# 图像处理大作业

**无 52 肖善誉 2015011009**

## 第一章 基础知识

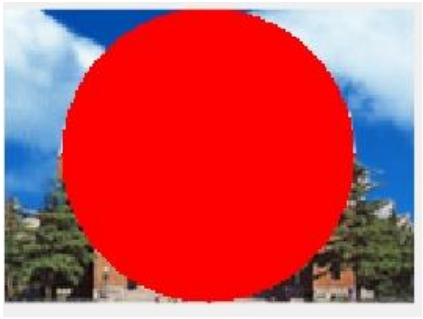
**2.** 利用 MATLAB 提供的 Image file I/O 函数分别完成以下处理：

**(a)** 利用如下代码计算圆区域：

```
x = [1 : imgSize(1)]';
x = repmat(x, [1, imgSize(2)]);
y = [1 : imgSize(2)];
y = repmat(y, [imgSize(1), 1]);
circleRegion = ((x - imgSize(1)/2).^2 + (y - imgSize(2)/2).^2) <= rad^2;
```

利用代表图像每个像素xy坐标的矩阵x, y，根据圆的定义求得画圆的区域，并利用逻辑索引将圆形区域的图像像素置为[255, 0, 0]。具体代码见 Chap1 基础知识/ex2.m。

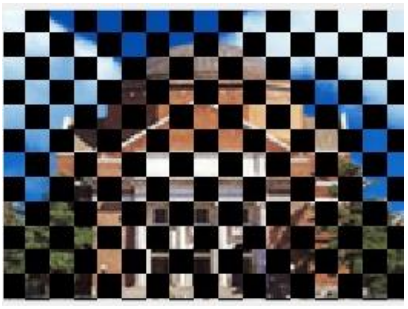
效果如下：



**(b)** 为画图黑白格，采用：分别画出横向的条纹和竖向的条纹，并对横向与竖向的条纹采用“异或”操作（即某区域只有横向或竖向条纹中的一种时，该区域才涂成黑色）。同样利用每个像素的xy坐标得到相间的横向、竖向条纹区域。关键代码如下:

```
Stride = 10;
blockRegion = xor((mod(x, Stride*2) < Stride), (mod(y, Stride*2) < Stride));
```

效果如下，具体代码见 Chap1 基础知识/ex2.m：



# 第二章 图像压缩编码

**1.**

减去 128 在变换域中不可行。在空间域减去 128 再做 DCT，相当于仅将 DCT 系数中的直流分量减去一常数。而在变换域减去 128，所有分量都发生了改变。从图片中随机选取一 8 x 8 区域测试，具体代码请看 Chap2 图像压缩编码/ex1.m，结果如下：

```
268.3750  291.2218   28.2347   93.3544   96.6250   15.5687  -43.2199    2.0235
123.0465 -264.8784   89.8593  131.2664  -15.2346  -12.8604    2.9968   22.1724
 49.6973   36.6880  -45.4051  -23.0055   -0.4531    6.0629  -24.8148    1.9809
 91.0604 -100.1628  -25.6475   82.5136  -12.2822  -13.2791   -6.8536    1.5196
  5.8750  -23.4815   49.2359  -36.2853  -10.8750   29.2197   11.4011   -0.8007
-24.6444   65.4753  -40.5616   -9.7488   19.3947  -11.7525   -6.5600    5.3334
-34.5211    0.2979    9.4352   14.8199   -8.6067    6.1539   -2.0949   -2.3635
-21.3231   19.7028   11.9366  -30.1358   10.2756    0.2715   -1.3866    0.1173

1.0e+03 *

 1.2924    0.2912    0.0282    0.0934    0.0966    0.0156   -0.0432    0.0020
 0.1230   -0.2649    0.0899    0.1313   -0.0152   -0.0129    0.0030    0.0222
 0.0497    0.0367   -0.0454   -0.0230   -0.0005    0.0061   -0.0248    0.0020
 0.0911   -0.1002   -0.0256    0.0825   -0.0123   -0.0133   -0.0069    0.0015
 0.0059   -0.0235    0.0492   -0.0363   -0.0109    0.0292    0.0114   -0.0008
-0.0246    0.0655   -0.0406   -0.0097    0.0194   -0.0118   -0.0066    0.0053
-0.0345    0.0003    0.0094    0.0148   -0.0086    0.0062   -0.0021   -0.0024
-0.0213    0.0197    0.0119   -0.0301    0.0103    0.0003   -0.0014    0.0001

1.0e+03 *

 1.1644    0.1632   -0.0998   -0.0346   -0.0314   -0.1124   -0.1712   -0.1260
-0.0050   -0.3929   -0.0381    0.0033   -0.1432   -0.1409   -0.1250   -0.1058
-0.0783   -0.0913   -0.1734   -0.1510   -0.1285   -0.1219   -0.1528   -0.1260
-0.0369   -0.2282   -0.1536   -0.0455   -0.1403   -0.1413   -0.1349   -0.1265
-0.1221   -0.1515   -0.0788   -0.1643   -0.1389   -0.0988   -0.1166   -0.1288
-0.1526   -0.0625   -0.1686   -0.1377   -0.1086   -0.1398   -0.1346   -0.1227
-0.1625   -0.1277   -0.1186   -0.1132   -0.1366   -0.1218   -0.1301   -0.1304
-0.1493   -0.1083   -0.1161   -0.1581   -0.1177   -0.1277   -0.1294   -0.1279
```

