

Matlab 高级编程与工程应用

实验 3 图像处理

姓名：黎佳维

学号：2022010540

文件说明：

code 文件夹中，名如 code2_3_1 的，即为解决第“2”章节，第“3”题的第“1”份代码；DCT_QUANT_ZIG 等为 function 封装函数代码。

result 文件夹中，名如 2_3_1 的，即为第“2”章节，第“3”题的第“1”份输出结果。

第一章 基础知识

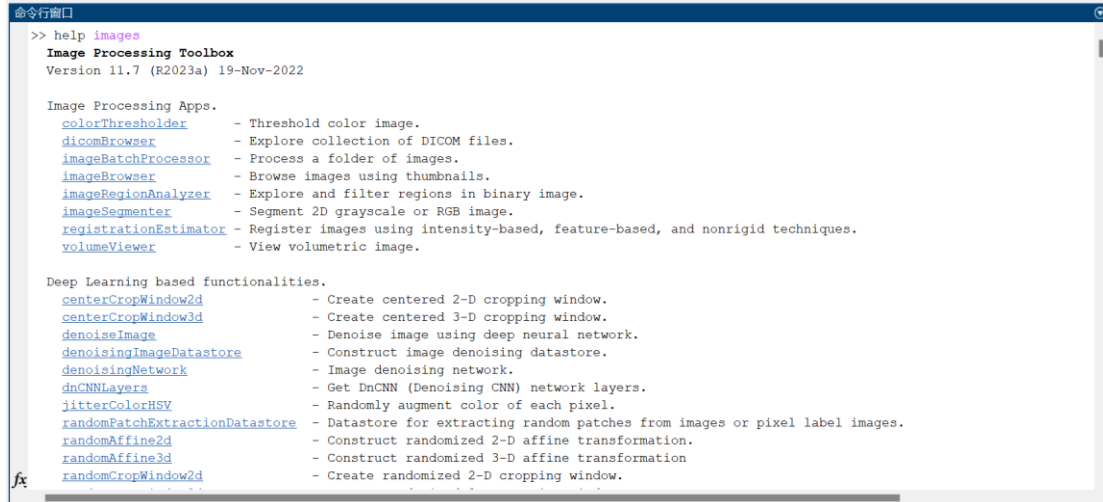
1.

如下图，在命令行输入指令后输出了 images 图像处理的相关函数，其中典型的有：

colorThresholder - 对彩色图像进行阈值化处理

denoiseImage - 使用深度神经网络对图像降噪

countlabels - 计算唯一标签的数量



```
>> help images
Image Processing Toolbox
Version 11.7 (R2023a) 19-Nov-2022

Image Processing Apps.
    colorThresholder - Threshold color image.
    dicomBrowser     - Explore collection of DICOM files.
    imageBatchProcessor - Process a folder of images.
    imageBrowser      - Browse images using thumbnails.
    imageRegionAnalyzer - Explore and filter regions in binary image.
    imageSegmenter     - Segment 2D grayscale or RGB image.
    registrationEstimator - Register images using intensity-based, feature-based, and nonrigid techniques.
    volumeViewer       - View volumetric image.

Deep Learning based functionalities.
    centerCropWindow2d - Create centered 2-D cropping window.
    centerCropWindow3d - Create centered 3-D cropping window.
    denoiseImage       - Denoise image using deep neural network.
    denoisingImageDatastore - Construct image denoising datastore.
    denoisingNetwork    - Image denoising network.
    dnCNNLayers         - Get DnCNN (Denoising CNN) network layers.
    jitterColorHSV      - Randomly augment color of each pixel.
    randomPatchExtractionDatastore - Datastore for extracting random patches from images or pixel label images.
    randomAffine2d      - Construct randomized 2-D affine transformation.
    randomAffine3d      - Construct randomized 3-D affine transformation.
    randomCropWindow2d - Create randomized 2-D cropping window.
```

2.

(a) 如下方关键代码所示，导入图像后复制到 image1 中进行处理，找到图像的中心点作为圆心，以及较短的一边为直径，作圆。作圆的方法是从极坐标出发，定义 theta 变量后确定相对的直角坐标系中的位置，用 plot 函数作圆。

输出结果如下左图所示。所画的圆看起来上面缺了一截的原因是因为该图片纵向长度为偶数，此处的 y 取值为直接取一半，因此存在半个像素点的误差，若为 y+0.5 则得到下图，此时画出的圆形在图中达到对称。此种画圆的方法实际上改变了图片的原始分辨率，另一种不改变分辨率的方法是扫描图片上每一个像素点，若其到圆心的距离值等于半径附件，则将其改变为红色，因题目未作此要求，故在此处采用了第一种方案，方案二的处理与(b)同理。

关键代码 (code1_2_1):

```
load("source/hall.mat");
image1 = hall_color;%create a copy
[col1, row1, channel1] = size(image1);%size of the image
...
theta = 0:pi/20:2*pi;%plot a red circle on the image
x = centre_x + r*cos(theta);%the x coordinate of the circle
y = centre_y + r*sin(theta);%the y coordinate of the circle
plot(x,y,'r','LineWidth',2)
```

输出结果：



(b) 国际象棋棋盘大小为 8×8 。如下方关键代码所示，图像中处于“编号（=竖列数+横列数）”为奇数的部分被变成了黑色，采用了直接修改像素点参数的方法，输出结果如下方所示。

关键代码 (code1_2_2):

```
for i = 1 : 8%find the row and column to rewrite: 8x8
    for j = 1 : 8
        if(mod((i + j),2))%color black in an area
            image1(1+col1/8*(j-1):col1/8*j,1+row1/8*(i-1):row1/8*i,1) = 0;
            image1(1+col1/8*(j-1):col1/8*j,1+row1/8*(i-1):row1/8*i,2) = 0;
            image1(1+col1/8*(j-1):col1/8*j,1+row1/8*(i-1):row1/8*i,3) = 0;
        end
    end
end
end
```

输出结果：



第二章 图像压缩编码

1.

