Matlab 图像处理实验报告

暮月

2020 年 8 月 3 日

目录

1 基础知识

1.1 练习题

利用 MATLAB 提供的函数完成以下任务

(a) 以测试图像的中心为圆心，图像的长宽中较小值的一半为半径画一个红颜色的圆

经在文档中搜索，Computer Vision Toolbox 中存在函数insertShape可以直接在图像中绘制圆形。下方代码用于绘制满足要求的一个红色半透明圆形：

```
[h, w, c] = size(img);

pos = [[w, h] / 2, min([w, h] / 2)];
img_circle = insertShape(img, 'Circle', pos, ...
                         'color', 'red', 'Opacity', 1);
```

此外，还可以通过圆的一般方程或者参数方程绘制：

```
circle_mask = ((x-pos(1)).^2 + (y-pos(2)).^2 >= pos(3).^2+60) ...
              |((x-pos(1)).^2 + (y-pos(2)).^2 <= pos(3).^2-60);
img_circle = img .* circle_mask;
img_circle(:,:,1) = img_circle(:,:,1) + ~circle_mask;

phi = 0:1:359;
x_ = max(min(round(pos(1) + pos(3) * cos(phi)), w), 1);
y_ = max(min(round(pos(2) + pos(3) * sin(phi)), h), 1);
img_circle = img;
img_circle(sub2ind(size(img), y_, x_, ones(1, 360))) = 1;
img_circle(sub2ind(size(img), y_, x_, 2 * ones(1, 360))) = 0;
img_circle(sub2ind(size(img), y_, x_, 3 * ones(1, 360))) = 0;
```

(b) 将测试图像涂成国际象棋状的“黑白格”的样子，其中“黑”即黑色，“白”即意味着保留原图

只需构造一个分块为 1 或 0 的矩阵，再与图像矩阵进行逐元素乘，便可让图像分块。即类似：

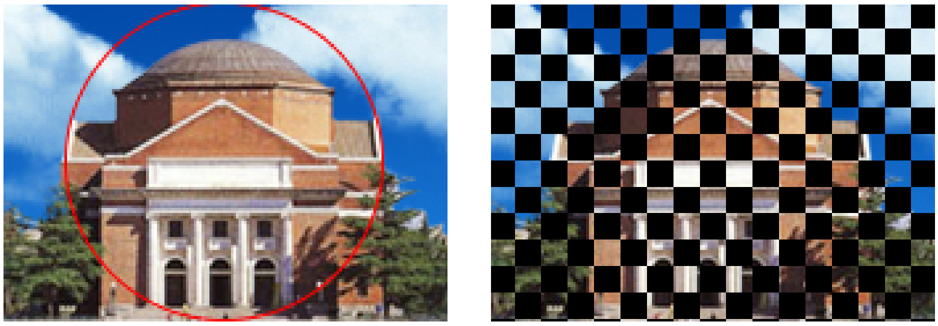
$$\mathrm{Img}.*\begin{bmatrix} 1 & \cdots & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 1 & \cdots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} \mathrm{Img}_1 & 0 \\ 0 & \mathrm{Img}_2 \end{bmatrix}$$

故只需将每个像素的坐标先除以格宽，再对 2 求余，便可以得到横纵方向上的 01 分布。然后将这个 01 分布进行异或，便可以得到一个 01 组成的块相间分布的矩阵。最后与图像矩阵进行逐元素乘得到所需结果。核心代码如下：

```
[x, y] = meshgrid(1:w, 1:h);

grid_width = 10;
x_mask = mod(floor(x / grid_width), 2);
y_mask = mod(floor(y / grid_width), 2);
```

```
grid_mask = double(xor(x_mask, y_mask));
img_grid = grid_mask .* img;
```

最终结果如图1



(a) 绘制圆形 (b) 绘制棋盘

图 1: 基础知识练习图像结果

2 图像压缩编码

2.1 图像预处理

对图像的预处理和二维 DCT 变换结合，记从原始图像中取得小块为 P，DCT 算子为 D，最终系数为 C，则过程为

$$
\begin{aligned}
C &= D\cdot(P-\begin{bmatrix}128 & \cdots & 128\\ \vdots & \ddots & \vdots\\ 128 & \cdots & 128\end{bmatrix})\cdot D^T\\
&= DPD^T - D\begin{bmatrix}1\\ \vdots\\ 1\end{bmatrix}128\begin{bmatrix}1 & \cdots & 1\end{bmatrix}D^T
\end{aligned}
$$