从上至下三个结果分别为：空间域，空间域减去 128，变换域减去 128。可以看出，第一个和第二个结果仅有第一个元素差一常数，其他均相同。而第三个结果则和前两个显著不同。结果和理论分析相同。

**2.**

通过构造二维 DCT 的左右乘矩阵，在进行矩阵乘法，可完成二维 DCT。具体代码请看 Chap2 图像压缩编码/ex2.m。随机从图片中选取一 6 x 8 区域测试，对比与 dct2 函数结果：

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 709.1305 | -167.2315 | -52.4222 | -32.6962 | -23.5270 | -26.4182 | 55.6159 | 19.4698 |
| 97.5851 | 61.2704 | 29.5160 | -2.0285 | 3.6084 | -3.2902 | -0.1267 | 2.6276 |
| -9.5459 | 93.6277 | 16.7996 | 52.5070 | 1.7678 | -0.4365 | -22.2660 | -8.9402 |
| 33.9193 | 20.4383 | 32.2639 | 2.3379 | 3.3198 | -7.3558 | 5.8521 | -10.1548 |
| 31.4351 | 3.5705 | 11.4145 | -5.5438 | -0.4082 | 0.1692 | 0.3536 | 0.7564 |
| -2.5851 | 1.6242 | -0.8565 | 2.3075 | -3.6084 | 1.5391 | -2.5441 | 0.3099 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 709.1305 | -167.2315 | -52.4222 | -32.6962 | -23.5270 | -26.4182 | 55.6159 | 19.4698 |
| 97.5851 | 61.2704 | 29.5160 | -2.0285 | 3.6084 | -3.2902 | -0.1267 | 2.6276 |
| -9.5459 | 93.6277 | 16.7996 | 52.5070 | 1.7678 | -0.4365 | -22.2660 | -8.9402 |
| 33.9193 | 20.4383 | 32.2639 | 2.3379 | 3.3198 | -7.3558 | 5.8521 | -10.1548 |
| 31.4351 | 3.5705 | 11.4145 | -5.5438 | -0.4082 | 0.1692 | 0.3536 | 0.7564 |
| -2.5851 | 1.6242 | -0.8565 | 2.3075 | -3.6084 | 1.5391 | -2.5441 | 0.3099 |