从理论上分析，由于 dct 具有线性性质，因此在“时域”上每点减去 128，和在频域上减去一个同样尺寸的每点都为 128 的图像的 dct 变换所达到的效果是一致的。

如下关键代码所示，`image2` 存储的是在“时域”减去 128 后进行 dct 变换的图像，`image1` 反之（原始数据钩取自测试图像同一部分，在本章中其他题也均采取该部分），因为要得到 `image1` 的值，必须经过逆 dct 过程，故对 `image2` 也进行同样的过程进行对照试验。所得的两个图像之间均方误差约为 $\text{mse}=2.95\text{e-}27$ ，差距非常小；输出结果部分的图像从肉眼上看也几乎无法分辨出其区别，说明在变换域进行预处理具有同样的效果。

关键代码 (`code2_1_1`):

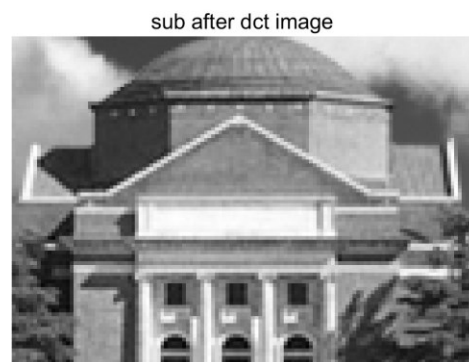
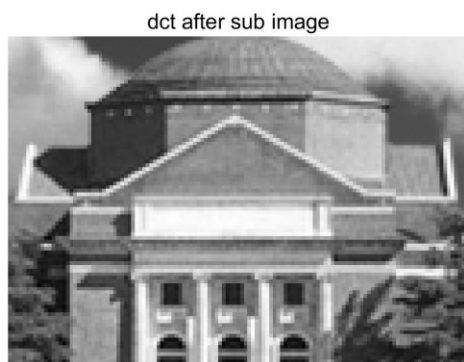
```
%sub 128 then dct
image2 = double(image1) - 128 * ones(size(image1));
image2_dct = dct2(image2);
image2 = idct2(image2_dct);

%dct then "sub 128"
image1_dct = dct2(image1);
image1_dct = image1_dct - dct2(128 * ones(size(image1)));
image1 = idct2(image1_dct);

%compute the similarity
mse = sum((image1-image2) .^2, 'all') / (col2*row2);

%show the images
image1 = uint8(double(image1) + 128 * ones(size(image1)));
image2 = uint8(double(image2) + 128 * ones(size(image2)));
```

输出结果:



2.

如下所示的关键代码展示了使用函数所写的二维 DCT 变换，函数先生成了二位 DCT 变换的左乘与右乘矩阵，再对输入数据进行矩阵乘得到变换后的结果由 `res` 返回。与调用系统的 `dct2` 函数相比，其相差用均方误差估算约为 `mse=9.70e-25`，差距非常小，故可以认为自行编写的 `dct2_self` 函数达到了二维 DCT 变换的效果。

关键代码 (`dct2_self`):

```
%the dect2 written by myself
function res = dct2_self(in)
    %the size of the two mats
    [N1, N2] = size(in);
    row1 = (0: N1 - 1)';
    col1 = 1: 2: 2 * N1 - 1;
    %the left matrix
    D1 = cos(row1 * col1 * pi / (2*N1));
    D1(1,:) = sqrt(1/2) * ones(1,N1);
    row2 = (0: N2 - 1)';
    col2 = 1: 2: 2 * N2 - 1;
    %the right matrix
    D2 = cos(row2 * col2 * pi / (2*N2));
    D2(1,:) = sqrt(1/2) * ones(1,N2);
    %left multi and right multi
    res = sqrt(4/N1/N2) * D1 * in * D2';
end
```

3.

从理论上分析，`dct` 系数矩阵左上方代表图像直流分量的强度，左下方系数代表图像块中纵向变化纹理的强度，右上方系数代表图像块中横向变化纹理的强度，右下方系数代表图像块中横纵向两个方向变化纹理的强度。`dct` 系数矩阵右侧全部置 0 后其横纵向两个方向变化尤其横向变化变得不那么明显，左侧全部置 0 后其纵向变化的分量和直流分量变小。

第 1、2 题是对 `dct` 性质的验证，故不对图像进行分块也可以进行。从第 3 题开始，将对图像进行 8×8 的分块，由于该章节具有较多的流程化、重复功能，故设计了一些函数实现操作。在第 3 题，设计了 `split`、`reconstruct` 两个函数分别将矩阵分块为 8×8 大小的 `blocks`、由 8×8 大小的分 `blocks` 块进行复原；`sub8`、`add8` 两个函数分别将图像值-128 并转化为 `double` 类型、+128 并复原为 `uint8` 型。

通过调用以上函数，对每个分块的 `block` 进行 `dct` 变换后对系数进行处理得到新的 `block2`（右四列置 0）与 `block3`（左四列置 0），复原后输出图像如下图所示。可见，右四列置 0 的图像横向变化变得不那么明显，左四列置 0 的图像直流分量被清除，且纵向变化变得不那么明显，纵向分量得以保留。

输出结果：



关键代码 (code2_3_1):

`%change the dct parts`

```
for temp1 = 1:num_row
    for temp2 = 1:num_col
        currentblock = block1{temp1, temp2};
        block_dct{temp1, temp2} = dct2(currentblock);

        %rightside 4s -> zeros
        block_dct2 = block_dct{temp1, temp2};
        block_dct2(:,end-3:end) = 0;
        block2{temp1, temp2} = idct2(block_dct2);

        %leftside 4s -> zeros
        block_dct3 = block_dct{temp1, temp2};
        block_dct3(:,1:4) = 0;
        block3{temp1, temp2} = idct2(block_dct3);
    end
end
```