将矩阵算子 D 与全 1 向量乘法展开，且注意到 D 中除了第一行外行和为 0

$$
D_{N\times N}\begin{bmatrix}1\\ \vdots\\ 1\end{bmatrix} = \sqrt{\frac{2}{N}}\begin{bmatrix}N\sqrt{\frac{1}{2}}\\ \sum_{i=1}^{N}\cos\frac{(2i-1)\pi}{2N}\\ \vdots\\ \sum_{i=1}^{N}\cos\frac{(N-1)(2i-1)\pi}{2N}\end{bmatrix} = \begin{bmatrix}\sqrt{N}\\ 0\\ \vdots\\ 0\end{bmatrix}
$$

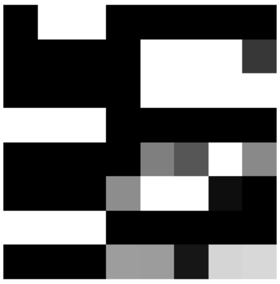
代入 C 的计算式，得变换域的预处理方法

$$
C = DPD^T - 128\begin{bmatrix}N & 0_{1\times(N-1)}\\ 0_{(N-1)\times 1} & 0_{(N-1)\times(N-1)}\end{bmatrix}
$$

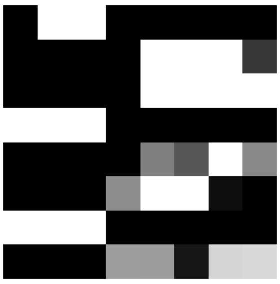
化为 Matlab 代码，在图像中随机选取一块 8×8 的区域，分别进行空域和频域的处理，再使用 2-范数衡量差异

```
width = 8;
[h, w] = size(hall_gray);
D = dctmtx(width);

piece_x = floor(rand * (h - width) + 1);
piece_y = floor(rand * (w - width) + 1);
test_piece = double(hall_gray(piece_x(piece_x  + width - 1), ...
                              piece_y:(piece_y  + width - 1)));

% spatial domain
piece_sd = test_piece - 128;
dct_sd = D * piece_sd * D';

% frequency domain
dct_fd = D * test_piece * D';
dct_fd(1,1) = dct_fd(1,1) - 128 * width;

disp(norm(dct_sd - dct_fd));
```

执行程序，得到的范数约为 1.5×10^{-12}，随机选取的小区域处理后的结果示例如图2



(a) 空域处理



(b) 频域处理

图 2: 图像 DCT 变换预处理结果

显然两种处理方式没有区别，都完成了预处理工作。考虑到代码的易读性，后面将更多采用空域处理的方式。

2.2 二维 DCT 的实现

分别实现dct_mat和dct_2两个函数，用于生成矩阵算子 D 和进行二维 DCT。由于考虑的是对图片的正方形区域进行 DCT，所以函数中进行判断确保输入为方阵。

```
function D = dct_mat(N)
% DCT_MAT DCT N * N operator
% param N: width, height of the piece
% return D: N * N square matrix
```

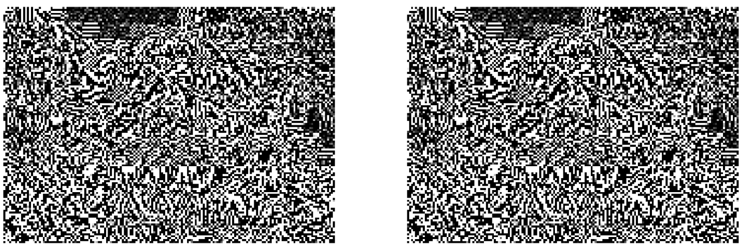
```
t = 0 : N - 1;
D = cos(t' * (2 * t + 1) * pi / (2 * N));
D(1,:) = sqrt(0.5);
D = D * sqrt(2 / N);
end
```

```
function C = dct_2(P)
% DCT_2 2D discrete cosine transform
% param P: square matrix
% return C: transformed result

[w, h] = size(P);
assert(w==h, "P must be a square matrix");
P = double(P);
D = dct_mat(w);
C = D * P * D';
end
```