可以看出，结果完全一致。

**3.**

从图片中随机选取一 8 x 8 区域测试。将右 4 列全部置零与左 4 列，具体代码请看 Chap2 图像压缩编码/ex3.m。结果如下：


与原图相比，右侧 4 列置零后的结果没有发生太大改变。因为图像中高频分量很少，因此消去一部分交流分量对图像影响不大。

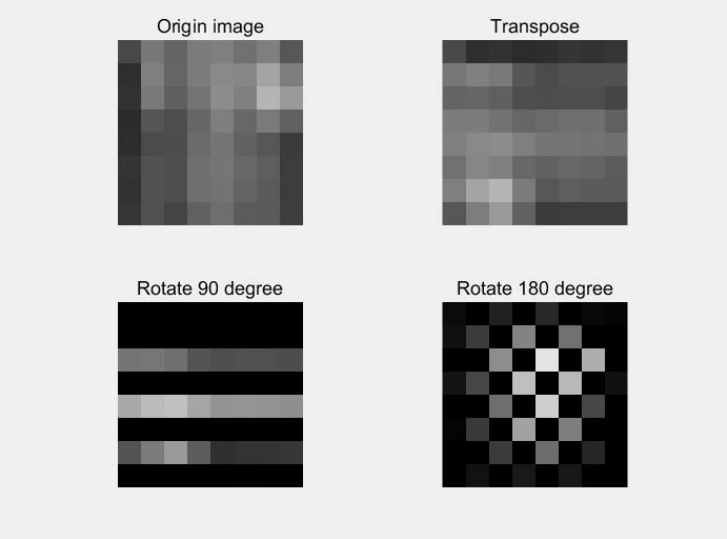
而将左侧 4 列置零则极大地影响了图像。与原图的白色不同，新图像几乎全黑。由于图像中直流分量占主导地位，左侧 4 列置零使得直流分量消失，完全改变了图像。

**4.**

由于未改变各频率分量的大小，仅改变了空间分布（x, y 交换），对 DCT 系数进行转置相当于对原图进行转置。利用 DCT 的变换酉矩阵的转置性质也可得到相同的结果。

而对系数进行旋转则影响了各频率分量的大小，得到的图像应该与原图由很大区别。

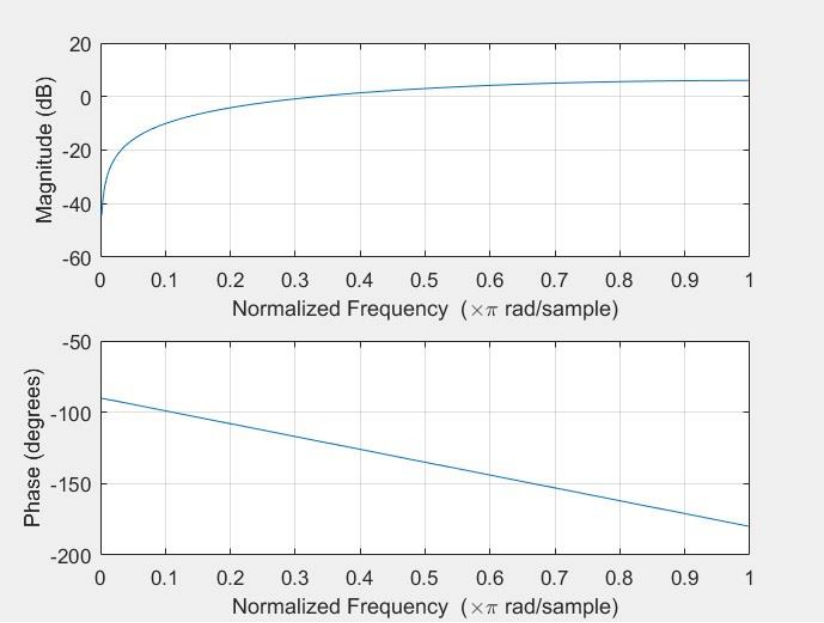
从图片中随机选取一 8 x 8 区域测试，具体代码请看 Chap2 图像压缩编码/ex4.m。结果如下：




对系数转置，得到的图像同样转置。而进行旋转，图像完全不同。且由于旋转 180 度，将直流分量转变成了最高频分量，因此使得图片高频分量很多。结果与分析一致。

**5.**

对于差分系统，该系统响应如下：




可以看出，该系统是一个**高通**滤波器。由于是高通滤波器，基于减小编码码流长度的要求下，说明 DC 系数的**低频分量**更多。

**6.**

如果预测误差为 0，则其 category 为 0。

其他情况下，category = floor( log2( abs(magnitude) ) ) + 1;