(下为 split, reconstruct 基本逻辑与其一致, 区别在于将分割改为拼接)

```
function blocks = split(A)
    %partition numbers
    [rows, cols] = size(A);
    row_num = floor(rows/8);
    col_num = floor(cols/8);

    %generate the blocks
    blocks = cell(row_num, col_num);
    for temp1 = 1:row_num
```

```

for temp2 = 1:col_num
    rowStart = (temp1-1) * 8 + 1;
    rowEnd = temp1 * 8;
    colStart = (temp2-1) * 8 + 1;
    colEnd = temp2 * 8;
    blocks{temp1, temp2} = A(rowStart:rowEnd, colStart:colEnd);
end
end
end

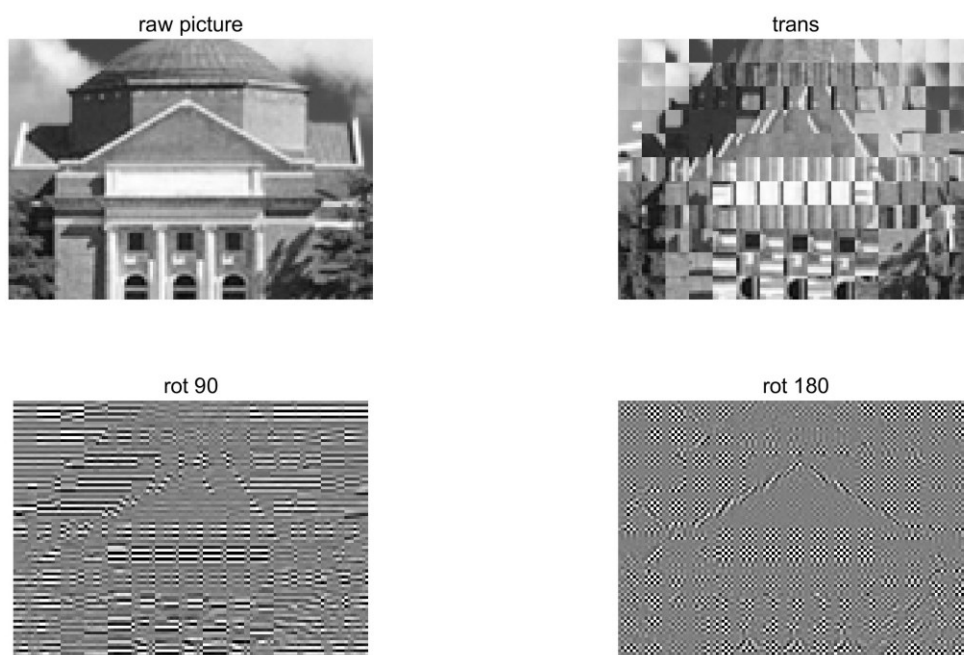
```

4.

从理论上分析，转置就是将横向分量与竖向分量相交换而大小不发生变化，因此输出图像应该为横竖转置；rot90 为逆时针旋转 90 度，此时直流分量来到左下方，因此图像横向纹理增强，纵向变化明显；旋转 180 度，则直流分量来到右下方横纵向两个方向变化纹理的强度都变强。

本题只需要在 3 的基础上对矩阵的变换方式进行修改，如下方关键代码所示。得到的输出图像如下所示，其中 trans 为转置所得，可以看到其分块后的每个部分横竖转置；rot90 为逆时针旋转 90 度所得，可以看到图像横向纹理增强，纵向变化明显；rot 180 为旋转 180 度所得，可以看到图像横竖纹理均增强，呈格子状。

输出结果：



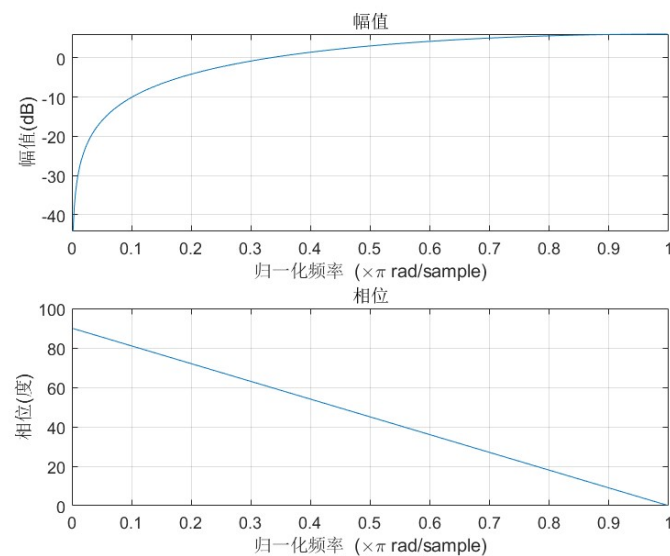
关键代码 (code2_3_1):

```
for temp1 = 1:num_row
    for temp2 = 1:num_col
        currentblock = block1{temp1, temp2};
        block_dct{temp1, temp2} = dct2(currentblock);
        %trans and idct back
        block_dct2 = block_dct{temp1, temp2}.';
        block2{temp1, temp2} = idct2(block_dct2);
        %rot 90 and idct back
        block_dct3 = rot90(block_dct{temp1, temp2});
        block3{temp1, temp2} = idct2(block_dct3);
        %rot 180 and idct back
        block_dct4 = rot90(block_dct{temp1, temp2}, 2);
        block4{temp1, temp2} = idct2(block_dct4);
    end
end
```

5.

如果认为差分编码是一个系统，如下输出结果所示利用 `freqz` 函数绘出了其频率响应，说明其为高通滤波器：DC 系数先进行差分编码再进行熵编码，说明 DC 系数高频分量更多。

输出结果：



关键代码：

```
z = [1, -1];  
p = [1];  
figure;  
freqz(z, p);
```

6.

DC 预测误差 δ 与 Category 的关系是：

$$Category = \text{ceil}(\log_2(|\delta| + 1))$$

据此公式，根据误差即可计算出其 Category。

7.

实现 `zigzag` 可以对矩阵按照对角线的顺序进行逐一扫描，也可以直接通过查找表的方式进行变换。在本题中因为索要扫描的矩阵大小均一致，因此直接采用查找表的方式进行变换，具有更高的效率，其功能封装在函数 `zigzag` 中。

关键代码（`zigzag`）：

```
%zigzag scan into a vector  
function B = zigzag(A)  
    temp = reshape(A, [64, 1]);  
    B = (temp([1, 9, 2, 3, 10, 17, 25, 18, ...]));  
end
```

8.

这一步主要需要完成量化的功能，如下关键代码所示，将 `dct2`、量化与 `zigzag` 排序后写入矩阵这几个功能封装到了 `DCT_QUANT_ZIG` 函数中。函数首先将分块好的 `block` 按顺序进行 `dct2`、量化后得到新的 `block`，再对新的 `block` 进行 `zigzag` 排序成列向量后按序写入输出矩阵中。主程序 `code2_8_1` 调用了该函数以及前序处理函数，输出结果矩阵存储在了 `2_8_1.mat` 中。

关键代码（`DCT_QUANT_ZIG`）：

```
%DCT, Quantify and Zigzag to matrix  
function B = DCT_QUANT_ZIG(A)  
    load("source/JpegCoeff.mat");  
    [num_row, num_col] = size(A);  
    block_dct = cell(num_row, num_col);  
  
    %dct2 then quantify  
    for temp1 = 1 : num_row
```

```

        for temp2 = 1 : num_col
            block_dct{temp1, temp2} = round(dct2(A{temp1, temp2})./QTAB);
        end
    end

    %zigzag then matrix
    [num_row1, num_col1] = size(block_dct);
    num_mat1 = num_row1 * num_col1;
    B = zeros(8^2, num_mat1);%output matrix
    temp3 = 1;%the wrting column
    for temp1 = 1 : num_row1
        for temp2 = 1 : num_col1
            currentmatrix = block_dct{temp1, temp2};
            currentvector = zigzag(currentmatrix);%zigzag into a vector
            B(:, temp3) = currentvector;
            temp3 = temp3 + 1;
        end
    end
end
end

```

9.