再对`hall_gray`分别用 Image Processing Toolbox 中的`dct2`和前面实现的`dct_2`进行处理，使用 2-范数衡量差异。

```
mat_dct2 = @(block_struct) dct2(block_struct.data);
my_dct2 = @(block_struct) dct_2(block_struct.data);

mat_c = blockproc(img, [width width], mat_dct2, ...
                'PadPartialBlocks', true);
my_c = blockproc(img, [width width], my_dct2, ...
                'PadPartialBlocks', true);
```

最终 2-范数差异为 1.1133×10^{11}，可以认为功能一致。对`hall_gray`处理的结果见图3

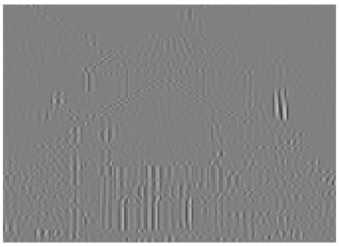


(a) Matlab 方法　　(b) 自行实现方法

图 3: 二维 DCT 变换

2.3 二维 DCT 系数矩阵性质 1

将 DCT 处理后的系数矩阵左右四列分别置零，再经逆变换得到图像如图4



(a) 左侧置零

(b) 右侧置零

(c) 不置零

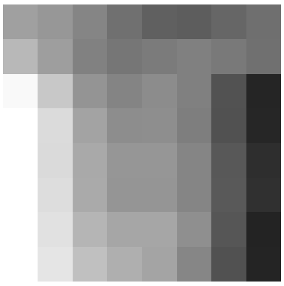
(d) 原始图像

图 4: 二维 DCT 变换系数矩阵变化的影响 1

可以看到，左侧置零的结果图4a损失了大量人用来理解图像的原始信息，右侧置零的结果图4b则仍可清晰辨认。从 DCT 变换的原理来看，系数矩阵 C 的左上角为直流和低频分量，左下角为竖直高频和水平低频分量，右上角为竖直低频和水平高频分量，右下角为高频分量。当左侧置零时，水平方向的低频分量被消去，只剩高频分量，表现在测试图片上就是保留了明暗剧烈变化处的分界线；当右侧置零时，水平方向的高频分量被消去，只剩低频分量，表现在测试图片上就是明暗剧烈变化处模糊化。