**7.**

为实现 ZigZag 扫描，设计了三种实现方法（实际为两种），具体代码请见 Chap2_图像压缩编码\en-decoder\ZigZag.m。

第一种：从已有数据中读取**索引数组**，并利用该数组索引得到 ZigZag 扫描结果。利用索引数组，扫描速度很快。

第二种：**利用算法生成索引数组，再利用索引数组进行扫描**。因为扫描的每一层，行与列序号之和为定值，利用行、列序号及其之和设计出一种算法（具体为：某个元素被扫描的次序，与其行、列序号及其之和为二次函数关系）。实现过程中没有显式利用循环，因此速度也较快。具体实现如下：

```
function ZigZagIndex = indexGen(blockSize)
    %blockSize为扫描方块的大小
    coorX = [1:blockSize]'; coorX = repmat(coorX, [1, blockSize]); %#ok<*NBRAK>
    coorY = [1:blockSize] ; coorY = repmat(coorY, [blockSize, 1]);
    coorSum = coorX + coorY;
    %用于记录扫描方向(scanDrt为1，从左下往右上扫描，否则相反)
    scanDrt = ~mod(coorSum, 2);
    f1 = @(sum, x)( (sum - 1).*(sum - 2)/2 + x );
    f2 = @(sum, x)( sum .* (sum + 1)/2 + x);
    %生成扫描左上部分每个元素被索引的顺序
    ZigZagIndex1 = (coorSum <= (blockSize + 1)) .* ...
    ( f1(coorSum, coorX) .* ~scanDrt + f1(coorSum, coorY) .* scanDrt );
    %生成扫描右下部分每个元素被索引的顺序
    ZigZagIndex2 = (coorSum > (blockSize + 1)) .* ...
    ( (blockSize^2 - f2(2*blockSize+1-coorSum, coorX - blockSize - 1)) .* scanDrt + ...
    (blockSize^2 - f2(2*blockSize+1-coorSum, coorY - blockSize - 1)) .* ~scanDrt );

    [~, ZigZagIndex] = sort(reshape((ZigZagIndex1 + ZigZagIndex2), 1, []));
    return
end
```

第三种：直接利用循环，一个一个得到 ZigZag 扫描的结果。该方法最慢，效率不高。

**8.**

由于一张图片中的块数量较少（数百至千），因此对块的遍历采用直接循环实现。ZigZag扫描利用索引数组实现。核心代码如下，具体代码请见Chap2_图像压缩编码\en-decoder\jpgEncoder.m。

```
%扩展原图大小为8的倍数，并将像素值减去128
imgSize = size(img);
imgHeight = imgSize(1); imgWidth = imgSize(2);
```

```
extImgSize = ceil(imgSize / 8) * 8;
extImg = zeros(extImgSize, 'double');
extImg(1:imgSize(1), 1:imgSize(2)) = double(img) - 128;
%初始化
[~, index] = ZigZag(ones(8), 0);
flatMat = zeros([prod(extImgSize / 8), 64]);

%DCT & ZigZag扫描
for i = 0 : extImgSize(1)/8 - 1
    for j = 0 : extImgSize(2)/8 - 1
        imgRegion = extImg(8*i+1:8*i+8, 8*j+1:8*j+8);
        coff = dct2(imgRegion);
        %量化系数
        quantCoff = round(coff ./ QTAB);
        %ZigZag
        flatMat(extImgSize(2)/8 * i + j + 1, :) = quantCoff(index);
    end
end
```

**9.**

完整 jpg 编码代码请见 Chap2 图像压缩编码\ex9.m，以及其他子函数（包括 jpgEncoder.m, encodeDC.m, encodeAC.m）。

实现过程中，利用 MATLAB 中的 **arrayfun, cellfun** 等函数代替循环，实现了较快的编码速度。对于多行（列）的数组，则利用 num2cell 函数将每一行（列）转化为 cell 数组中的一个元素后，再用 cellfun 操作。最后将 cell 数组中的元素拼接在一起就得到了编码码流。如对 AC 编码：

```
%对每一位编码
f = @(Run, Size, ampAC)([encodeRunSize(Run, Size, ACTAB), encodeAmp(ampAC)]);
%对每一行（一行代表一个block，返回cell数组，其中每个元素代表一位的编码01序列）
g = @(Run, Size, ampAC) arrayfun(f, Run, Size, ampAC, 'UniformOutput', 0);
codeAC = cellfun(g, Run, Size, ampAC, 'UniformOutput', 0);
```

