

信号与系统——MATLAB 综合实验

图像处理实验报告

学号：2019011008

无 92 刘雪枫

2021 年 9 月 7 日

目录

一、	实验目的.....	1
二、	实验平台.....	1
三、	目录结构.....	1
四、	实验原理.....	2
	（一） 基础知识	2
	（二） 图像压缩编码	2
	（三） 信息隐藏	2
	（四） 人脸识别	3
五、	实验内容.....	4
	（五） 彩色图像	4
	1. MATLAB 图像处理工具箱	4
	2. 彩色图像绘制.....	4
	（六） 图像压缩编码	5
	1. 变换域改变直流分量.....	5
	2. 二维 DCT 的实现.....	6
	3. 将系数矩阵部分置零.....	8
	4. 将系数矩阵部分转置与旋转.....	10
	5. 差分编码的频率响应.....	12
	6. DC 预测误差与 Category 的关系	13
	7. 利用 MATLAB 进行 ZigZag 扫描	13
	8. 图片的分块、DCT 与量化.....	15
	9. JPEG 编码	15
	10. 计算压缩比	17
	11. JPEG 解码	18
	12. 减小量化步长	23
	13. 处理雪花图像	24
	（七） 信息隐藏	25

1. 空域隐藏.....	25
2. DCT 域隐藏.....	26
(八) 人脸识别	30
1. 人脸标准提取.....	30
2. 人脸识别.....	31
3. 图像变换.....	34
4. 重新选择标准.....	36

一、实验目的

1. 了解计算机处理和存储图像的基本知识；
2. 了解离散余弦变换的基本知识；
3. 了解 JPEG 编码的基本原理；
4. 了解信息隐藏的基本概念和方法；
5. 了解人脸识别的概念，了解人脸识别的一种方法；
6. 熟悉 MATLAB 的使用，掌握用 MATLAB 处理二维信息的基本方法；
7. 在实验中提高动手能力，培养对 MATLAB 进行信息处理的兴趣。

二、实验平台

本次实验使用 Windows 10 平台 MATLAB R2021a (64-bit) 进行实验。

三、目录结构

本次实验报告的目录结构为：

report.pdf: 实验报告文件；

src: 源代码文件夹；

resource: 实验所需资源文件夹；

hw3_1: 第一章练习题源代码文件夹；

hw3_2: 第二章练习题源代码文件夹；

hw3_3: 第三章练习题源代码文件夹；

hw3_4: 第四章练习题源代码文件夹。

四、实验原理

（一）基础知识

彩色数字图像一般用三个数字描述：R、G、B，分别代表每个像素所含的红绿蓝三基色分量。

（二）图像压缩编码

JPEG 编码需要离散余弦变换（DCT）：对于长度为 N 的向量：

$$\vec{p} = [p_0, p_1, \dots, p_{N-1}]^T$$

其离散余弦变换为：

$$\vec{c} = \vec{D}p = \sqrt{\frac{2}{N}} \begin{bmatrix} \sqrt{\frac{1}{2}} & \dots & \sqrt{\frac{1}{2}} \\ \vdots & \ddots & \vdots \\ \cos \frac{(N-1)\pi}{2N} & \dots & \cos \left(\frac{(N-1)(2N-1)\pi}{2N} \right) \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-1} \end{bmatrix}$$

对于任意矩阵 P ，先对其逐列做离散余弦变换，再对其逐行做离散余弦变换，可以得到 P 的二维离散余弦变换：

$$C = DPD^T$$

JPEG 编码中，对于 8 位灰度图（每个像素颜色取值为 0~255），编码方式如下：

先将图片每个像素减去 128，再对矩阵进行二维离散余弦变换，得到变换的系数矩阵。然后再根据给定的量化表对每个元素进行量化，得到量化后的矩阵。

最后提取出矩阵的直流分量（即左上角元素）做熵编码，再对每个矩阵的交流分量进行 Zig-Zag 扫描将其化为一维，即可对交流分量进行熵编码。得到的直流码流和交流码流，加上图像的宽度和高度就是编码结果。

（三）信息隐藏

我们可以利用图片进行信息隐藏，将信息隐藏在图片的二进制码中。

一种方法是将信息直接隐藏在像素值中，即隐藏在空域；另一种方法是将信息隐藏在变换域中，对像素矩阵离散余弦变换得到的系数矩阵并量化后得到的矩阵进行处理，隐藏信息。隐藏的方法有多种，一种方法是在每个元素中均隐藏信息，另一种方法是在部分元素中隐藏信息，还可以在 Zig-Zag 扫描得到的最后一个非零位的后方隐藏信息。

（四）人脸识别

我们可以利用颜色进行人脸识别，通过各种颜色占总颜色的比例识别人脸。在识别之前，预先提取一定量的人脸样本，计算每种颜色占总颜色的平均值，称为“训练”。在进行识别时，计算要进行识别的图片中各种颜色占总颜色的比例，并与训练得到的比例进行比对，相似度高则识别人脸。

对于彩色图像，每个像素块有三个颜色值，即 RGB 颜色值。将它们三个颜色值的二进制位依次拼接在一起，即可将三个值映射为一个值，即 $R \ll 16 + G \ll 8 + B$ 。设训练得到的各种颜色占总颜色的比例为向量 \mathbf{u} ：

$$\vec{u} = [u_1, u_2, \dots, u_N]^T$$

其中， u_i 为第 i 种颜色占总颜色的比例，因此该向量的各个元素之和为 1。对于要识别的图像，设其各种颜色占总颜色的比例为向量 \mathbf{v} ，则：

$$\vec{v} = [v_1, v_2, \dots, v_N]^T$$

由于两个向量各个元素之和均为 1，因此我们通过计算：

$$1 - \rho = 1 - \sum_n \sqrt{u_n v_n}$$

（其中 ρ 不大于 1）即可确定相似度。因此我们可以设定一个阈值，当相似度小于阈值时即确定为人脸。

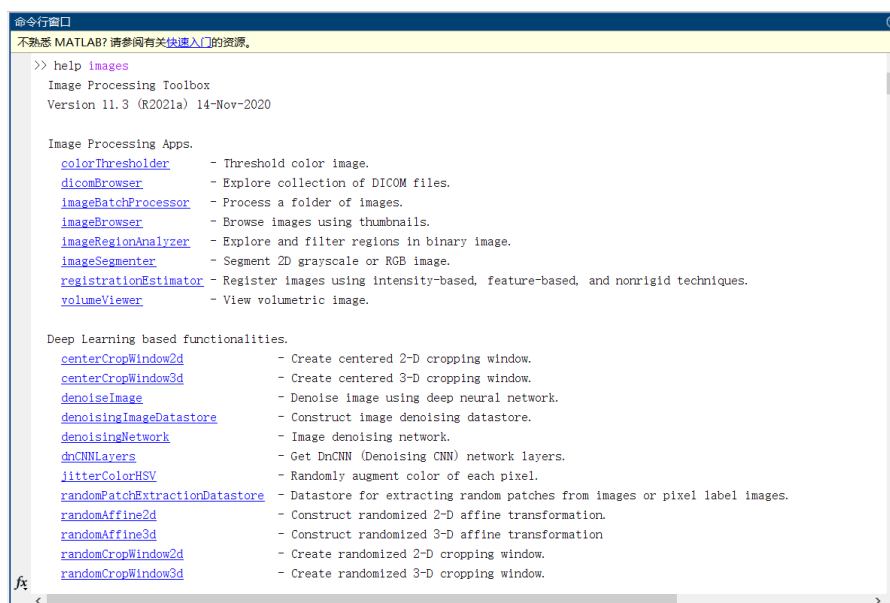
不过采用 24 位来记录颜色并进行匹配可能耗费时间和空间过大，因此我们可以将 RGB 中每个值压缩到更少的比特来存储，以提高时间和空间效率。

五、实验内容

(五) 彩色图像

1. MATLAB 图像处理工具箱

在 MATLAB 中输入指令: `help images`, 显示出很多图像处理函数:



接下来我们将利用这些图像处理函数进行实验

2. 彩色图像绘制

要以图像的中心点为圆心画圆, 需要将圆上的点的 RGB 值中的 R 值设为 255, G 和 B 均设为 0。我们以长和宽的最小值的一半为半径, 半径的 0.95 倍为内径画圆, 关键代码为:

```
circle = hall_color;
dist = (row_idx - (h+1)/2).^2 + (col_idx - (w+1)/2).^2;
radius = min(h/2, w/2)^2;
is_in_circle = dist <= radius & dist > 0.95 * radius;
draw_r_circle = cat(3, is_in_circle,
    logical(zeros(size(is_in_circle))),
    logical(zeros(size(is_in_circle))));
draw_gb_circle = cat(3, logical(zeros(size(is_in_circle))),
    is_in_circle, is_in_circle);
```

```
circle(draw_r_circle) = uint8(255);  
circle(draw_gb_circle) = uint8(0);
```

最后将其用 `imwrite` 函数绘制成位图，得到：



同样我们在图像上绘制棋盘。我们将棋盘的每一格的边长设置为 10 像素，把每个像素的横纵坐标分别除以 10 并取整得到的值对 2 的模作逻辑同或运算，结果为真则将 RGB 均设为 0，则可以得到棋盘的效果：



本问题的完整代码位于文件 `hw3_1_3_2.m` 中，得到的两个图片文件为该图片文件为 `hw3_1_3_2_circle.bmp` 和 `hw3_1_3_2_chess_board.bmp`。

（六）图像压缩编码

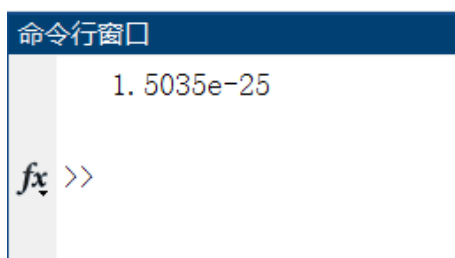
1. 变换域改变直流分量

预处理时，先将灰度值减去 128。实际上，这个过程也可以变换域进行。由于离散余弦变换具有线性性，因此两个矩阵的离散余弦变换之和等于两个矩阵和的离散余弦变换。矩阵减去 128 就相当于原矩阵与元素全为 128 的矩阵相减，因此结果等于原矩阵的离散余弦变换减去全为 128 的矩阵的离散余弦变换。注意到全为 128 的矩阵只有直流分量，因此它的离散余弦变换只有左上角的元素非零，

其余全是零。因此，只需要将原矩阵进行离散余弦变换，再把左上角元素减去一个数字即可。取 `hall_gray` 的左上角的 8×8 的部分进行验证，关键代码如下：

```
test_mat = double(hall_gray(1:8, 1:8));  
C1 = dct2(test_mat - 128);  
C2 = dct2(test_mat);  
offset = dct2(zeros([8, 8]) + 128);  
C2(1, 1) = C2(1, 1) - offset(1, 1);  
disp(sum((C1 - C2).^2, 'all'));
```

程序最后输出用两种方式计算的离散余弦变换之间的误差，误差采用各个元素误差的平方之和进行度量，程序输出结果如下：



命令窗口

1.5035e-25

fx >>

可以看到，两种方法得到的离散余弦变换相差很小，几乎为零，这也验证了上述方法的正确性。本题完整代码位于文件 `hw3_2_4_1.m` 中。

2. 二维 DCT 的实现

要实现二维 DCT，首先要计算 DCT 的变换矩阵。我将其封装为函数 `my_get_dct2_mat`，位于文件 `my_get_dct2_mat.m`。该函数接收一个整数 N 作为参数。返回大小为 N 的方阵 D 。该函数的关键代码如下：

```
D = (zeros([N - 1, N]) + [1 : 1 : N-1]') .* [1 : 2 : 2*N-1];  
D = sqrt(2 / N) * [zeros(1, N) + sqrt(1/2); cos(D * pi /  
(2*N))];
```

然后编写 DCT 变换函数，封装为函数 `my_dct2`，位于 `my_dct2.m` 文件中。在实验过程中我注意到，虽然课件中介绍的 DCT 是均为方阵进行 DCT，但是我

注意到 MATLAB 提供的 `dct2` 函数的输入参数不必为方阵。因此，我对非方阵的 DCT 进行了猜想与推导。后面我利用我自己编写的函数与 MATLAB 提供的函数进行对照，可以证明我的写法是正确的。

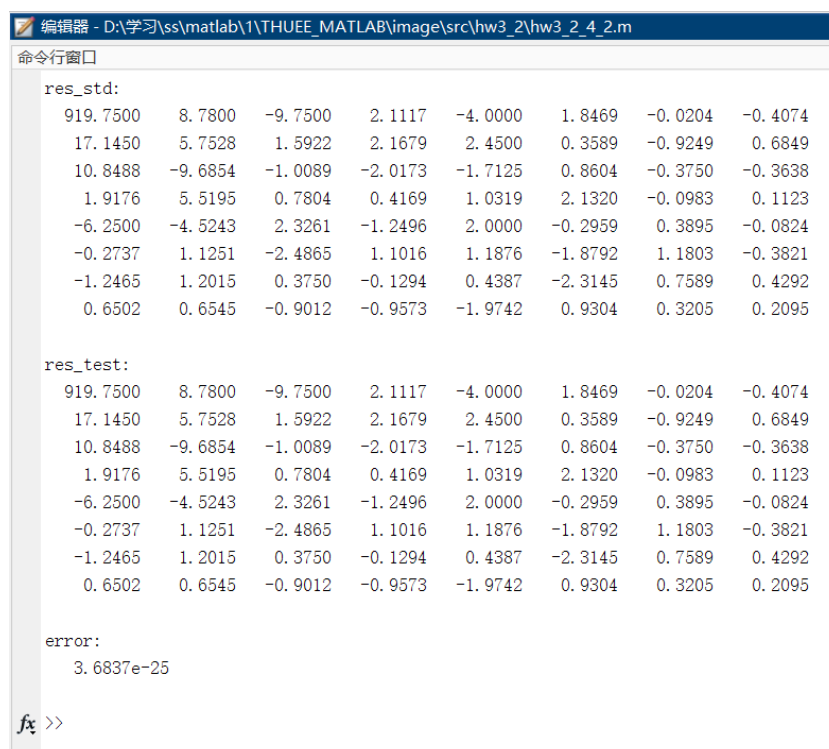
按照离散余弦变换的定义，我编写的函数的关键代码如下：

```
function C = my_dct2(P)
    [h, w] = size(P);
    C = my_get_dct2_mat(h) * double(P) * my_get_dct2_mat(w)';
end
```

下面对函数进行验证。仍然取 `hall_gray` 的左上角 8×8 的矩阵进行验证，验证代码如下：

```
test_mat = double(hall_gray(1:8, 1:8)) - 128;
res_std = dct2(test_mat);
res_test = my_dct2(test_mat);
disp('res_std: ');
disp(res_std);
disp('res_test: ');
disp(res_test);
disp('error: ');
disp(sum((res_std - res_test).^2, 'all'));
```

分别打印出 MATLAB 给出的 `dct2` 函数的结果以及我自己编写的函数的结果，并计算误差：



```

编辑器 - D:\学习\ss\matlab\1\THUUEE_MATLAB\image\src\hw3_2\hw3_2_4_2.m
命令行窗口

res_std:
    919.7500    8.7800   -9.7500    2.1117   -4.0000    1.8469   -0.0204   -0.4074
    17.1450    5.7528    1.5922    2.1679    2.4500    0.3589   -0.9249    0.6849
    10.8488   -9.6854   -1.0089   -2.0173   -1.7125    0.8604   -0.3750   -0.3638
     1.9176    5.5195    0.7804    0.4169    1.0319    2.1320   -0.0983    0.1123
    -6.2500   -4.5243    2.3261   -1.2496    2.0000   -0.2959    0.3895   -0.0824
    -0.2737    1.1251   -2.4865    1.1016    1.1876   -1.8792    1.1803   -0.3821
    -1.2465    1.2015    0.3750   -0.1294    0.4387   -2.3145    0.7589    0.4292
     0.6502    0.6545   -0.9012   -0.9573   -1.9742    0.9304    0.3205    0.2095

res_test:
    919.7500    8.7800   -9.7500    2.1117   -4.0000    1.8469   -0.0204   -0.4074
    17.1450    5.7528    1.5922    2.1679    2.4500    0.3589   -0.9249    0.6849
    10.8488   -9.6854   -1.0089   -2.0173   -1.7125    0.8604   -0.3750   -0.3638
     1.9176    5.5195    0.7804    0.4169    1.0319    2.1320   -0.0983    0.1123
    -6.2500   -4.5243    2.3261   -1.2496    2.0000   -0.2959    0.3895   -0.0824
    -0.2737    1.1251   -2.4865    1.1016    1.1876   -1.8792    1.1803   -0.3821
    -1.2465    1.2015    0.3750   -0.1294    0.4387   -2.3145    0.7589    0.4292
     0.6502    0.6545   -0.9012   -0.9573   -1.9742    0.9304    0.3205    0.2095

error:
    3.6837e-25

fx >>