提取横坐标 57-64，纵坐标 81-88 的区域进行观察，如图5

(a) 左侧置零



(b) 右侧置零

(c) 不置零

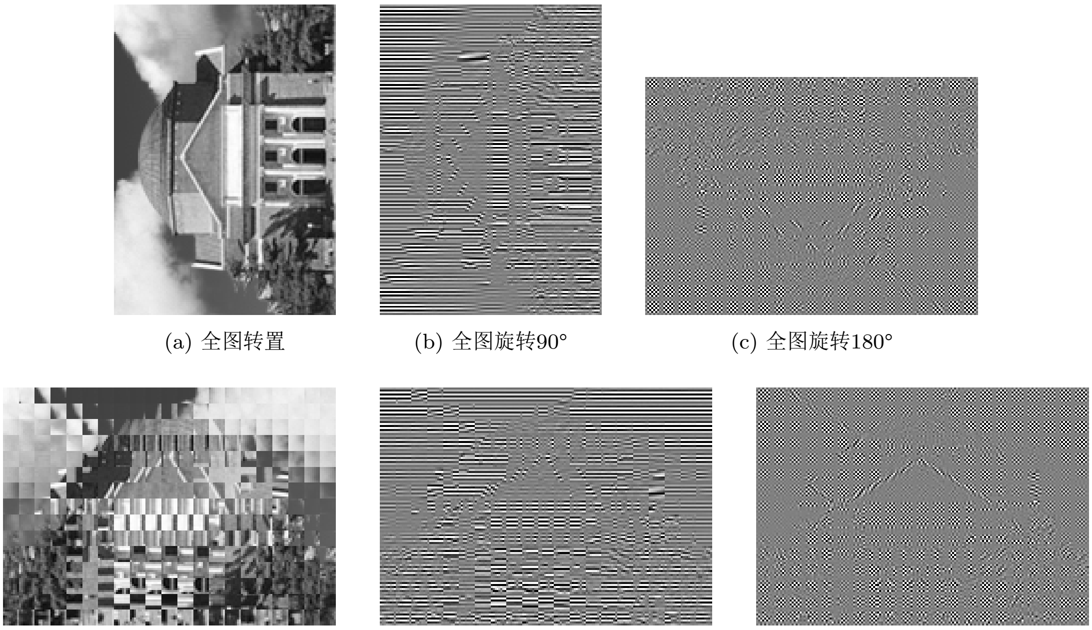
图 5: 二维 DCT 变换系数矩阵变化的影响 2

相对于不置零的图5c，右侧置零的图5b的水平变化看上去更加平滑，左侧置零的图 5a更加凸显了水平方向上的不同。这与前面的分析是一致的。

2.4 二维 DCT 系数矩阵性质 2

对整张图的系数和对每一个 8×8 的系数矩阵进行转置、旋转的操作，结果如图 6

除了全图的系数进行转置以外，都难以识别。考虑变换的过程，对系数矩阵 C 进行转置，只是将水平和竖直方向的系数做了交换，逆变换得到的结果也确实将水平和竖直方向的图案进行了“交换”，得到了将空域转置的效果。采用数学形式的描述即



(a) 全图转置　(b) 全图旋转90°　(c) 全图旋转180°

(d) 分块转置　(e) 分块旋转90°　(f) 分块旋转180°

图 6: 二维 DCT 变换系数矩阵变化的影响 3

$$
\begin{aligned}
P &= D^T C D \\
P^T &= (D^T C D)^T \\
&= D^T C^T D
\end{aligned}
$$

而对每个 8×8 的区域做转置，虽然同样符合此公式，但变换回空域后图案“不连续”，人难以识别该图片。

对于旋转90°，由章节 2.3中的分析，系数较大的直流与低频区域旋转至了竖直方向高频区域，故恢复的图像中由非常明显的横条纹，即竖直方向的剧烈变化。而旋转180°则将系数较大的直流与低频变换到了水平与竖直的共同高频区，造成了恢复后图像中大量的“棋盘状”黑白交替区域。