使用matlab的探查器，对代码运行时间进行测试。测试结果如下：

## 探查摘要

*基于performance时间于 04-Sep-2017 18:04:23 生成。*

| 函数名称 | 调用次数 | 总时间 | 自用时间* | 总时间图 (深色条带 = 自用时间) |
|---|---|---|---|---|
| jpgEncoder | 1 | 0.567 s | 0.008 s | |
| encodeAC | 1 | 0.457 s | 0.024 s | |
| ...n(f,Run,Size,ampAC,'UniformOutput',0) | 315 | 0.406 s | 0.068 s | |
| ...ze(Run,Size,ACTAB),encodeAmp(ampAC)]) | 4112 | 0.338 s | 0.049 s | |
| encodeAC>encodeAmp | 4112 | 0.266 s | 0.027 s | |
| str2num | 4427 | 0.218 s | 0.098 s | |
| str2num>protected_conversion | 4427 | 0.119 s | 0.119 s | |
| encodeDC | 1 | 0.058 s | 0.011 s | |
| dec2bin | 4427 | 0.053 s | 0.053 s | |
| ...TAB(2:bitLen+1)==1,encodeMag(magDC)]) | 315 | 0.040 s | 0.005 s | |
| dct2 | 315 | 0.039 s | 0.008 s | |
| encodeDC>encodeMag | 315 | 0.036 s | 0.005 s | |
| images\private\dct | 630 | 0.031 s | 0.031 s | |
| encodeAC>encodeRunSize | 4112 | 0.023 s | 0.023 s | |
| encodeAC>h | 315 | 0.014 s | 0.014 s | |
| num2cell | 6 | 0.011 s | 0.011 s | |
| encodeAC>computeRun | 315 | 0.005 s | 0.005 s | |
| ZigZag | 1 | 0.004 s | 0.003 s | |
| encodeAC>@(array)(array(array~=0)) | 630 | 0.004 s | 0.004 s | |
| int2str | 1 | 0.001 s | 0.001 s | |
| pwd | 1 | 0.000 s | 0.000 s | |

编码一次运行总时长0.567秒。其中时间占比最大的为**encodeAC函数**（对AC分量进行编码），占0.457秒。继续往下分析。

**子集**(调用的函数)

| 函数名称 | 函数类型 | 调用次数 | 总时间 | % 时间 | 时间 绘图 |
|---|---|---|---|---|---|
| encodeAC>encodeAmp | 子函数 | 4112 | 0.266 s | 78.8% | |
| encodeAC>encodeRunSize | 子函数 | 4112 | 0.023 s | 6.7% | |
| 自用 时间 (内置项、开销等) | | | 0.049 s | 14.5% | |
| 总计 | | | 0.338 s | 100% | |

对**AC分量的的幅度编码**用时最多，对run/size编码反倒用时很短（一开始令我十分困惑）。再往下探查**encodeAmp**中各行代码用时，发现：


大量的时间耗费在**str2num**函数上。原来，对run/size的编码直接查边（ACTAB）,因此运行速度很快。而对AC幅度编码时，由于matlab将十进制数转化为二进制数时，是将其**转为化对应的01字符数组**，而无法直接转化为数字（数组），因此还需调用一次str2num函数来实现转化。

如果想要优化代码运行速度，可考虑的方案有：底层重新实现将十进制数转化为01数字数组的函数，如利用c++混合编程完成该功能。

**10.**

具体代码请见Chap2_图像压缩编码\ex10.m，及函数Chap2_图像压缩编码\en-decoder\computeCompressRatio.m函数。

DC编码2054位，AC编码23072位，因此**压缩比为120*168*8/（2054 + 23072）= 6.4188**。

**11.**

完整解码代码请见 Chap2_图像压缩编码\ex11.m, 以及其他子函数（包括 jpgDecoder.m, decodeDC.m, decodeAC.m, huffmanDecodeDC.cpp, huffmanDecodeAC.cpp）。

为实现快速解码，**利用 matlab 构建用于解码的二叉树数组，huffman 树的解码过程由C++混合编程实现**（huffmanDecodeDC.cpp, huffmanDecodeAC.cpp）。

以 DC 解码为例，幅度编码最长为 9 位，因此需要一个长度为 2^(9+1)的数组表示解码二叉树。各节点有一相应值，代表幅度一位 DC 分量编码的 category（某节点未编码，则值为-1）。