```

可以看到两者的结果几乎完全一致，平方误差接近于零。这也验证了我自己编写的函数的正确性。本题完整代码位于文件 `hw3_2_4_2.m` 中。

3. 将系数矩阵部分置零

将灰度图每 8 个像素进行分块，每个块分别将 DCT 变换尔达右边四列和左边四列置零，再逆变换为图像。

由于离散余弦变换的系数矩阵中，左上角的元素代表直流和低频分量，左下角的元素代表纵向变化的高频分量，右上角的元素代表横向变化的高频分量，右下角的元素代表横向和纵向变化的高频分量。因此做出猜测：

将右侧置零，对图片整体影响不大，因为直流和低频的分量并没有受到过多的影响。但是由于横向的高频分量有所减弱，所以图片可能在横向上的色彩变化稍显模糊，但是不会过多地影响观感，因为图像的高频分量本来就比较小。

但是若将左侧四列元素置零，则会有很大的影响，因为低频分量被置为了零，所以图片原来的颜色平缓变化的部分消失了，而且图像的低频分量本来就比较小，且人眼对低频分量更敏感，因此图片质量可能严重受损。此外，由于左下角代表的是纵向变化的低频分量，因此图像纵向的灰度变化会降低；但是横向高频分量

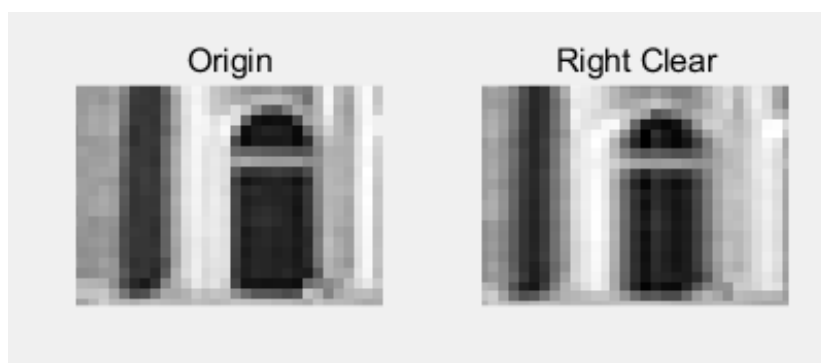
得到了很好的保留，因此图片可能会在横向上有明显的色彩变化，结果是造成图片上出现一条一条的纵向纹理。

整张图片处理结果如下：

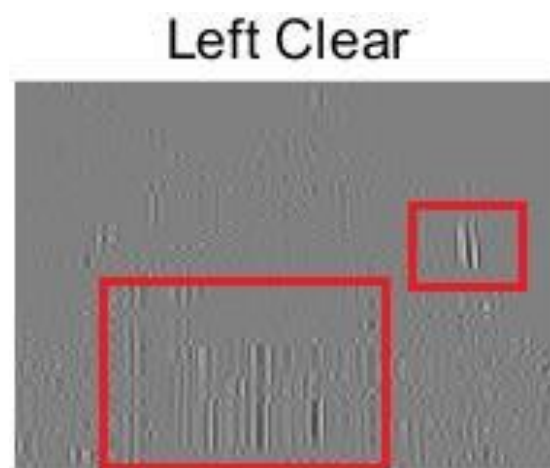


其中，左侧图为原图，中间的图为将每个小块的右侧四列置零得到的图，右边的图为将每个小块的左侧四列置零得到的图片。

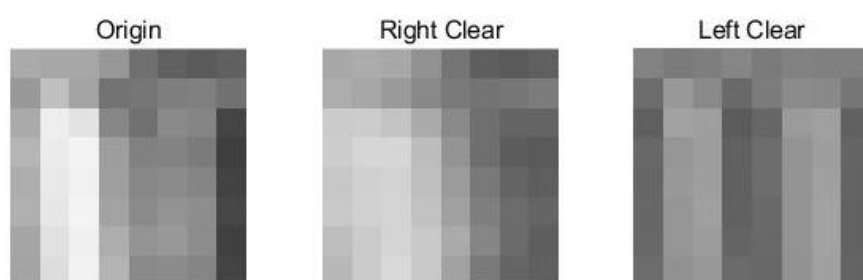
可以看到，将右侧清零得到的图片与原图差别不大，而且确实出现了轻微的重影：在礼堂的大门与门之间的柱子处由黑突然转白，即存在高频分量，此处变化尤为明显——黑与白的交界明显缓和了。将该处放大查看效果更佳：



至于清除左边的四列，可以看到得到的图片已经基本上面目全非了，并且图片上有明显的纵向纹理（下图中方框内尤为明显），与预期符合：



为了更清晰看到像素点的变化，我们选取其中的一小块进行观察。本题中取 (81:88, 73:80) 部分进行观察：



如图可以看到，原图像在横向上有黑白的剧烈变化，但在清除了右侧的四列的小块中，剧烈变化得到了很大的缓和，即横向变化变得平缓了许多；而在清楚左侧四列的小块中，几乎只留下了纵向纹理，即几乎只有横向变化，而纵向变化不明显。这些均符合最开始的预期。

本题完整代码位于文件 `hw3_2_4_3.m` 中。

4. 将系数矩阵部分转置与旋转

下面对稀疏矩阵进行转置与旋转。

理论上分析，对于矩阵的转置，导致的结果是横向的变化幅度与纵向的变化幅度，即原来具有较强横向纹理的区块会变为较强的纵向纹理，原来具有较强纵向纹理的区块会变为较强的横向纹理。

更进一步注意到离散余弦变换和逆变换的公式：

$$C = DAD^T$$

$$A = D^T CD$$

如果两边取转置：

$$C^T = DA^T D^T$$

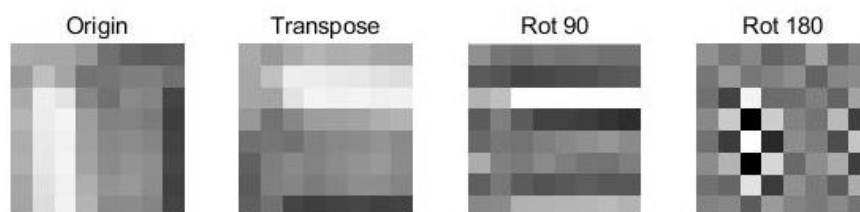
$$A^T = D^T C^T D$$

会发现，系数矩阵的转置就是原矩阵转置的离散余弦变换！因此系数矩阵的转置做逆变换得到的就是原图像的转置！

对于矩阵旋转 90 度来说，由于对于一般的图片，低频分量是很大的，但是旋转 90 度会导致低频分量的系数变为了纵向变化高频分量的系数，因此得到的图片会有较强的横向纹理。

对于矩阵旋转 180 度来说，原来低频分量的系数变为了右下角的横纵向均高频的分量的系数，因此得到的图像应该会类似于棋盘状的黑白交替。

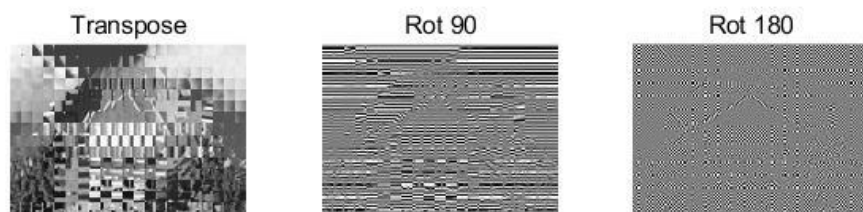
下面仍然取 (81:88, 73:80) 区块进行绘制：



可以看到，这块图片原来是有较强的纵向纹理的，即横向变化较为突出，而在系数矩阵转置后得到的图片中，横向纹理比较突出。而且仔细观察就能发现，系数矩阵转置得到的图片就是原图片的转置，这与之前的理论分析是一致的。