值得一提的是，全图系数旋转90°后，还能隐约看出与全图转置一致的结构信息。或许结构信息的分布经转置和旋转后差异较小，仍可以被识别。

2.5 差分编码系统的频率响应

该差分编码系统给出如下差分方程

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

对其进行 Z 变换，易得系统函数为

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

故采用`freqz([-1, 1], 1)`函数绘制系统的频率响应如图

显然，这是一个高通系统，对 DC 系数进行差分编码便是为了略去过多的低频分量。

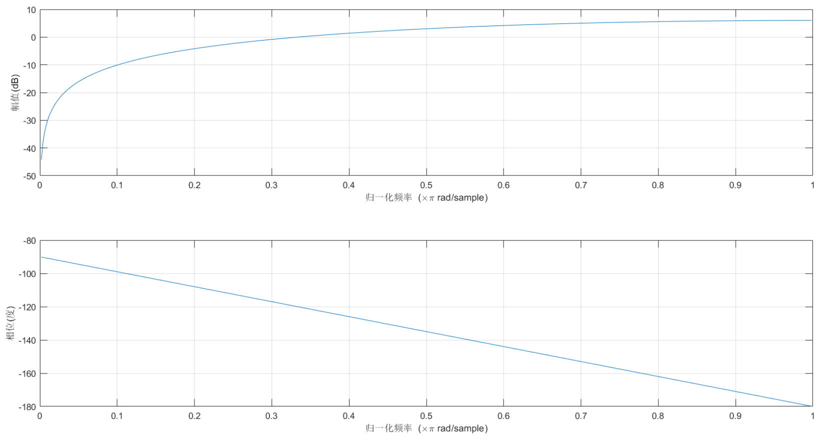


图 7: 差分编码系统的频率响应

2.6 DC 预测的误差

根据表格，易得二者间满足如下关系

$$\text{category} = \lceil \log_2(|\text{预测误差}| + 1) \rceil$$

对应到 Matlab 代码为

```
category = ceil(log2(abs(x) + 1))
```

需注意的是，上方代码给出的 category 从 0 开始，之后若需用作索引，应加 1 满足 Matlab 从 1 开始的规定。

2.7 Zig-Zag 扫描的实现

考虑 Zig-Zag 扫描在压缩编码中使用的场景永远是 8×8 的矩阵，可以简单的使用一个索引数组作为表，然后使用查表的方法进行扫描

$$\text{index} = \begin{bmatrix} 1 & 9 & 2 & \cdots & 56 & 64 \end{bmatrix}$$

考虑更一般的情形，可以对任意形状的矩阵使用循环，以 $O(n^2)$ 的复杂度进行一轮 Zig-Zag 扫描。函数实现类似于

```
function output = zigzag(input)
    [r, c] = size(input);
    output = zeros(1, r * c);
    i = 1;
    j = 1;
    cnt = 1;

    while ((i <= r) && (j <= c))
        output(cnt) = input(i, j);
```

```
        cnt = cnt + 1;
        if (mod(i + j, 2))
            % odd => down
            if (i == r)
                j = j + 1;
            elseif (j == 1)
                i = i + 1;
            else
                i = i + 1;
                j = j - 1;
            end
        else
            % even => up
            if (j == c)
                i = i + 1;
            elseif (i == 1)
                j = j + 1;
            else
                i = i - 1;
                j = j + 1;
            end
        end
    end
end
```

在 Stack Overflow 上面搜索后，找到了下方性能优异的实现方式[1]

```
function output = zigzag_amro(input)
    ind = reshape(1:numel(input), size(input));
    ind = fliplr( spdiags( fliplr(ind) ) );
    ind(:,1:2:end) = flipud( ind(:,1:2:end) );
    ind(ind==0) = [];
    output = input(ind);
end
```

```
function output = zigzag_luis(input)
    [r, c] = size(input);
    M = bsxfun(@plus, (1:r).', 0:c-1);
    M = M + bsxfun(@times, (1:r).'/(r+c), (-1).^M);
    [~, ind] = sort(M(:));
    output = input(ind).';
end
```

其中 Amro 的实现方式利用`spdiags`函数将矩阵元素放在对角线上，从而将 Zig-Zag 扫描变为了竖直方向的扫描，再通过翻转特定的列实现了扫描方向的改变，最后删除 0 元素得到

[1]这两段代码将 Stack Overflow 上的问题Matrix “Zigzag” Reordering中 Amro 和 Luis Mendo 的答案改为函数。

Zig-Zag 矩阵作为索引。而 Luis 的实现方式巧妙的建立一个矩阵，其元素与其位置相关，大小按 Zig-Zag 扫描顺序排列，从而得到了一个索引矩阵。

这两种方法虽然巧妙、高效，但是在本次实验的流程中，并不比查表有优势，故此后将使用查表的方式实现 Zig-Zag 扫描。此外，在解码过程中需要将 Zig-Zag 扫描得到的序列还原，即还需要一个逆过程的表。逆过程的表可以使用如下方式创建

```
    zigzag_inv_ind = zeros(1, 64)
    zigzag_inv_ind(zigzag_ind) = 1:64
```

即使用`zigzag_ind`作为索引矩阵，按照位置排列 1-64。

2.8 分块量化的实现

与前面一样，先构造一个处理函数，然后使用`blockproc`实现分块处理，代码如下

```
img = double(hall_gray) - 128;
process = @(block_struct) dct_process(block_struct.ata, ...
                                      QTAB, zigzag_ind);

DCT = blockproc(img, [8 8], process, 'PadPartialBlocks', true);
DCT = reshape(DCT', size(DCT, 2), 64, []);
DCT = permute(DCT, [2, 1, 3]);
DCT = reshape(DCT, 64, []);

function output = dct_process(input, QTAB, zigzag_ind)
    C = dct_2(input);
    C_quant = round(C ./ QTAB);
    C_zigzag = C_quant(zigzag_ind);
    output = C_zigzag';
end
```

这会得到一个 64 行的矩阵 DCT，每列即每块经 DCT 处理和量化后的系数。

注意使用了**`reshape`**和**`permute`**将`blockproc`得到的矩阵改为需要的格式。这种方法参考了 Stack Overflow 的问题Matlab reshape horizontal cat下 Luis Mendo 的回答。此方法巧妙地利用转置和**`permute`**，利用 Matlab 矩阵列优先的特性，将`blockproc`得到的 $64m \times n$ 的矩阵先划分为 m 层 $64 \times n$ 的矩阵，再拼成一个 $64 \times mn$ 的矩阵，完成了行优先的变形。

2.9 JPEG 编码

在章节 2.8的基础上，加入编码的部分。因为使用 Matlab 中`dec2bin`等函数（封装为 1 补码并转换为向量），且编码长度不定，故编码采用一维向量不断扩展的方式，可能造成性能较低。