由解码过程可以发现，jpeg 编码对二进制码流的错误十分敏感。下一个分量的解码依赖上一个分量的解码结果。如果二进制码流中少了一个二进制位，则解码过程完全乱套了。

使用matlab的探查器，对代码运行时间进行测试。测试结果如下：

| 函数名称 | 调用次数 | 总时间 | 自用时间* | 总时间图<br>(深色条带 = 自用时间) |
|---|---|---|---|---|
| jpgDecoder | 1 | 0.132 s | 0.011 s | |
| decodeAC | 1 | 0.049 s | 0.001 s | |
| decodeAC>decodeACTree | 1 | 0.048 s | 0.004 s | |
| num2str | 172 | 0.046 s | 0.007 s | |
| ...bin2dec(num2str(ACTAB(4:depth+3)')')) | 160 | 0.043 s | 0.002 s | |
| idct2 | 315 | 0.041 s | 0.007 s | |
| int2str | 173 | 0.040 s | 0.040 s | |
| images\private\idct | 630 | 0.034 s | 0.034 s | |
| decodeDC | 1 | 0.024 s | 0.002 s | |
| decodeDC>decodeDCTree | 1 | 0.021 s | 0.002 s | |
| ...bin2dec(num2str(DCTAB(2:depth+1)')')) | 12 | 0.014 s | 0.002 s | |
| bin2dec | 172 | 0.008 s | 0.008 s | |
| ZigZag | 1 | 0.007 s | 0.005 s | |
| num2cell | 4 | 0.006 s | 0.006 s | |
| conv | 1 | 0.001 s | 0.001 s | |
| huffmanDecodeAC (MEX-file) | 1 | 0.000 s | 0.000 s | |
| pwd | 1 | 0.000 s | 0.000 s | |
| huffmanDecodeDC (MEX-file) | 1 | 0.000 s | 0.000 s | |

解码过程总用时 0.132 秒，其中主要耗时也包括 **num2str 函数**，以及 **idct2 函数**。而利用 huffman 树解码（huffmanDecodeDC.cpp, huffmanDecodeAC.cpp）所用时间则极短。