对于旋转 90 度的图片来说，可以看到它有很强的横向纹理，即纵向色彩的变化非常剧烈；而系数矩阵旋转 180 度得到的图片也显然呈现棋盘状。这些与之前的理论分析也都是一致的。

下面将观察完整的图形：



由于图像时分成一个个小块进行变换的，因此转置的图像呈现出了明显的分块性——即每一块取转置再拼在一起，自然造成不连贯的结果。

对于旋转 90 度的图片，可以看到，图片显然呈现明显的横向纹理；而对于旋转 180 度的图片来说，其呈现的网格状、棋盘状也是符合预期的。

本题完整代码位于文件 hw3_2_4_4.m 中。

5. 差分编码的频率响应

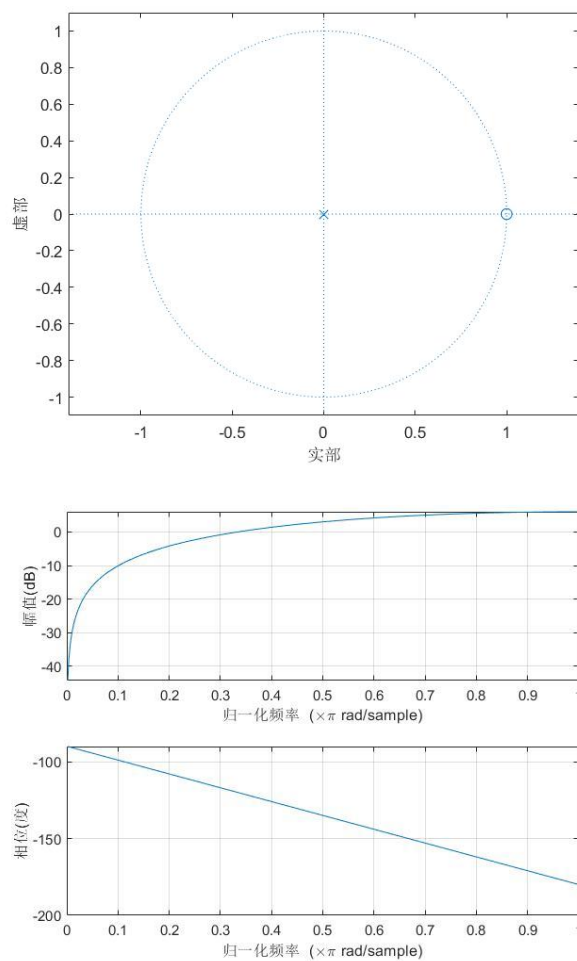
对序列取差分的相反数（注意本题 DC 编码的差分是前项减后项）：

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

因此绘制零极点图和频响的代码为：

```
a = [1];  
b = [-1, 1];  
figure(1);  
zplane(b, a);  
figure(2);  
freqz(b, a);
```

可以得到零极点图和频响：



从频响中可以看出,该滤波器是一个高通滤波器,这说明我们要滤除低频分量,即忽略小量。同时,这说明 DC 系数的高频分量更多。

本题完整代码位于文件 hw3_2_4_5.m 中。

6. DC 预测误差与 Category 的关系

根据表格可以看出,Category 是 DC 预测误差的绝对值加 1,在对 2 取对数,然后向上取整得到的数。

此外,仔细观察也会发现,Category 同时也是 DC 预测误差的二进制表示的位数。

7. 利用 MATLAB 进行 ZigZag 扫描

由于在 JPEG 编码中,进行 Zig-Zag 的矩阵均为 8×8 的矩阵,因此可以直接通过打表的方式完成 Zig-Zag。将其封装为函数 zig_zag_8,保存在文件

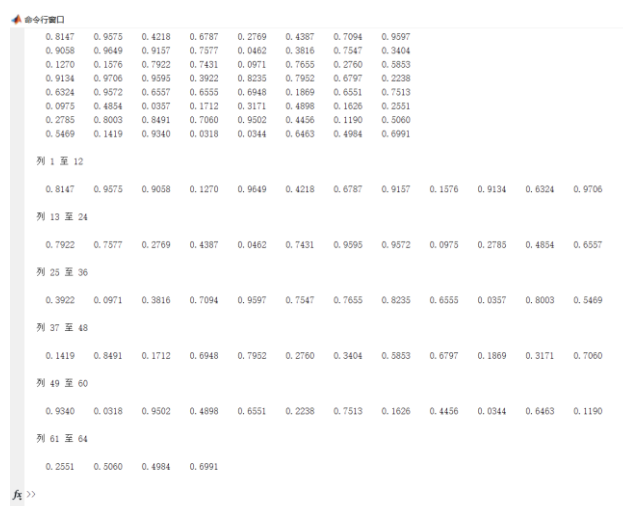
zig_zag_8.m 中。其代码如下：

```
function y = zig_zag_8(x)
    order = [ ...
        1, 2, 6, 7, 15, 16, 28, 29; ...
        3, 5, 8, 14, 17, 27, 30, 43; ...
        4, 9, 13, 18, 26, 31, 42, 44; ...
        10, 12, 19, 25, 32, 41, 45, 54; ...
        11, 20, 24, 33, 40, 46, 53, 55; ...
        21, 23, 34, 39, 47, 52, 56, 61; ...
        22, 35, 38, 48, 51, 57, 60, 62; ...
        36, 37, 49, 50, 58, 59, 63, 64];
    idx(order) = reshape([1 : 64], 8, 8);
    y = x(idx)';
end
```

下面进行测试，将一个随机的 8*8 的矩阵输入该函数：

```
A = rand(8, 8);
y = zig_zag_8(A)';
disp(A);
disp(y);
```

结果如下图所示：



可以看到，确实将矩阵成功地进行了 Zig-Zag 扫描。

该测试脚本位于文件 hw3_2_4_7.m 中。

8. 图片的分块、DCT 与量化

将图片划分为按 8×8 的大小划分，对每一块减去 128，再进行 DCT，然后按量化表量化，再逐行依次排列，关键代码为：

```
hall_gray = hall_gray(1:24, 1:16);  
img2proc = double(hall_gray) - 128;  
C = blockproc(img2proc, [8, 8], @(blk)  
zig_zag_8(round(dct2(blk.data) ./ QTAB)));  
[h, w] = size(C);  
hp = h / 64;  
res = zeros([64, hp * w]);  
for i = 1 : 1 : hp  
    res(:, (i - 1) * w + 1 : i * w) = C((i - 1) * 64 + 1 : i *  
64, 1 : w);  
end
```

则得到的 `res` 即为本题的目标。

本题完整代码位于文件 hw3_2_4_8.m 中。

9. JPEG 编码

首先我们需要一个将十进制转化为二进制数组的函数，该函数如下：

```
function y = dec2bin_array(x)  
    if x == 0  
        y = [];  
    else  
        y = double(dec2bin(abs(x))) - '0';  
        if x < 0  
            y = ~y;  
        end  
    end  
end
```

接下来进行 DC 编码。DC 编码需要将 Huffman 编码与 1 的补码进行拼接。
设上个问题得到的量化矩阵为 C_tilde （即上个问题中的 `res`），DC 编码如

下:

```
dc = C_tilde(1, :);  
diff_dc = [dc(1); -diff(dc)];  
category_plus_one = min(ceil(log2(abs(diff_dc) + 1)), 11) + 1;  
  
dc_stream = arrayfun(@(i) ...  
    [DCTAB(category_plus_one(i), 2 :  
DCTAB(category_plus_one(i), 1) + 1), ...  
    dec2bin_array(diff_dc(i))]', ...  
    [1 : length(diff_dc)]', 'UniformOutput', false);  
dc_stream = cell2mat(dc_stream);
```