这一步主要需要完成熵编码，分为对 DC 系数与对 AC 系数进行编码两部分，该功能封装在 coding 函数中，前置步骤与 8 相同。

如关键代码所示，其前半部分%1 dc 为 DC 系数编码，由于前一步已将其提取到量化矩阵的第一行中，其流程为：差分 -> 寻找霍夫曼码（计算 DCTAB 查找表的索引） -> 寻找数字的 1 补码（dec2bin -> char -> double -> 小于 0 补码） -> 插入这两部分，最终得到 DC 编码后的 B1。代码中%1 the huff part of code 与%2 the maglitude part of the code 对应了流程中的两种插入情况。

其后半部分%2 ac 为 AC 系数编码，也是在量化矩阵的基础上进行编码，其流程为：按序比对当前系数直到非 0，同时计数 0 的个数 -> 以 16 为一组插入 ZRL -> 插入剩余部分 0 与数字的 1 补码 -> 遍历所有数后插入 EOB。代码中%1 insert ZRL 与%2 insert the rest 0 and amplitude 与%3 insert EOB 对应了流程中的三种插入情况。

最后，整体文件 code2_9_1 在调用了第八题所示的所有函数基础上，还调用了 coding 函数对量化矩阵处理，通过 save('result/jpegcodes.mat', 'DC', 'AC', 'row', 'col');将数据输出到了 result 文件夹的 jpegcodes.mat 中。

关键代码 (coding):

%code into dc and ac paras, dc = B1, ac = B2, A is the input matrix after
%quantify and zigzag

```
function [B1 B2] = coding(A)
load("source/JpegCoeff.mat");
%1 dc
A1 = A(1, :);
A2 = [A1(1); (A1(1: end - 1) - A1(2: end))'];%diff
A3 = ceil(log2(abs(A2) + 1));%category
B1 = [];
for temp1 = 1 : length(A2)
    %1 the huff part of code
    row_huff = A3(temp1) + 1;
    col_huff = DCTAB(row_huff, 1) + 1;
    code_huff = DCTAB(row_huff, 2 : col_huff)';

    %2 the maglitude part of the code
    code_magn = dec2bin(abs(A2(temp1)))' - '0';
    if (A2(temp1) < 0)
        code_magn = ~code_magn;
    end%transfer to 1-bin
    B1 = [B1; code_huff; code_magn];
end

%2 ac
A4 = A(2: end, :);
ZRL = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1]';
EOB = [1, 0, 1, 0]';
[row, col] = size(A4);
B2 = [];
for temp1 = 1 : col
    %counting zero
    zero_count = 0;
    for temp2 = 1 : row
        if (A4(temp2, temp1))
            %1 insert ZRL
            while (zero_count >= 16)
                B2 = [B2; ZRL];
                zero_count = zero_count - 16;
            end
        end
    end
end
```

```

end

%2 insert the rest 0 and amplitude
row1 = zero_count * 10 + ceil(log2(abs(A4(temp2, temp1)) +
1));%category
col1 = ACTAB(row1, 3) + 3;
code_zero = ACTAB(row1 , 4 : col1)';
cond_ampt = dec2bin(abs(A4(temp2,temp1)))' - '0';
if (A4(temp2, temp1) < 0)
    cond_ampt = ~cond_ampt;
end%transfer to 1-bin
B2 = [B2; code_zero; cond_ampt];
zero_count = 0;
else
    zero_count = zero_count + 1;
end
end
end
%3 insert EOB
B2 = [B2; EOB];
end
end

```

10.

此处采用 1bit 作为基本单位，因码流是二进制表示；对于图像，一个像素点的像素取值范围是 0~255，需要 8bit 存储，故原始参数需要*8，如关键代码所示，按照此方式计算了压缩率，得到其值为 6.418849，可见图像得到了明显的压缩。

关键代码：

```

%computing the compressing rate
compressing_rate = 8 * col * row / (length(DC) + length(AC));

```

11.

在该部分，要对熵编码、zigzag、量化、dct2、分块、减 128 流程进行反向操作，其中后面几项已经在前面的题目中实现，在该部分还需要完成熵编码、zigzag、量化这三步的逆向操作，其分别封装在 decoding 和 IDCT_QUANT_ZIG 函数中。如下方关键代码所示，调用了这些函数对之前生产的 jpegcodes.mat 数据进行处理后得到了解码后的图片 image2。

在 decoding 函数中，对于 DC 系数的解码是通过寻找差分值 -> （比对霍夫曼编码 -> 通过霍夫曼编码恢复数值 -> 1 补码转 10 进制） -> 恢复原数列这样的流程实现的，恢复 AC 系数的流程类似，也是比对霍夫曼编码后恢复幅值，稍有不同的地方是 AC 系数中存在

两个特殊的编码需要单独处理。整体的 decoding 中，核心思想就是比对霍夫曼编码，因其为前缀码，不存在前缀重合的情况，使得比对的算法设计较为清晰。

在 IDCT_QUANT_ZIG 函数中，首先对序列进行了“反 ZIGZAG”处理，再重新乘了 QTAB 系数反量化后在进行 idct2 操作，设计思想与正序的过程类似。以上两个函数均较长，设计思想与正向过程都类似，代码中含有注释，因此在此仅作大致阐释而没有直接 copy 在文档里。

在客观数据上，计算所得到的 PSNR 值为 31.187403，说明图像失真较小。而为了直观体现其失真情况，还输出了编码前后的图片，如下方输出结果所示，其失真较小，只有一些细微的差别。