解码后，图片效果比较如下：


PSNR 为 34.8926。峰值信噪比较高，压缩效果较好。

主观来看，两张图片看上去几乎没有差别。但有的地方也可以看到，压缩降低了图像质量。如压缩后的图片，中间正上方的天空与建筑交接处，由一个个“正方形的小块”，这是由

于压缩是以 8x8 的块进行的。与原图相比，图片右下角的树木的细节也不如原图丰富。

**12.**

将量化步长减小一半后，PSNR 上升到 37.2810。而 DC 编码长度增加至 2423 位，AC 编码增加至 33946 位。压缩比减小至 4.4345。

图片效果对比：

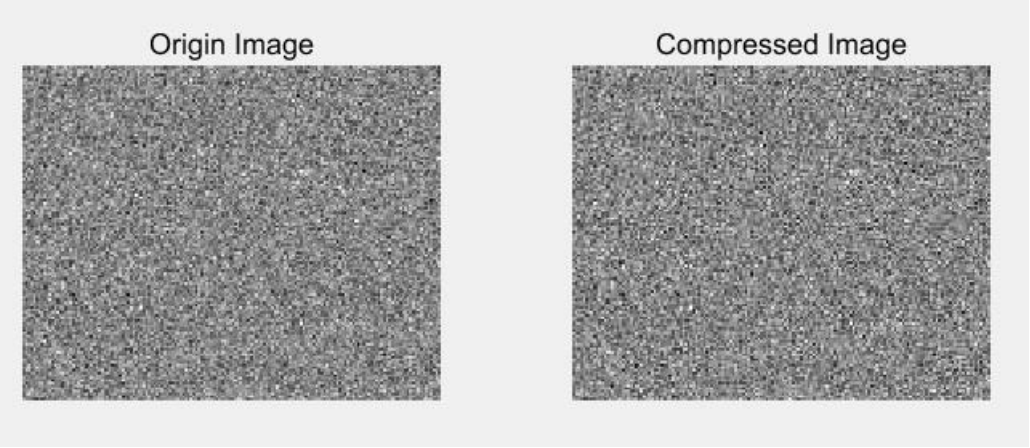

天空与建筑交接处的一个个“正方形的小块”没有那么明显，树木的质量也更好。但仍不如原始图像。

**13.**

雪花图像的压缩比为3.6407，PSNR为29.5601。与测试图像相比，压缩比低，压缩后质量也不如测试图像。

雪花图像的高频分量很多，因此压缩时丢失了大量交流分量的信息，使得压缩后恢复出的图像效果较差。且交流分量多，也不利于压缩。

图片比较如下：



# 第三章 信息隐藏

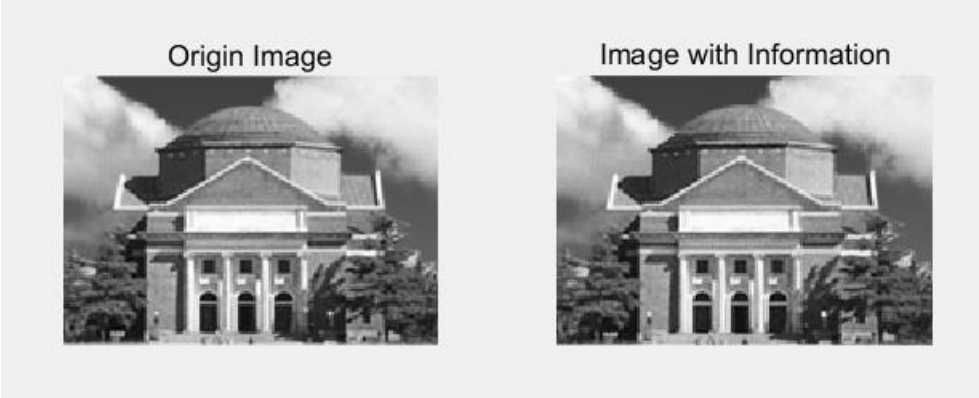
**1.**

从图片最左上方的像素开始，将信息转化为相应二进制并隐藏在像素二进制的最低位。具体代码请见 Chap3 信息隐藏\ex1.m 及 spatialInfoHide.m，spatialInfoExtract.m 函数。

在图片中插入字符串信息："**I love you**"。先尝试直接从直接从原图中提取信息，可提取到原信息。

将图像经过压缩后，尝试提取信息，发现提取的信息是"**B $** "，为乱码。说明压缩完全影响了原来隐藏的信息。空间域信息隐藏没有抗 jpeg 编码能力。

插入信息的图片与原图比较：




完全看不出差别。

**2.**

具体代码请见 Chap3 信息隐藏\ex2.m 及 DCTInfoHide.m，DCTInfoExtract.m 函数。三种在变换域隐藏信息的方法，对应于 DCTInfoHide 函数中 mode 参数（mode = 1, 2, 3）

运行 ex2.m，三种信息隐藏方法均提取到了相应信息。对三种隐藏方法分析与测试如下：

嵌入字符串信息"Love"，测试结果如下：

第一种方法压缩比和 PSNR 分别为 6.4033，34.8712。
第二种方法压缩比和 PSNR 分别为 6.3434，34.6558。
第三种方法压缩比和 PSNR 分别为 6.3944，34.7996。

第一种方法，可隐藏的信息量最大，最多每 8bits 可隐藏 1bit 的信息。当信息量较大时，交流分量中 0 的数量会大量下降，使得图像的压缩比下降。且与其他两种方法相比，由于这种方法改变了所有 DCT 分量，因此对图像质量影响最大。但当信息量比较小时，该种方法影响的 block 数少，与其他两种方法相比，可能压缩比更高，质量较好。由上述测试可以看出，信息量比较少时，第一种方法压缩比和质量都是最高的。

而第二种方法，可隐藏的信息量在中等水平。但由于影响的块数较多，而且全部影响最高频的数个分量，导致隐藏信息的图片与原图有更大的差异，使得压缩比和质量都最低。

第三种方法，隐藏信息量极少，最多仅能隐藏与图片 block 数相同的 bit 数。由于仅影响每个 block 中最后一个非零分量附近的分量大小，因此对图像的压缩比和质量影响不大。