对于 DC 编码，有如下代码呈现的 3 个部分：

```
DC_coef = coef(1, :);
DC_coef = [2 * DC_coef(1), DC_coef(1:end - 1)] - DC_coef;

dc_block_process = @(data) DC_process(data, DCTAB);
```

```
DC = arrayfun(dc_block_process, DC_coef, 'UniformOutput', false);
DC = cell2mat(DC);

function output = DC_process(input, DCTAB)
    category = ceil(log2(abs(input)+1)) + 1;
    huffman_code = DCTAB(category, ...
                     2:DCTAB(category, 1) + 1);
    dc_bin_mag = dec2bin_vec(input);
    output = [huffman_code, dc_bin_mag];
end
```

第一部分将量化 DC 系数进行差分，第二部分使用`arrayfun`对所有系数进行编码再使用`cell2mat`连接，第三部分为编码的函数。

使用的`dec2bin_vec`为基于`dec2bin`和`bitcmp`实现的 1 补码进制转换。

```
function bin_vec = dec2bin_vec(dec_int)
    if(dec_int > 0)
        bin_vec = split(dec2bin(dec_int), '');
        bin_vec = str2double(bin_vec(2:end - 1))';
    elseif(dec_int == 0)
        bin_vec = [];
    else
        bin_vec = dec2bin(bitcmp(abs(dec_int), 'uint16'));
        bin_vec = split(bin_vec(end ...
            - size(dec2bin(abs(dec_int)), ...
            2) + 1 : end), '');
        bin_vec = str2double(bin_vec(2:end - 1))';
    end
end
```

AC 编码做类似的处理，尚未找到方法替代循环实现其中的部分功能，且输出为不定长数组不能应用`blockproc`，可能效率较低。行数较多，故略去，详见随报告附源代码。

2.10 压缩比

原图为 120 × 168 的 uint8 矩阵，即有 20160 字节。经过 JPEG 压缩后为 2031 位 DC 码流与 23073 位 AC 码流，以二进制存储占据 25104 位，即 3138 字节，故压缩比约为 6.42。考虑到还需添加图片宽高等信息，并存储码表，实际文件大小可能会稍大一些。

2.11 JPEG 解码

解码主要分为 DC 解码、AC 解码、重组为系数矩阵、逆 DCT 变换四个部分。

对于 DC 编码使用的 Huffman 码表，观察发现以下三类情况：

1. 编码 00，幅度 0，序列 00
2. 编码三位，幅度序列长度为编码转十进制减 1

3. 编码多于 3 位，幅度序列长度为 0 索引加 2

故可以通过查找 0 出现的位置，判断编码类型，进而获取后面幅度的二进制表示。

```
cnt = 1;
while(cnt <= blocks)
    if(all(DC(1:2) == [0 0]))
        % category 0 mag 0
        DC(1:2) = [];
    else
        zero_ind = find(DC==0, 1);
        if(zero_ind < 4)
            huffman_code = DC(1:3);
            DC(1:3) = [];
            zero_ind = bin2dec( ...
                            strjoin( ...
                                string(huffman_code), '')) - 1;
        else
            DC(1:zero_ind) = [];
            zero_ind = zero_ind + 2;
        end
        mag = bin_vec2dec(DC(1:zero_ind));
        DC_coeff(cnt) = mag;
        DC(1:zero_ind) = [];
    end
    cnt = cnt + 1;
end
```

而对于 AC 编码，Huffman 码表较为复杂，故采用`ismember`和`find`结合的方式，查找序列的前 i 位是否存在于表中。

```
huffman(:) = 0;
for i = 1:16
    huffman(i) = AC(i);
    idx = find(ismember(ACTAB_code, huffman, 'rows'), 1);
...
```

对解码后的系数矩阵进行`reshape`和`permute`，然后再做逆变换，得到图像

```
coef = [DC_coeff; AC_coeff];

inv_zigzag = @(block_struct) ...
    idct2(reshape(block_struct.data(zigzag_inv), [8 8]) .* QTAB);
coef = reshape( ...
       permute( ...
       reshape(coef, 64, blocksize(2), []), ...
           [2 1 3]), ...
```

```
                blocksize(2), [])';

inv_img = blockproc(coef, [64, 1], inv_zigzag);
img = uint8(inv_img(1:height, 1:width) + 128);
```