得到的 `dc_stream` 即为 DC 码流。

接下来进行 AC 编码，采用的方法与 DC 编码类似，通过循环遍历每个块的信息，然后进行熵编码。

关键代码如下：

```
ac = C_tilde(2 : end, :);  
Size = min(ceil(log2(abs(ac) + 1)), 10);  
ac_stream = [];  
ZRL = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1];  
EOB = [1, 0, 1, 0];  
for i = 1 : 1 : size(ac, 2)  
    this_ac = ac(:, i);  
    this_Size = Size(:, i);  
    last_not_zero_idx = 0;  
    not_zero = this_ac ~= 0;  
    while sum(not_zero) ~= 0  
        [~, new_idx] = max(not_zero);  
        Run = new_idx - last_not_zero_idx - 1;  
        num_of_ZRL = floor(Run / 16);  
        Run = mod(Run, 16);  
        this_tab_row = Run * 10 + this_Size(new_idx);  
        ac_stream = [ac_stream, repmat(ZRL, num_of_ZRL, 1),  
ACTAB(this_tab_row, 4 : ACTAB(this_tab_row, 3) + 3),  
dec2bin_array(this_ac(new_idx))];  
        not_zero(new_idx) = 0;  
        last_not_zero_idx = new_idx;
```

```
end
    ac_stream = [ac_stream, EOB];
end
ac_stream = ac_stream';
```

得到的 `ac_stream` 即为 AC 码流。

最后将码流和图像的宽度和高度保存到文件 `jpegcodes.mat` 当中：

```
img_height = size(hall_gray, 1);
img_width = size(hall_gray, 2);
save('jpegcodes.mat', 'dc_stream', 'ac_stream', 'img_height',
    'img_width');
```

本问题的完整代码位于文件 `hw3_2_4_9.m` 中。

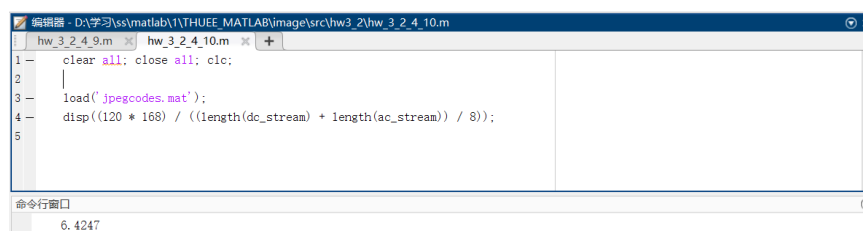
10. 计算压缩比

原图像为 120×168 的灰度图，每个像素占一字节，因此共占用空间 20160 字节。而压缩后，占用空间为 DC 码流的比特数与 AC 码流的比特数值和除以 8（化为字节数）。两者之比即为压缩比。

加载上一问题得到的脚本计算压缩比：

```
load('jpegcodes.mat');
disp((120 * 168) / ((length(dc_stream) + length(ac_stream)) / 8));
```

运行脚本，DC 码流长度为 2031 比特，AC 码流长度为 23072 比特。下图为 MATLAB 计算结果：



因此压缩比约为 6.4247。

本问题完整代码位于文件 hw3_2_4_10.m 中。

11. JPEG 解码

接下来进行解码。先进行 DC 解码。解码与编码的过程正好相反，先进行熵解码。对此，我的解决方案是，在 DC 的编码表 DCTAB 内逐行对码流进行比对，如果比对成功则开始解码。这部分解码的关键代码如下：

```
hp = img_height / 8;
wp = img_width / 8;
dc_decoding_res = zeros([hp * wp, 1]);
i = 1;
cnt = 1;
while i <= length(dc_stream)
    for j = 1 : 1 : size(DCTAB, 1)
        if DCTAB(j, 2 : DCTAB(j, 1) + 1)' == dc_stream(i : i +
DCTAB(j, 1) - 1)
            i = i + DCTAB(j, 1);
            category = j - 1;
            if category == 0
                dc_decoding_res(cnt) = 0;
            else
                Magnitude = dc_stream(i : i + category - 1);
                if Magnitude(1) == 1
                    dc_decoding_res(cnt) =
bin2dec(char(Magnitude' + '0'));
                else
                    dc_decoding_res(cnt) = -
bin2dec(char(~Magnitude' + '0'));
                end
            end
            i = i + category;
            break;
        end
    end
    cnt = cnt + 1;
end
```

其中 `dc_decoding_res` 即为熵解码的结果。

由于该结果原来是差分得到的，因此现在进行反差分：

```
dc_cum = cumsum([dc_decoding_res(1); -dc_decoding_res(2:end)]);
```

DC 解码完毕，接下来进行 AC 解码。

AC 解码方式与 DC 解码大致相同，不同的有两点：一是 AC 解码时，需要单独考虑 EOB 和 ZRL；二是 AC 的哈夫曼表中 ACTAB 的码长不是递增的，因此正在匹配的码长可能比剩余的 AC 码流总长度要长造成越界，因此需要增加越界判断。AC 解码的关键代码如下：

```
ZRL = [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1];
EOB = [1, 0, 1, 0];
ac_decoding_res = zeros([63, hp * wp]);
i = 1;
block_cnt = 1;
inner_block_cnt = 1;
while i < length(ac_stream)
    if i + length(ZRL) - 1 <= length(ac_stream) &&
sum(~(ac_stream(i : i + length(ZRL) - 1) == ZRL')) == 0
        inner_block_cnt = inner_block_cnt + 16;
        i = i + length(ZRL);
    elseif i + length(EOB) - 1 <= length(ac_stream) &&
sum(~(ac_stream(i : i + length(EOB) - 1) == EOB')) == 0
        block_cnt = block_cnt + 1;
        inner_block_cnt = 1;
        i = i + length(EOB);
    else
        for j = 1 : 1 : size(ACTAB, 1)
            if i + ACTAB(j, 3) - 1 > length(ac_stream)
                continue;
            end
            if ACTAB(j, 4 : ACTAB(j, 3) + 3) == ac_stream(i : i
+ ACTAB(j, 3) - 1)'
                inner_block_cnt = inner_block_cnt + ACTAB(j, 1);
                i = i + ACTAB(j, 3);
                Amplitude = ac_stream(i : i + ACTAB(j, 2) - 1);
                if Amplitude(1) == 1
```

```
        ac_decoding_res(inner_block_cnt, block_cnt) =  
bin2dec(char(Amplitude' + '0'));  
    else  
        ac_decoding_res(inner_block_cnt, block_cnt) =  
-bin2dec(char(~Amplitude' + '0'));  
    end  
    i = i + ACTAB(j, 2);  
    inner_block_cnt = inner_block_cnt + 1;  
    break;  
end  
end  
end  
end
```

然后将 DC 和 AC 解码得到的结果拼合成矩阵：

```
decoding_res = [dc_cum'; ac_decoding_res];
```

然后将结果按逐行重新排列为图片的行、列分布：

```
decoding_C = zeros(64 * hp, wp);  
for i = 1 : 1 : hp  
    decoding_C((i - 1) * 64 + 1 : i * 64, :) = decoding_res(:,  
(i - 1) * wp + 1 : i * wp);  
end
```

接下来要将其分块进行反 Zig-Zag。类似于 Zig-Zag，我们先编写一个用于对长度为 64 的序列进行反 Zig-Zag 的函数 `i_zig_zag_8`：

```
function y = zig_zag_8(x)  
    order = [ ...  
        1, 2, 6, 7, 15, 16, 28, 29; ...  
        3, 5, 8, 14, 17, 27, 30, 43; ...  
        4, 9, 13, 18, 26, 31, 42, 44; ...  
        10, 12, 19, 25, 32, 41, 45, 54; ...  
        11, 20, 24, 33, 40, 46, 53, 55; ...  
        21, 23, 34, 39, 47, 52, 56, 61; ...  
        22, 35, 38, 48, 51, 57, 60, 62; ...  
        36, 37, 49, 50, 58, 59, 63, 64];
```

```
idx(order) = reshape([1 : 64], 8, 8);  
y = x(idx)';  
end
```