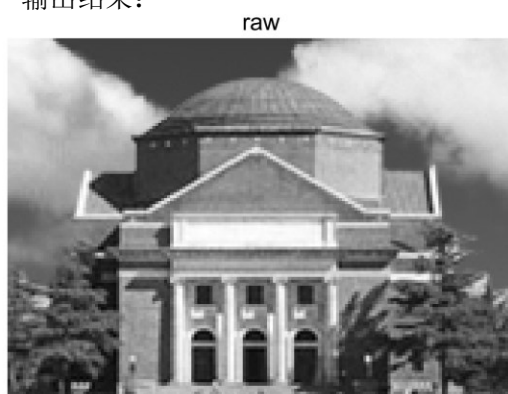
关键代码 (code2_11_1):

```
%decoding the JPEG
matrix_decoded = decoding(DC, AC, row, col);
%iZigzag, deQuantify and iDCT into blocks
block2 = IDCT_QUANT_ZIG(matrix_decoded, row, col);
%reconstruct
image2 = reconstruct(block2);
%add 128 and transfer to uint8
image2 = add8(image2);

%calculate the PSNR
MSE = sum((double(hall_gray) - double(image2)) .^ 2, 'all') / (row * col);
PSNR = 10 * log10(255^2 / MSE);

%show the two images
figure;
subplot(1, 2, 2);
imshow(image2);
title("decoded", "FontSize", 20);
subplot(1, 2, 1);
imshow(hall_gray);
title("raw", "FontSize", 20);
```

输出结果:



12.

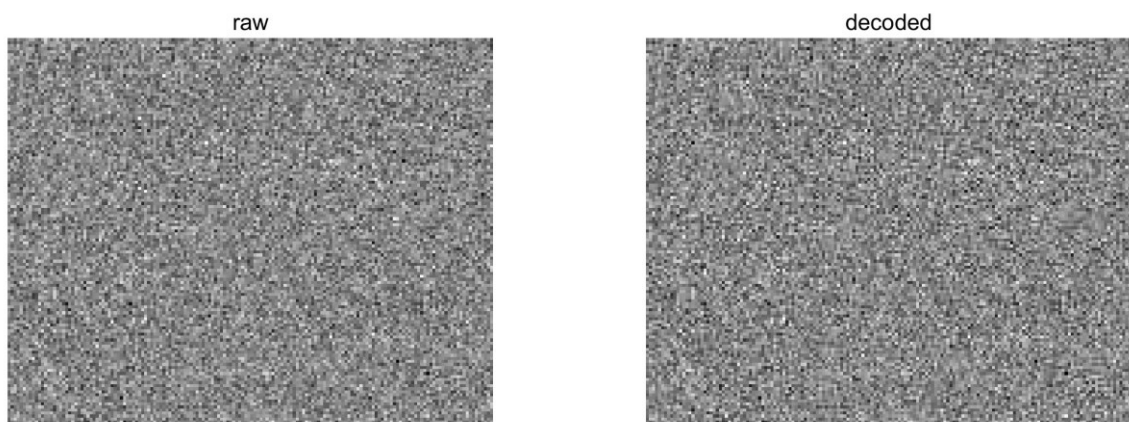
在原来设计的基础上，直接对 QTAB 参数除以 2，即将量化步长减小为原来的一半。在程序实现中，为调用了 QTAB 的函数 DCT_QUANT_ZIG 与 IDCT_QUANT_ZIG 增加了参数 para 自定义量化步长，设置 para=2 时得到 QTAB 参数/2，此时的 PSNR=34.206704，可见量化步长减小，失真同时减小。

13.

将原来的程序输入改为 snow，其余项均不需要改变，得到的输出结果如下：

图像压缩比=3.640727，PSNR=22.924444。其压缩前后的图像如下所示。直观上来看并不一定能对比出二者的差异，因为人眼对于高频信息的识别能力有限；而从数据上可以看到对于雪花图像的压缩比和 PSNR 都较小，说明图像的失真程度较大，这是因为雪花图像十分不规则，高频分量较多，而在此处设计的压缩算法中，正是对高频分量的削减程度更大，因此造成了较大的失真。而对比前面对于规则图像的压缩，可以看到因为人眼对于高频分量和低频分量的敏感性不同，因此算法在采取压缩高频分量的方式对于人眼阅读来说是影响不大的。

输出结果：



第三章 信息隐藏

1.

如下关键代码所示，原始图像为 image1，首先以最低有效位的替换的方式实现了空域隐藏，替换位存储在随机生成的 LSB_rand 矩阵中；替换了最低位的 image1 经过 JPEG 编码和解码后得到 imag2，使用与逻辑即可得到 image2 每个值的最低位恢复信息，统计此时的信息与原来写入的 LSB_rand 的匹配度，输出到 equal_rate 中。

以下输出结果是经过多次随机试验得到的匹配率，可见准确率都在 0.5 上下浮动，由于对比的位数只有 1 个 bit，因此该准确率实际上体现了“不准确”：即空域隐藏方法使得替换的最低位数据丢失，最终匹配体现的只是 1bit 中随机出来的准确率。这说明其不具有抗 JPEG 编码能力。

输出结果：

0.5015	0.5000	0.4990	0.5014	0.4999
0.5029	0.4992	0.5016	0.5014	0.5060

关键代码 (code3_1_1):

```
%hide the LSB
LSB_rand = randi([0, 1], row, col);
image1 = bitset(image1, 1, LSB_rand);
...
%compare the LSBs of the raw image and the decoded image
equal_num = 0;%the number of equal values
for temp1 = 1 : row
    for temp2 = 1 : col
        if(bitand(image2(temp1,temp2),1) == LSB_rand(temp1,temp2))
            equal_num = equal_num + 1;
        end
    end
end
total_num = row * col;
equal_rate = equal_num / total_num;%the rate of equal values
disp(equal_rate);
```

2.

