前相近。

- 横向拉伸为1.5倍

横向拉伸后的效果整体提升。这主要是因为横向拉伸后变相地提升了横向的分辨率。原先分块时一张人脸可能只能横向被分为两块，拉伸后可能就能被分为四块。这样能更好地将所有含有人脸的块识别出来（取个极限，如果分辨率无穷大，假设仍能准确识别每一块是否包含人脸，那么就能完美地将整张人脸提取出来，而不会将周围的杂像素也包含进来，或是错误地滤除部分含有人脸的块）。这样就有了更好的初值，使后面的微调结果也得到改善。

- 适当改变颜色

之前就分析过识别结果受到高光部分和阴影部分的颜色分布差异影响。由于阴影部分的颜色分布距离标准人脸的颜色分布较远，贪心算法倾向于将更接近标准人脸的高光部分保留，而剔除暗部。改变颜色后阴影和高光的区别就更明显了，这也导致这一趋势被进一步放大。

Task 4.3.4

如果可以重新选择训练样本标准，我认为我会更加注重人种、光照条件、人脸角度和表情的影响。当前人脸训练样本存在的一个较大的问题是大多数图片都是欧美白人。这样如果被测人种和训练人种不同，就更容易造成误判。另一个问题是较多的训练样本图片都是在光照较为均匀的环境下拍摄的，只有少部分存在较明显的明暗变化。这一点造成的问题是如果我用依靠这些图片训练出的模型去检测光照条件较差的图片中的人脸（就比如我选择的图片），就容易造成部分人脸区域被漏掉。针对以上两点，我会根据待检测的人种选择相同人种的训练样本，并确保涵盖较为丰富的光照条件，从光照充足均匀到光照较弱且不均。再者，我认为现有样本的两个优点是涵盖了较为丰富的人脸角度和表情。这两点显然也会改变不同颜色的占比。比如张着嘴时嘴里的黑色就会拉高黑色的占比，侧过脸时脸颊和腮部的肉色会更加占主要地位。这两点优点在新的选取标准中应当加以保留。最后，针对我的算法，我认为如果要优化初筛的结果，可以专门对某块完整的皮肤进行样本采集，训练一个专门服务于初筛的模型，这样也许可以改善对小块人脸的检测准确度。

总结

这次大作业比音乐合成大作业要更有挑战性，也让我学到了更多新东西。音乐合成更多是与一维的东西打交道，傅里叶变换等处理一维波形的方法也在信号与系统中学过了。而图像处理涉及到二维DCT变换、多个颜色通道和二维图像上的人脸识别等，这些无疑带来了更大的陌生感。同时，在完成这个大作业时，我也遇到了很多熟悉的东西。比如新学习的JPEG压缩原理涉及到了在数据与算法中学习的Huffman编码，我的人脸检测算法用到了之前学习过的DFS算法、贪心算法设计思想等。可以说做这次作业的过程就是在融汇旧知识的同时现学现卖新知识。

最后感谢助教和老师对这门课程的付出。