对其进行测试：若输入为 1~64 的连续数字，则输出应该与上面的 order 数组相同：

```
>> i_zig_zag_8([1:64]')  
  
ans =  
  
     1     2     6     7    15    16    28    29  
     3     5     8    14    17    27    30    43  
     4     9    13    18    26    31    42    44  
    10    12    19    25    32    41    45    54  
    11    20    24    33    40    46    53    55  
    21    23    34    39    47    52    56    61  
    22    35    38    48    51    57    60    62  
    36    37    49    50    58    59    63    64  
  
>>
```

结果符合预期。

然后依次分块进行反 Zig-Zag、反量化、离散余弦逆变换：

```
decoding = blockproc(decoding_C, [64, 1], @(blk)  
idct2(i_zig_zag_8(blk.data) .* QTAB));
```

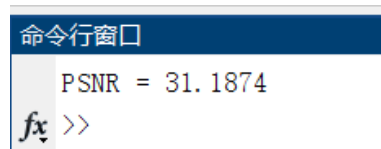
最终得到结果 decoding。将其加上亮度 128 得到图片：

```
decoding_img = uint8(decoding + 128);
```

根据公式计算结果的 PSNR：

```
MSE = sum((double(decoding_img) - double(hall_gray)).^2,  
'all') / (img_height * img_width);  
PSNR = 10 * log10(255 * 255 / MSE);  
disp("PSNR = " + PSNR);
```


输出结果为：

A screenshot of the MATLAB Command Window. The title bar says "命令行窗口". The text inside shows "PSNR = 31.1874" and the MATLAB prompt "fx >>".

```
命令行窗口
PSNR = 31.1874
fx >>
```

因此 PSNR 的值为 31.1874 dB。

绘制图片，并与原图形进行比对：

```
subplot(1, 2, 1);
imshow(hall_gray);
title('Origin');
subplot(1, 2, 2);
imshow(decoding_img);
title('Decoding');
```

得到下图：



其中左侧是原图片，右侧是解码后的图片。总体来看，两图差别不大，都能比较清晰地看出原本的情景。但是仔细看会发现，解码后的图较原图稍模糊，尤其是色彩的剧烈变化处变化变缓，这是滤掉高频分量的结果。此外，在解码后的图中，尤其是白云内部，可以看到较为明显的方块状，这可能是按 8 个像素分块量化的结果。

本问题完整代码位于文件 hw3_2_4_11.m 中。

12. 减小量化步长

将上述编码、解码功能分别封装成函数 `jpeg_encode` 和 `jpeg_decode`，并且增加了边长不足 8 的倍数时进行用零进行边界扩展的功能；并将计算 PSNR 封装为函数 `mypsnr`，将量化表除以 2 进行编码和解码：

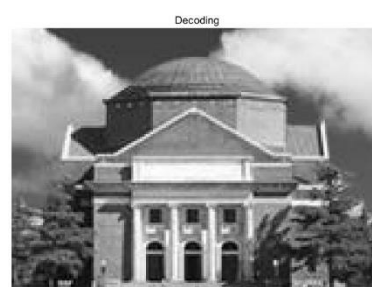
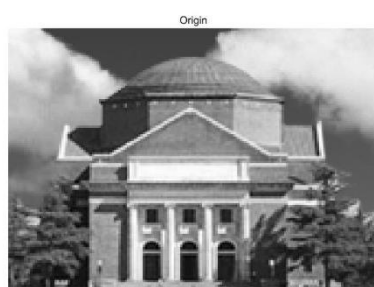
```
QTAB = QTAB / 2;  
[dc_stream, ac_stream, img_height, img_width] =  
jpeg_encode(hall_gray, DCTAB, ACTAB, QTAB);  
image_res = jpeg_decode(dc_stream, ac_stream, img_height,  
img_width, DCTAB, ACTAB, QTAB);  
disp("Compress ratio = " + (size(hall_gray, 1) *  
size(hall_gray, 2)) / ((length(dc_stream) + length(ac_stream))  
/ 8));  
disp("PSNR = " + mypsnr(hall_gray, image_res));
```

得到结果：

```
命令窗口  
Compress ratio = 4.4097  
PSNR = 34.2067  
fx >>
```

因此可以看到，压缩比约为 4.4097，略有降低，而 PSNR 为 34.2067 dB，略有升高。推测其原因，因为量化步长缩短，使得高频信息损失更少，因此 PSNR 升高；而量化步长缩短导致得到的量化值增大，因此编码结果的长度变长，减低压缩比。

绘制图片得到：

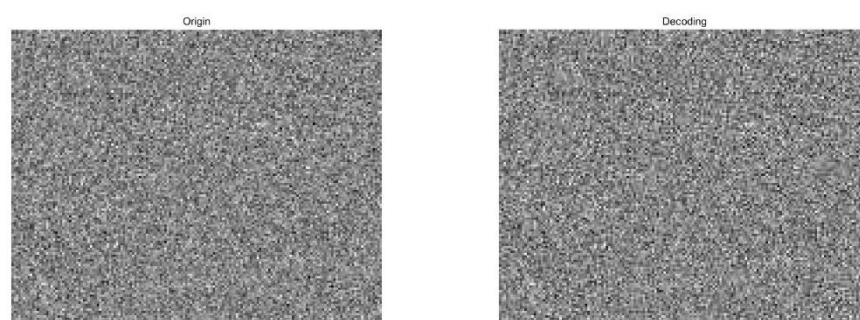


可以看到，图片质量与没有将量化步长除以 2 的时候相比，可能略有提高，但是变化不是很大，和之前的效果难以用肉眼分辨。

本问题的完整代码位于文件 `hw3_2_4_12.m` 中。

13. 处理雪花图像

将输入图片变成雪花图像，得到的图片如下图：



压缩比和 PSNR 分别为：

```
命令行窗口
Compress ratio = 3.645
PSNR = 22.9244
fx >> |
```

即压缩比为 3.645，PSNR 为 22.9244 dB。相比来看，雪花图像的压缩比较之前更小，而且 PSNR 值也更低。仔细对比两幅图，压缩并解码后的图片也失真较多，与原图片相似度降低。

究其原因，电视上雪花图的雪花是在电视屏幕上随机出现的，并且深浅度不一，因此图片的高频分量很大，故高频分量不容易被量化至 0 或其他较低的值。因此压缩比相对较低；而高频分量被量化导致高频分量存在失真，而雪花图高频分量较多，因此图片的失真比较严重，PSNR 较低。

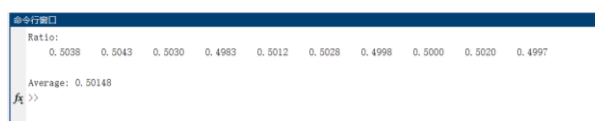
（七）信息隐藏

1. 空域隐藏

将图片进行空域信息隐藏。我们先用随机 01 序列当作信息进行隐藏，然后检查其正确率。为了让结果更准确，我们重复 10 次实验。关键代码如下：

```
test_time = 10;
[h, w] = size(hall_gray);
correct_ratio = zeros([test_time, 1]);
for i = 1 : 1 : test_time
    seq = uint8(randi([0, 1], h, w));
    secret_image = bitset(hall_gray, 1, seq);
    [dc, ac, imh, imw] = jpeg_encode(secret_image, DCTAB,
    ACTAB, QTAB);
    decoding_image = jpeg_decode(dc, ac, imh, imw, DCTAB,
    ACTAB, QTAB);
    decoding_seq = bitand(decoding_image, uint8(ones([h,
    w])));
    correct_seq = ~xor(seq, decoding_seq);
    correct_ratio(i) = sum(correct_seq, 'all') / (h * w);
end
```

运行结果：



可以看到正确率几乎都是 0.5 上下——这代表信息根本没有被获取到——有因为我们都是 01 序列，所以即使是随便猜测信息都回平均有 50% 的正确率。因此信息完全被丢失！

据此我本来存在一个猜想：随机的序列不能代表真实信息，真实信息可能有一定的规律，而随机的信息代表高频，高频分量很容易被过滤掉，因此我决定使用全 0 信息和全 1 信息进行测试。