测试发现，第三种方法压缩比和 PSNR 处于中等水平。

# 第四章 人脸识别

**1.**

**（a）**缩放图片不改变颜色特征的分布，颜色特征具有**尺度不变性**。因此不需要将图片调整到相同大小。

**（b）**L 每增加 1，矢量 v 的长度就增加 8 倍。

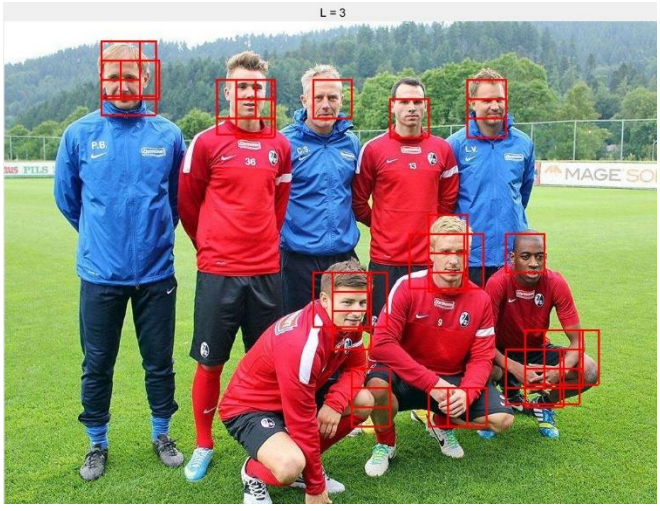
**2.**

选取 sample.jpg 作为测试图片。具体代码请见 Chap4 人脸检测\ex2.m 及函数 computeFeature.m, faceTrain.m, computeAngle.m, faceDetect.m, drawBox.m。

对于人脸的检测，采用滑动窗口进行检测。从固定大小检测窗从图片最左上方开始，每次滑动固定像素位置，每次滑动后，对检测窗区域进行人脸检测。为检测不同大小的人脸，且颜色特征具有尺度不变性，使用多种不同大小的检测窗进行检测。

演示中使用[30 30] * 2^n，n = 0, 0.5, 1，大小的检测窗对全图进行检测。同时设置距离阈值为 0.3。演示结果如下：

L = 3 时：




可以发现：人脸区域基本上都被检测出来，但是有大量重叠的矩形框。且下方运动员腿部位置的皮肤也被识别为人脸。这是因为 L = 3 时，我们仅适用 RGB 每个分量 8bits 中的前 3bits，对于颜色的分辨能力相对较差，使得人脸识别的错误检测率很高。为减少误检率，可以减小阈值。调整参数，设置阈值为 0.26。结果如下：


被认为是人脸的矩形框数量明显减少，但同时，也有一张人脸被漏检。可以得出结论，识别出所有人脸，与检测出人脸的准确率不可兼得。

L = 4 时：


L = 4 时，对人脸颜色的分辨能力更高。检测出的人脸区域都是正确的，但也漏掉了许多人脸。因此考虑增大阈值至 0.35：


同样出现了漏检的情况，而且误检率也有所上升。

当 L = 5 时，由于对颜色的分辨率过高，出现了完全无法检测到人脸的情况：


**3.**

具体代码请见 Chap4_人脸检测\ex3.m 及函数 computeFeature.m, faceTrain.m, computeAngle.m, faceDetect.m, drawBox.m。

旋转：


识别人脸的效果还不错。因为颜色特征是旋转不变的，旋转图片后不改变某个区域的颜色分布。

拉伸：


拉伸图片后，被检出的人脸数目明显增多。原因是拉伸使得人脸面积增大，滑动检测窗进行检测时，检测出人脸的概率增大。

改变颜色：


改变颜色后，完全无法检测出人脸。说明以颜色作为人脸的检测标准对图片色彩十分敏感。当颜色改变时，检测率大大下降。

**4.**

重新选取人脸训练标准，应选取正常光影下，肤色比较正常的正脸。而且由于训练出的模型对颜色敏感，最好用于训练的人脸颜色有一定的差异性，使得训练的模型鲁棒性更好。

# 总结

本次图像处理大作业内容相当丰富，在极大地提高 MATLAB 运用与编程水平的同时，还让我们初步了解了图像处理和信息编码领域。

本次实验完全由本人单独完成。由于与图像有关的操作和处理基本上涉及矩阵、多维数组的操作，因此本次实验大大提高了我的 matlab 应用水平，熟悉了 matlab 的各种基本操作（如数组、逻辑索引等），特别是学会了利用强大的“help”查询各种未见过的函数的功能。例如，我花费了大量时间研究如何用 arrayfun、cellfun 等函数代替直接进行大量循环，提高了程序的运算速度。同时，学会了如何利用 matlab 与 c/c++混合编程，在不得不进行大量简单的循环操作的地方利用 c/c++重写，优化速度。在 jpeg 码流解码过程中，利用 huffman 树对码流解码的算法便是用 c++编写完成的。

这次大作业也让我体会到了 matlab 的强大之处：对于矩阵、多维数组运算的强大支持。底层可能需要利用大量循环的操作（如两个矩阵的每个元素相乘），matlab 中只写需要一个特殊的运算符，让编程者可以专注于算法本身，同时提高开发效率。

相信进过此次 matlab 大作业，可以为以后使用 matlab 解决更多专业相关的难题打下良好的基础。