在该部分，依次实现了三种变换域信息隐藏与提取的方法，首先解释实现功能的代码，如下关键代码所示：三种变换域信息隐藏的方法依次对应函数 DCT_QUANT_ZIG1、DCT_QUANT_ZIG2、DCT_QUANT_ZIG3，信息提取的方法对应 IDCT_QUANT_ZIG1、IDCT_QUANT_ZIG3，1、2 方法的信息提取函数本质一样，在实现上写为了一个函数。

以下为关键代码与其阐释，分别在各个代码标题的下方有简要的阐释。

关键代码与阐释：

(DCT_QUANT_ZIG1)

方法一隐藏在进行 dct2 变换后，直接将 block_dct 中的 cell 依次取出，使用 bitset 将 LSB 替换为 rand_data 对应的 cell 的值。

```
%dct2 then quantify
for temp1 = 1 : num_row
    for temp2 = 1 : num_col
        block_dct{temp1, temp2} = round(dct2(A{temp1, temp2})./(QTAB ./
para));

        %replace the LSB, celltemp is the mat to wrtie and randtemp is
        %the mat used to write
        cell_temp = int32(block_dct{temp1, temp2});
        rand_temp = int32(rand_data{temp1, temp2});
        cell_temp = bitset(cell_temp, 1, rand_temp);
        block_dct{temp1, temp2} = double(cell_temp);
    end
end
```

(DCT_QUANT_ZIG2)

为方便观察，方法二在一的基础上选择将最右边的列的 dct2 系数进行 LSB 替换隐藏，既达到了部分隐藏的目的，也便于后续的观察。在代码实现上，直接将方法一的修改列数固定为最右列即可。

```
%only change the rightmost column
for temp1 = 1 : num_row
    temp2 = num_col;
    cell_temp = int32(block_dct{temp1, temp2});
    rand_temp = int32(rand_data{temp1, temp2});
    cell_temp = bitset(cell_temp, 1, rand_temp);
    block_dct{temp1, temp2} = double(cell_temp);
end
```

(DCT_QUANT_ZIG3)

方法三在 zigzag 实现后对暂存变量 currentvector 进行替换后再写入输出变量中。寻找写入地址的方法是从后往前扫描至第一个非零值即找到相对的写入位置，之后将信息写入对应位置。值得注意的是该函数调用的随机生成的信息需要是 1、-1 编码的，否则提取信息时无法提取后续作为信息写入的“0”。

```
%zigzag then matrix
```

```

[num_row1, num_col1] = size(block_dct);
num_mat1 = num_row1 * num_col1;
B = zeros(8^2, num_mat1);%output matrix
temp3 = 1;%the wrting column
index_writing = 1;
for temp1 = 1 : num_row1
    for temp2 = 1 : num_col1
        currentmatrix = block_dct{temp1, temp2};
        currentvector = zigzag(currentmatrix);%zigzag into a vector
        for temp4 = 64 : -1 : 1
            if(currentvector(temp4))
                if(temp4 == 64)
                    currentvector(temp4) = rand_data(index_writing);
                else
                    currentvector(temp4 + 1) = rand_data(index_writing);
                end
                index_writing = index_writing + 1;
                break;
            end
        end
        B(:, temp3) = currentvector;
        temp3 = temp3 + 1;
    end
end
end

```

(IDCT_QUANT_ZIG1)

信息提取方法一与隐藏方法一实现上相反，找到对应区间的序列后将其提取出来，直接通过与操作即可得到最低位，也就是写入信息的位置，再写入到输出变量中。由于方法二未写入信息的位置不发生改变，因此也可以采用此方法直接提取来统计正确率。

```

for temp1 = 1 : block_y
    for temp2 = 1 : block_x
        %the idx of the vector using now
        start_idx = ((temp1-1) *(block_x) + temp2 - 1) * 64 + 1;
        end_idx = start_idx + 63;
        current_block = A(start_idx:end_idx);
        blocks{temp1,temp2} = izigzag(current_block);%fix the blocks
        blocks{temp1,temp2} = bitand(int8(blocks{temp1,temp2}),1);%write
back the LSB
    end
end

```

end

(IDCT_QUANT_ZIG3)

信息提取方法三的设计与隐藏方法实现上相反，从后往前寻找非 0 值的位置，即得到了写入信息。因写入信息为 1、-1，因此寻址不存在矛盾的情况。

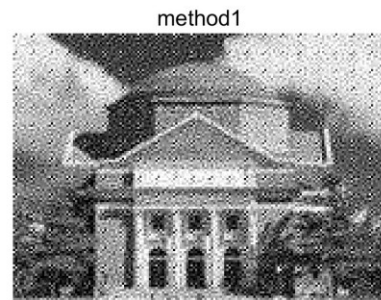
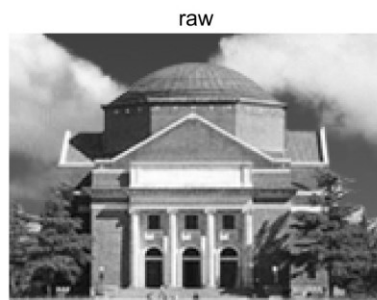
```
index_writing = 1;
for temp1 = 1 : block_y
    for temp2 = 1 : block_x
        %the idx of the vector using now
        start_idx = ((temp1-1) *(block_x) + temp2 - 1) * 64 + 1;
        end_idx = start_idx + 63;
        current_block = A(start_idx:end_idx);
        for temp3 = 64 : -1 : 1%find the last no-zero value
            if(current_block(temp3))
                B(index_writing) = current_block(temp3);%write back
                index_writing = index_writing + 1;
                break;
            end
        end
    end
end
end
```

综上，因为信息隐藏实现于 dct2 与量化后，通过修改之前部分的 DCT_QUANT_ZIG 函数即可实现不同的信息隐藏功能。

通过调用上述函数，得到了输出结果后对其进行验证，如下关键代码所示，得到了这样一些输出结果用于主观认识与客观分析。从数据上来看，三种方法的 equal_rate 都为 1，说明其具备抗 JPEG 编码的能力。在压缩率上，方案一明显更小，方案二、三接近，几乎为方案一的两倍；而在失真情况上，方案三优于方案二，前两者又远好于方案一。直观来看，方案一相比原图失真明显，而方案二最右侧失真明显，其原因就是因隐藏过程中仅对右侧参数进行了替换；而方案三失真最为小，原因也是其隐藏方法将信息均匀地分布到了各个 block 的高频参量中，因此方案三实际有最佳的效果，理论分析与实际相一致。

输出结果：

方法	equal_rate	compressing_rate	PSNR
一	1	2.852897	15.439729
二	1	5.986415	26.836512
三	1	6.186184	28.899551



关键代码 (code3_2_1):

%the output stage:

%compare the LSBs of the raw image and the decoded image

equal_num1 = 0;

for temp1 = 1 : row

 for temp2 = 1 : col

 if(recover1(temp1,temp2) == rawdata(temp1,temp2))

 equal_num1 = equal_num1 + 1;

 end

 end

end

total_num1 = row * col;

equal_rate1 = equal_num1 / total_num1;

equal_num2 = 0;

for temp1 = 1 : row

 temp2 = col;

 if(recover1(temp1,temp2) == rawdata(temp1,temp2))

 equal_num2 = equal_num2 + 1;

```

        end
    end
    total_num2 = row;
    equal_rate2 = equal_num2 / total_num2;

    equal_num3 = 0;
    for temp1 = 1 : length(recover3)
        if(recover3(temp1) == rawdata_2(temp1))
            equal_num3 = equal_num3 + 1;
        end
    end
    total_num3 = length(recover3);
    equal_rate3 = equal_num3 / total_num3;

    %compute the PSNR and compressing rate
    image1 = add8(image1);
    MSE1 = sum((double(unhide1) - double(image1)) .^ 2, 'all') / (row * col);
    PSNR1 = 10 * log10(255^2 / MSE1);
    compressing_rate1 = row * col * 8 / (length(DC1) + length(AC1));

    MSE2 = sum((double(unhide2) - double(image1)) .^ 2, 'all') / (row * col);
    PSNR2 = 10 * log10(255^2 / MSE2);
    compressing_rate2 = row * col * 8 / (length(DC2) + length(AC2));

    MSE3 = sum((double(unhide3) - double(image1)) .^ 2, 'all') / (row * col);
    PSNR3 = 10 * log10(255^2 / MSE3);
    compressing_rate3 = row * col * 8 / (length(DC3) + length(AC3));

```

第四章 人脸检测

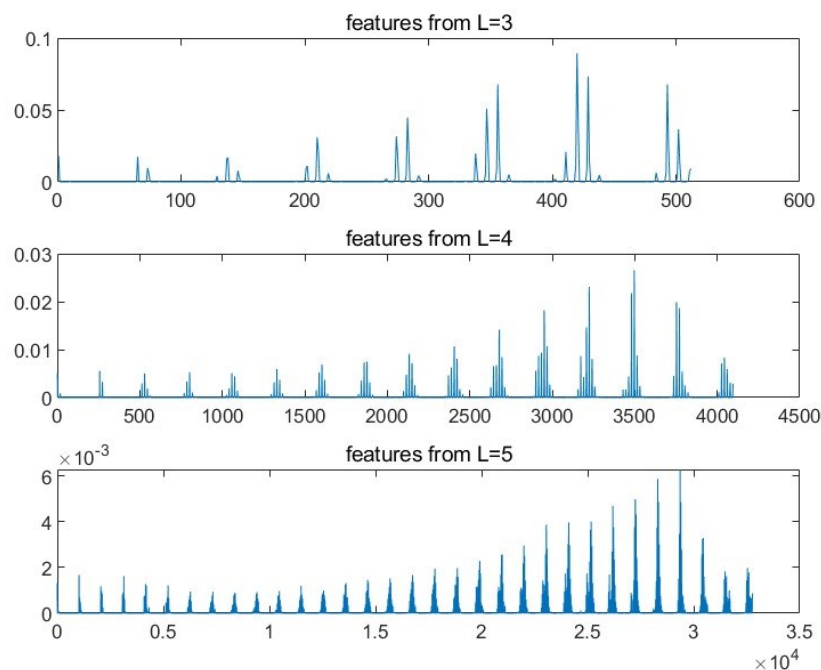
1.

样本大小对于检测并无影响，因为检测的对象是颜色的分布，其与大小无关。

L 取 3、4、5 三个值，所提取出来的特征分布规律一致，差异只体现在细节上，因为 L 越大，所耗比特数越多，因此特征保留也越多，但比特数的多少不改变总体的分布规律，训练集都一样。

本题所用代码如下所示，通过总程序 code4_1_1 调用了三次 training 函数，依次输入 L=3、4、5；training 函数依次对 33 张人像图片进行特征提取并记录在 feature 中，其中特征提取是依靠 extract 函数；extract 函数调用了自己编写的 rgbtrans 将对 rgb 值进行处理后得到特征并对图像依次进行处理。具体的参数设定可见于具体代码当中，根据输出结果参数设定不断迭代优化。

输出结果：



关键代码 (training):

```
%training all the images with para L
function feature = training(L)
    feature = zeros(1, 2 ^ (3 * L));
    %extract faces one by one
    for temp1 = 1: 33
        temp_road = "source/Faces/" + string(temp1) + ".bmp";%road to the
        image
```

```

        temp_face = imread(temp_image);
        temp_feat = extract(temp_face, L);
        feature = feature + temp_feat;%record the feat
    end
    feature = feature / 33;
end

```

2.

该部分算法主要由两个函数实现，函数 `detect` 对输入的图像按照输入参数 `blocksize` 的大小进行扫描，若该区域内的特征分布于输入的训练人脸特征的距离小于输入参数 `threshold` 则标记该区域为 1，并将向该区域添加红色蒙版便于观察。

找到了符合人脸颜色特征的小的 `blocks` 后，就要使用 `mark` 函数对这些 `block` 进行合并与标记。因为题目所要求的是框出一个矩形，同时人脸也具有一定的不规则性，因此 `mark` 部分选择了对前一步操作的 `blocks` 进行更大范围的矩形“框定”，具体实现为：（以下扫描步长均为 `blocksize`，因为上一步标定参数 0、1 时也是按照 `blocksize` 进行，增大步长有利于节约资源）对上一步输出的参数矩阵进行扫描，若有 1，则进入 `find_neighbour` 函数对该值的邻居进行扫描，“邻居”的意思是与原来的 1 响铃且值也为 1，位置可以在上下左右四个方向，由此即可得到矩阵中所有的 1 的分布范围，扫描各区域的同时将其置 0 以防止重复；邻居点的记录采用了 `queue` 的数据结构，先进先出。对于扫描获得的区域，若大于输入参数 `m`、`n`，则视作为人脸区域，在其四周画一个红色的矩形，同时还提供了 `x` 参数来改变矩形的比例。

根据以上思想与设计，编写了如下的关键代码：

关键代码：

（`detect`）

```

%detect the parts of faces: blocksize: length of basic unit; threshold:
%limit value; output: image matrix with 0 and 1(1 for faces)
function image_detect = detect(image, blocksize, trained_feature,
threshold, L)
    % Input size and output image
    [row, col, ~] = size(image);
    image_detect1 = image; % Start with the original image
    image_detect = zeros(row,col);
    % Define the red mask (light red)
    red_mask = cat(3, ones(blocksize, blocksize) * 255, zeros(blocksize,
blocksize), zeros(blocksize, blocksize));
    % Scan through each block of the image
    for temp1 = 1 : blocksize : row - blocksize + 1
        for temp2 = 1 : blocksize : col - blocksize + 1
            % The block detecting now