对于解码得到的 64 × blocks 的系数矩阵，先使用`reshape`，转换为 64 × 横向块数 × 纵向块数的张量。再使用`permute`交换行列，以满足 Matlab 列优先的特性。最后使用`reshape`和转置变形为 64纵向块数 × 横向块数 的矩阵。

然后利用之前存储的逆 Zig-Zag 扫描序列作为索引，再使用`reshape`将每个 64×1 的系数向量恢复成 8×8 的方阵。再使用`blockproc`对所有块做反量化和逆 DCT 变换。最后裁切、加上 128，再改为整型存储。

对`hall_gray`进行编解码的结果如图8。

(a) 压缩 (b) 原始

图 8: `hall_gray`编解码结果

经计算，PSNR 为 34.1878dB，失真应不算大。放大图片进行观察，经过编解码的图片中多出了一些噪声，也隐隐有了分块的边界。比较明显的如天空与云交界处，多出了许多噪点。云的颜色变化处有较为明显的分块特征。这或许正是 JPEG 压缩的特征：由分成若干个 8×8 的小块再编码的必然结果。

2.12 量化步长的影响

编码得到 DC 码流 2408 位，AC 码流 34156 位，以二进制存储占 36564 位，即 4570.5 字节，压缩比约为 4.41。相比前面的结果，压缩比上升，应是由于量化步长减半导致 DC 与 AC 系数较大，进而编码变长。

计算得到 PSNR 为 37.1791dB，效果较前面的结果有所上升。实际观察噪声情况略有改善，但仍有分块边界。

2.13 雪花图

对雪花图进行编码，得到 DC 码流 1403 位，AC 码流 43539 位，计算得压缩比约为 3.65，PSNR 为 25.95dB。这表明雪花图接近随机数矩阵，有较多高频分量，量化造成的压缩失真较大。

考虑每个小块整个 JPEG 编解码的过程（使用 $.*$ 和 $./$ 表示逐元素乘除）：

图 9: hall_gray量化步长/2 编解码

$$\hat{P} = D^T(Q. * \mathrm{round}(DPD^T./Q))D$$

失真由四舍五入导致，其幅度在 0 到量化步长之间。进一步考虑对每一块 MSE 的计算过程：

$$\begin{aligned}
MSE &= \frac{1}{64}\sum_i^8\sum_j^8 (P_{ij} - \hat{P_{ij}})^2 \\
&= \frac{1}{64}\sum (D^T(Q. * (DPD^T./Q - \mathrm{round}(DPD^T./Q)))D).\hat{\ } 2
\end{aligned}$$

故由于四舍五入产生的误差将被 Q 放大到相应的量化步长，经过 DCT 逆变换后才得到 MSE。中间的误差可以视为一系列在 0 到 1 之间均匀分布的随机数。

对于 DCT 逆变换，由于其为正交变换，且 MSE 的主干为 Frobenius 范数，由下式易知不会对结果产生影响：

$$\begin{aligned}
||A||_F &= \sqrt{\sum_{i=1}^m\sum_{j=1}^n a_{ij}} = \sqrt{\mathrm{tr}(A^TA)} \\
||D^TAD||_F &= \sqrt{\mathrm{tr}((D^TAD)^T(D^TAD))} \\
&= \sqrt{\mathrm{tr}(D^TA^TDD^TAD)} = \sqrt{\mathrm{tr}((D^TA^T)(AD))} \\
&= \sqrt{\mathrm{tr}(ADD^TA^T)} = \sqrt{\mathrm{tr}(AA^T)}
\end{aligned}$$

故 MSE 可以进一步转化为：

$$MSE = \frac{1}{64}\sum (Q. * A).\hat{\ } 2$$

其中 A 为 U(0,1) 分布的随机数组成的矩阵。故可以计算其期望为：

$$E(x) = \int_0^{0.5} x^2 dx + \int_{0.5}^1 (1-x)^2 dx = \frac{1}{12}$$

所以 MSE 的理论结果为 $\sum(Q.\hat{}\ 2)/(12\times 64)=375.0365$，相应的 PSNR 为 22.39dB。与对雪花图进行压缩的结果相近，这表明雪花图确实接近随机矩阵。

注意到对雪花图进行编码时，AC 码流的长度较之前图像编码有了明显的上升，同时 PSNR 明显下降。推测是量化表中的系数经过设计，目的是对一般的图像中不太重要的高频信息进行滤波。而雪花图接近随机数注册的矩阵，量化表不适合对其进行处理，故最终压缩比相对较低。