全 1 信息的代码与结果如下：

```

hw3_3_4_1.m = open('hw3_3_4_1.m')
load('hw3_3_4_1.m')
% 测试时间
test_time = 10;
[h, w] = size(hall_gray);
correct_ratio = zeros(test_time, 1);
for i = 1 : test_time
    seq = uint8(randi([0, 1], h, w));
    % 将序列转换为图像
    seq_img = zeros(h, w);
    % 将序列转换为图像
    seq_img = zeros(h, w);
    secret_image = bitset(hall_gray, 1, seq);
    [dc, ac, iah, iam] = jpeg_encode(secret_image, DCTAB, ACTAB, QTAB);
    decoding_image = jpeg_decode(dc, ac, iah, iam, DCTAB, ACTAB, QTAB);
    decoding_seq = bitand(decoding_image, uint8(ones(h, w)));
    correct_seq = xor(seq, decoding_seq);
    correct_ratio(i) = sum(correct_seq == 1) / (h * w);
end
% 输出结果
Ratio:
0.4789 0.4789 0.4789 0.4789 0.4789 0.4789 0.4789 0.4789 0.4789 0.4789
Average: 0.47887
% 显示结果
figure

```

全 0 信息的代码与结果如下：

```

hw3_3_4_1.m = open('hw3_3_4_1.m')
load('hw3_3_4_1.m')
% 测试时间
test_time = 10;
[h, w] = size(hall_gray);
correct_ratio = zeros(test_time, 1);
for i = 1 : test_time
    seq = uint8(randi([0, 1], h, w));
    % 将序列转换为图像
    seq_img = zeros(h, w);
    % 将序列转换为图像
    seq_img = zeros(h, w);
    secret_image = bitset(hall_gray, 1, seq);
    [dc, ac, iah, iam] = jpeg_encode(secret_image, DCTAB, ACTAB, QTAB);
    decoding_image = jpeg_decode(dc, ac, iah, iam, DCTAB, ACTAB, QTAB);
    decoding_seq = bitand(decoding_image, uint8(ones(h, w)));
    correct_seq = xor(seq, decoding_seq);
    correct_ratio(i) = sum(correct_seq == 1) / (h * w);
end
% 输出结果
Ratio:
0.4789 0.4789 0.4789 0.4789 0.4789 0.4789 0.4789 0.4789 0.4789 0.4789
Average: 0.47887
% 显示结果
figure

```

可以看到它们的正确率也还都是 50% 上下，也基本上没有保留任何信息。因此，问题不在于高频分量上。我认为，问题出在量化过程。由于量化过程会将比较接近的数字映射到同一个数字上，因此相接近的数字是无法做区分的，即在量化过程中损失掉了。而我们在空域进行信息隐藏时，由于只修改最低的一位，因此对数字影响可能不大，故信息在量化过程中丢失。

可以看到，空域隐藏的抗 JPEG 编码能力极低。

本问题的全部代码位于文件 hw3_3_4_1.m 中。

2. DCT 域隐藏

下面对量化后的量化系数进行处理以隐藏信息，下面采用三种方法进行隐藏。直接的操作对象是进行量化并且进行了 Zig-Zag 且逐行排列后的矩阵。

D) 信息隐藏在每个系数的最低位

将每个系数的最低为均置为要隐藏的信息，该功能封装为 freqdom_hide1 函数，位于 freqdom_hide1.m 函数文件中，关键代码如下：

```
function Out = freqdom_hide1(In, info)
    Out = double(bitset(int64(round(In)), 1, info));
end
```

对应的解密算法封装为 `freqdom_find1` 函数，位于 `freqdom_find1.m` 函数文件中：

```
function info = freqdom_find1(In)
    info = uint8(bitget(int64(round(In)), 1));
end
```

II) 信息隐藏在部分系数的最低位

另外一种方法是只挑选部分系数隐藏信息。为了尽量减小对图像的影响，我决定选取量化矩阵 `QTAB` 中最小元素的位置用于隐藏信息。这是因为量化矩阵的元素越小，对应位置的量化系数每一个单位代表的值越小，能最大程度上减小图像的失真。基于这个思路，设计的加密代码封装为 `freqdom_hide2` 函数，位于 `freqdom_hide2.m` 函数文件中。关键代码如下：

```
function Out = freqdom_hide2(In, info, QTAB)
    [~, min_idx] = min(zig_zag_8(QTAB));
    In(min_idx, :) =
double(bitset(int64(round(In(min_idx, :)))), 1, info));
    Out = In;
end
```

对应的解密算法封装为 `freqdom_find2` 函数，位于 `freqdom_find2.m` 函数文件中：

```
function info = freqdom_find2(In, QTAB)
    [~, min_idx] = min(zig_zag_8(QTAB));
    info = uint8(bitget(int64(round(In(min_idx, :)))), 1));
end
```

III) 信息隐藏在最后一个非零位之后

基于此思路设计的加密代码封装为 `freqdom_hide3` 函数，位于 `freqdom_hide3.m` 函数文件中，关键代码如下：

```
function Out = freqdom_hide3(In, info)
    info = double(info);
    info(info == 0) = -1;
    Out = zeros(size(In));
    for i = 1 : 1 : size(In, 2)
        this_seq = In(:, i);
        if this_seq == 0
            Out(:, i) = [info(i); In(2 : end, i)];
        else
            not_zero = flipud(this_seq ~= 0);
            this_seq = flipud(this_seq);
            [~, idx] = max(not_zero);
            if idx == 1
                this_seq(1) = info(i);
            else
                this_seq(idx - 1) = info(i);
            end
            Out(:, i) = flipud(this_seq);
        end
    end
end
```

对应的解密算法封装为 `freqdom_find3` 函数，位于 `freqdom_find3.m` 函数文件中：

```
function info = freqdom_find3(In)
    info = uint8(zeros([1, size(In, 2)]));
    for i = 1 : 1 : size(In, 2)
        this_seq = flipud(In(:, i));
        if this_seq == 0
            info(i) = 0; % error
            disp('Warning: Get message error: All is zero!');
        else
            is_zero = this_seq ~= 0;
            [~, idx] = max(is_zero);
```

```
        if this_seq(idx) == 1
            info(i) = 1;
        elseif this_seq(idx) == -1
            info(i) = 0;
        else
            info(i) = 0;    % error
            disp('Warning: Get message error: Not 1 or -
1!');
        end
    end
end
end
```

下面进行测试，信息仍然采用随机的 01 数值，运行 MATLAB 代码，得到的信息准确率、压缩比以及 PSNR：

```
Compress ratio:
    2.8681    6.3924    6.1916

PSNR:
    15.4584    31.1315    28.8732

Accuracy:
    1    1    1
```

可以看到，第一种方法的压缩比和 PSNR 最低；而第二种方法和第三种方法的压缩比和失真率都和没有隐藏信息是更接近。初步推测这是因为第一种方法的信息量大，对每个系数都有所改变，因此图像失真也比较严重，影响了 PSNR；并且每个系数都改变导致了编码中 0 的数量较少以及码字变长等，影响了压缩率。

此外，第二种方法比第三种方法还要优一些，压缩率和 PSNR 略高。推测这是因为第二种方法我是选取了 QTAB 最小的元素进行隐藏编码，因此对图像的影响较小，对 PSNR 影响小；并且，第三种方法在最后附加信息，本身便会增加码长，况且为零的值大多是高频分量，因此对图像影响也会较大。

将几幅图分别画出，如下：



可以看到，图片给人的感觉与上面理论分析相一致，第一种方法失真严重，第二种方法和第三种方法和原图片差别均不大，但第三种方法比第二种方法稍模糊。原因在上面已经进行了分析。

本问题的启动脚本为文件 `hw3_3_4_2.m`。

（八）人脸识别

1. 人脸标准提取

- 样本人脸大小虽然不一样，但是我们关心的只是各种颜色占整个图片的比例，与图片的大小无关，因此不必调整为相同的大小。
- 将所有图片均取出训练，首先需要将 RGB 的三个值拼合成一个值，此功能封装为函数 `rgb2val`，位于函数文件 `rgb2val.m` 中。关键代码如下：