```

```

        block = image(temp1 : temp1 + blocksize - 1, temp2 : temp2 +
blocksize - 1, :);
        block_feat = extract(block, L); % Convert block to grayscale for
feature extraction
        delta = 1 - sum(sqrt(block_feat .* trained_feature));

        % Mark the face area
        if delta < threshold
            % Add a light red overlay to the detected area
            image_detect(temp1 : temp1 + blocksize - 1, temp2 : temp2 +
blocksize - 1, :) = 1;
            image_detect1(temp1 : temp1 + blocksize - 1, temp2 : temp2 +
blocksize - 1, :) = ...
                0.7 * double(image(temp1 : temp1 + blocksize - 1, temp2 :
temp2 + blocksize - 1, :)) + 0.3 * red_mask;
        end
    end
end
figure;
imshow(image_detect1);

```

end

(mark)

```

function image_mark = mark(image, image_detect, m, n, x, blocksize)
    [row, col] = size(image_detect);
    image_mark = image;
    for temp1 = 1 : blocksize : row
        for temp2 = 1 : blocksize : col
            if (image_detect(temp1,temp2) == 1)
                %find a nonzero point, then find its neighbours to check if
                %this area is big enough representing human head
                [left, right, up, down] = find_neighbour(image_detect,
temp1, temp2,blocksize);
                if(right - left >= m && down- up >= n)
                    %if larger than pre input size of head, then draw a red
                    %rectangle
                    image_mark(up, left : right , 1) = 255;
                    image_mark(up, left : right , 2) = 0;
                    image_mark(up, left : right , 3) = 0;
                    ...
                end
            end
        end
    end
end

```



```

end
end

%find the neighbours of 1, the neighbour of is means its value is 1 and it
%is connected to the "host" 1
function [left, right, up ,down] = find_neighbour(image_detect, row, col,
blocksize)
    left = col;
    right = col;
    up = row;
    down = row;

    queue = [row, col];
    image_detect(row : row + blocksize - 1, col : col + blocksize - 1) = 0;

    %using queue to store the points to check
    while ~isempty(queue)
        current = queue(1, :);
        queue(1, :) = [];

        r = current(1);
        c = current(2);
        %record the area of the neighbour area in rectangle
        left = min(left, c);
        right = max(right, c);
        up = min(up, r);
        down = max(down, r);
        %check the neighbours around, four directions
        neighbors = [r+blocksize, c;
                    r, c+blocksize;
                    r-blocksize, c;
                    r, c-blocksize];
        for k = 1:size(neighbors, 1)
            nr = neighbors(k, 1);
            nc = neighbors(k, 2);

            %if == 1, into queue and let it be 0
            if (image_detect(nr, nc) == 1)
                queue = [queue; nr, nc];
                image_detect(nr : nr + blocksize - 1, nc : nc + blocksize -
1) = 0;
            end
        end
    end
end
end

```

```
right = right + blocksize;  
down = down + blocksize;  
end
```

总代码 `code4_2_1` 调用了上述函数，并将 L 设置为 3、4、5 运行了三次，得到如下所示的输出结果。图片从上至下依次为 $L=3$ 、4、5 的情况，左为 `detect` 的输出，右为 `mark` 的输出。可以看到 `detect` 阶段得到了较为准确的面部识别的同时也识别了手部等颜色类似的区域，因此需要 `mark` 部分对于区域大小的界定来过滤掉这些区域。经过 `mark` 后的右图可见人脸区域得到了准确的输出结果。在文件中可能存在压缩图片不清晰的情况，在 `result` 文件夹中右更清晰的版本。

对于不同的 L ，阈值的设定依次为 0.4、0.7、0.8（对应 3、4、5），这三个数得到了较好的结果，阈值随着 L 增大的原因是 L 越大，特征信息保留越丰富，因此对测试图像和训练数据的匹配更加准确。

输出结果：





上为 $L=3$



上为 $L=4$



上为 $L=5$



3.

采用如下关键代码的变换，对图像进行处理后进行分析。

关键代码 (code4_3_1):

```
test_image2 = imrotate(test_image1, -90);  
test_image3 = imresize(test_image1, [size(test_image1, 1), 2 *  
size(test_image1, 2)]);  
test_image4 = imadjust(test_image1, [.1, .9]);
```

变换 1:

左图为变换后 **detect** 所得到的结果，右图为 **mark** 所得结果。旋转实际上没有对图片信息造成改变，以区域、颜色为基础的算法并不受到影响，因此其检测结果与之前相同，准确识别了人脸。



变换 2:

与变换 1 不同的是，变换 2 的 mark 输出“意外地”将手部标识了出来，这实际是因为拉伸使得 detect 函数得到的参数为 1 的区域扩大为了原来的两倍，因此根据区域大小进行过滤的系统使得一些变大的区域能够保留下来，实际上，只要修改 detect 函数的输入限制区大小的 m、n 值即可过滤掉这些部分。



change 2



变换 3:

对图片颜色的更改没有影响最终的识别结果，从理论上分析，颜色的变化只对 detect 函数的比对阶段有影响，从下方输出结果的上图可见，识别区域相比于原来的确发生了一些变化，但仍有区域信息得到保留，或者说改变后的颜色与训练集仍具有相似性，因此得到保留。更严重的颜色变化才会导致检测失效。



change 3



4.

如果可以改变训练集，我认为可以采用“多对多”的训练、检测形式：对不同肤色的人种肤色建立资料库，在识别过程中通过其他前置信息推测人种能有效提高识别度；此外，训练集的背景选取要有所不同，不能过于一致，否则这部分信息也可能成为人脸的信息。

实验总结：

完成实验的任务所花费的精力比我想象中还要大，但也使我收获颇丰。对于信息编码解码的过程让我了解了文件的格式、压缩技术，也让我看到了信号与系统模型在具体技术中的应用，每一层的输入输出系统，本质上都能在课程中得到建模；对于人脸识别的尝试，让我感受到技术用于现实生活的乐趣，以及开发设计的灵感。实验带来的知识将对我终身有益。

也感谢老师与助教近半年的教学与指导！