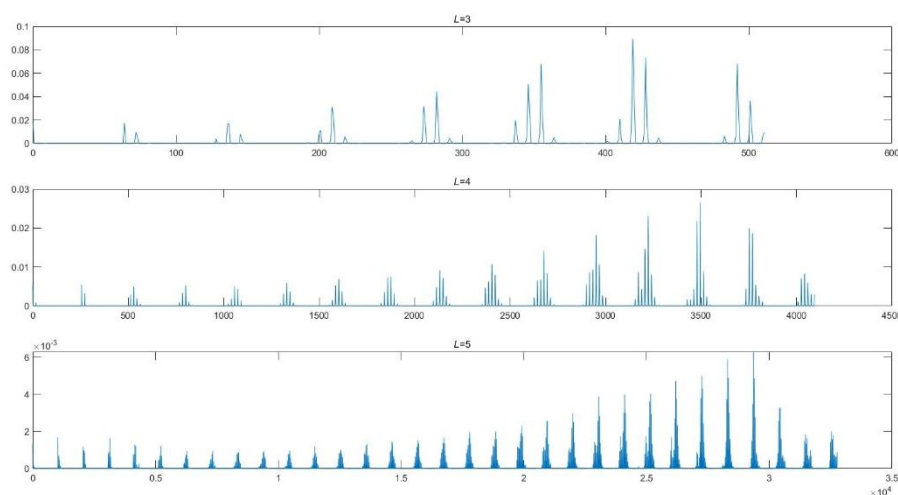
```
function val = rgb2val(r, g, b, L)
    r = floor(double(r) * 2^(L - 8));
    g = floor(double(g) * 2^(L - 8));
    b = floor(double(b) * 2^(L - 8));
    val = r * 2^(2 * L) + g * 2^L + b;
end
```

然后获取每个图片的概率密度向量，然后求平均值。获取单个图片的概率密度向量代码封装为函数 `get_property`，位于文件

get_property.m 中。代码如下：

```
function property = get_property(img, L)
    [h, w, ~] = size(img);
    img = reshape(img, h * w, 3);
    property = zeros([1, 2^(3 * L)]);
    for j = 1 : 1 : h * w
        val = rgb2val(img(j, 1), img(j, 2), img(j, 3), L);
        property(val + 1) = property(val + 1) + 1;
    end
    property = property / (h * w);
end
```

然后分别设置 L 值为 2、3 和 4，把绘制密度向量，如下图：



可以看到， L 值影响颜色采样的分辨，即精确程度。 L 值越大，不同颜色的区分度越高；而 L 值较小时，更多的相近的颜色被映射到了同一种颜色值。因此， L 值较低的向量的值可以看做时 L 值较高的向量的相应元素值周围的一些元素的值求和。

本问题的完整代码位于文件 hw3_4_3_1.m 中。

2. 人脸识别

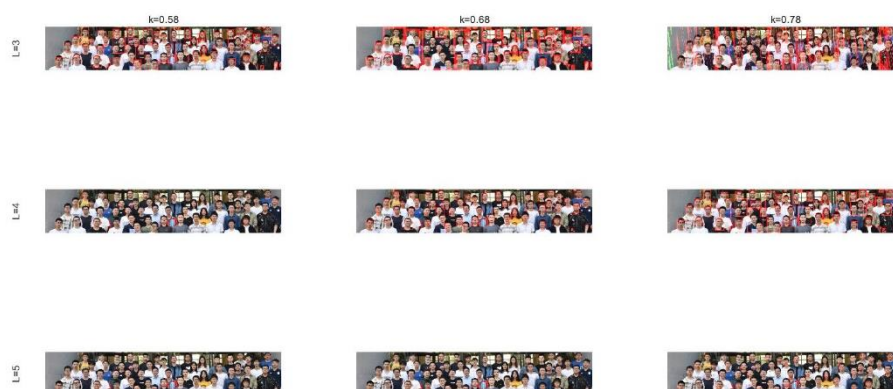
进行多人图片的人脸识别时，需要选择检测的区域范围。这里我使用小矩形

在图片上以一定的步长移动进行检测的方法,检测完毕后将重叠的矩形合并为一个大矩形。其中,移动的步长和矩形的大小经过图片实际大小调节。

选用下面的合影进行测试:



下面根据不同的 L 值和阈值进行实验 (本次取矩形的宽高均为 8, 步长均为 2):



通过图片的对比可以看到,若 L 值过大,则可能遗漏人脸或难以匹配到真正的人脸;而当 L 值过小时,则可能会将周围的非人脸匹配到;

此外,匹配的效果还与阈值有关。本次实验选取的阈值分别是 0.58、0.68 和 0.78。当阈值过小时,仍然可能遗漏人脸;当阈值过大时,也会匹配到非人脸。

经过多次调节参数,考虑到由于图片中人脸大多纵向比横向略长,因此矩形的选取为高度 10 像素,宽度 8 像素;而图片的宽度更宽,步长选取为高度方向 2 像素、宽度方向 4 像素; L 值取 4, 阈值取 0.685, 结果如下:

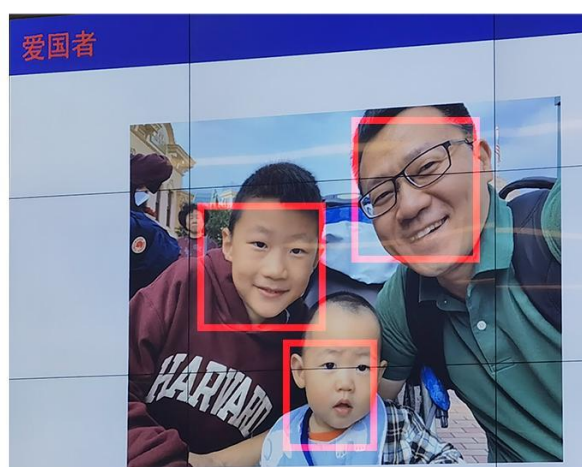


可以看到绝大多数人脸被成功识别。

下面换用不同的图片再进行实验，测试图片选用我们敬爱的谷老师：。



根据图片实际情况，采用 $L=4$ ，阈值为 0.7，检测矩形宽高均为 50 像素，步长为 20 像素，得到结果如下：



可以看到，识别效果非常好。

本问题源代码位于文件 hw3_4_3_2.m 中。

3. 图像变换

将图片顺时针旋转 90 度，结果如下图：



可以看到，检测效果基本没有发生太大变化。原因大概是我们在意的只是颜色占总区域的比例，而与方向无关，因此检测效果不会有大变化。

然后将图片宽度增大为原来的两倍。将图片宽度增大的代码如下（其中，`img_org` 为原图片，`img` 为增大后的图片）：

```
[h w ~] = size(img_org);  
img = uint8(zeros(h, 2 * w - 1, 3));  
n = mod([1 : 1 : 2 * w - 1], 2) == 1;  
img(:, n, :) = img_org;  
img(:, ~n, :) = uint8((double(img(:, [n(1:end-1), false], :))  
+ double(img(:, [false, n(2:end)], :))) / 2);
```

再进行识别，得到结果如下：



可以看到对结果的影响也不是很大，只是个别人的识别面积相对变窄了，推测是由于图片拉伸导致人脸其中一部分被检测到时难以覆盖整个脸的范围，但是影响也不大。

最后调节图片的亮度进行测试。分别将图片调亮和调暗，结果如下：



可以看到,调亮的图片对比原图变化还不是很大,但是调暗的图片相比原图,识别率差了很多,大多数人被识别的人脸面积变小,还有个别人没有被识别出来。初步推测是由于测试图片相比训练图片来说本来就偏暗,因此亮度继续调暗会导致颜色相对训练图片来说相差更多,因此不易识别。

本问题的完整代码位于文件 `hw3_4_3_3.m` 中。

4. 重新选择标准

从前面的实验中可以看到,出现的问题主要有以下几点:

首先,图片亮度会极大地影响识别率。因此,通过颜色来识别人脸是有局限的,会很大程度上受到人的肤色、光线的明暗的影响。

第二,个别的人手部和其他的一些和人肤色颜色相近建筑物可能会被识别成人脸。

因此,我认为可以通过物体的轮廓进行识别,主要看颜色的变化程度,即求图片的梯度,进而,通过对颜色变化剧烈部分取二维自相关函数来判断人脸。