3 信息隐藏

3.1 空域隐藏

使用`randi`生成一个随机 01 序列，并补 0 至长度与像素数一致。通过位移和加法，将像素最后一位替换为随机序列。经过 JPEG 编解码后与原有序列对比，记录正确率。代码如下

```
parfor i = 1:100
    secret = randi([0, 1], 1, randi(max_length));
    len = length(secret);
    secret_pad = [secret, zeros(1, max_length - len)];

    encode = reshape(bitshift(bitshift(img, -1), 1), 1, []);
    encode = reshape(encode + secret_pad, size(img));
    [DC, AC, height, width] = ...
        JPEG_encode(encode, QTAB, DCTAB, ACTAB, zigzag_ind);
    decode = ...
        JPEG_decode(DC, AC, height, width, QTAB, ACTAB, zigzag_inv_ind);

    secret_decode = reshape(decode, 1, []);
    secret_decode = mod(secret_decode(1:len), 2);

    correct = correct + sum(secret_decode == secret) / len;
end
```

最终得到正确率 0.4996，可以认为是生成了一段新的随机序列替换原有序列，完全破坏了隐藏信息。所以空域隐藏法抗编解码能力弱，不适于当前的互联网环境。

3.2 频域隐藏

根据说明，实现了以下三种频域隐藏方法：

a) `dct_embed_1`: 将二进制信息嵌入所有 dct 系数的最后一位

b) `dct_embed_2`: 将二进制信息嵌入经 Zig-Zag 扫描后的 dct 系数从`bias`开始的`len`个系数的最后一位

c) `dct_embed_3`: 将二进制信息变为 1，-1 序列，嵌入 dct 系数最后一个不为 0 数的后面一位

三种方法的信息容量分别约为：像素数、像素数/64 - 像素数、像素数/64。

测试时对于`dct_embed_2`采取将信息嵌入 2-9 位系数，另外两种方法则使用基本满容量的信息进行嵌入。得到结果如图10和表1。

表 1: 频域信息嵌入效果

	压缩比	PSNR	正确率
embed 1	2.8561	18.7162	1
embed 2	6.2485	33.362	1
embed 3	6.1914	31.8924	1

(a) embed 1

(b) embed 2

(c) embed 3

图 10: `hall_gray`频域信息嵌入结果

可以看到，方法 2 和 3 因为添加信息较少，且基本不会影响高频系数，故压缩比与未添加隐藏信息时相差无几。而方法一的质量和压缩比明显下降，并且人眼看来有明显的棋盘状图案，基本没有起到“隐藏”的意义。

由于是在频域进行嵌入，对于 JPEG 使用的 DCT 变换而言，这种嵌入可以保证信息无损。将信息尽可能嵌入中低频更有利于隐藏信息，即对于图像质量的影响尽可能低，对于压缩比的影响尽可能小。

4 人脸检测

4.1 训练标准特征

在生成颜色分布直方图时利用了所有像素，最后得到的频率分布与图像形状无关，故不需要调整图片形状，可以直接用于训练和检测。

使用如下函数得到一张图的颜色分布直方图，在计算所有训练数据的分布后求均值即可得到标准特征。

```
function color_histogram = color_histogram(img, L)
%COLOR_HISTOGRAM generate color histogram of img
% param img: image matrix
% param L: color bit length <= 8
% return color_histogram: vector, histogram

[~, ~, c] = size(img);
assert(c == 3, 'img should have 3 channels, i.g. RGB');
bins = 0 : 2 ^ (3 * L);

img = floor(double(img) / 2 ^ (8 - L));
pixel = img(:, :, 1) * 2 ^ (2 * L) + ...
```

```
        img(:, :, 2) * 2 ^L + img(:, :, 3);
pixel = reshape(pixel, 1, []);

color_histogram = histcounts(pixel, bins) * 3 / numel(img);
end
```

训练得到的标准特征如图11。可以看出，L 的大小影响的是色彩空间聚集区域的数量。如 L 为 3 时，将 RGB 为 0-31 的数量聚集到 [0, 0, 0]，而 L 为 4 时该点聚集的是 RGB 为 0-15 的像素数量，L 为 5 时聚集的是 RGB 为 0-7 的像素数量。L 会影响到颜色空间密度分布聚集后表现出来的“分辨率”。

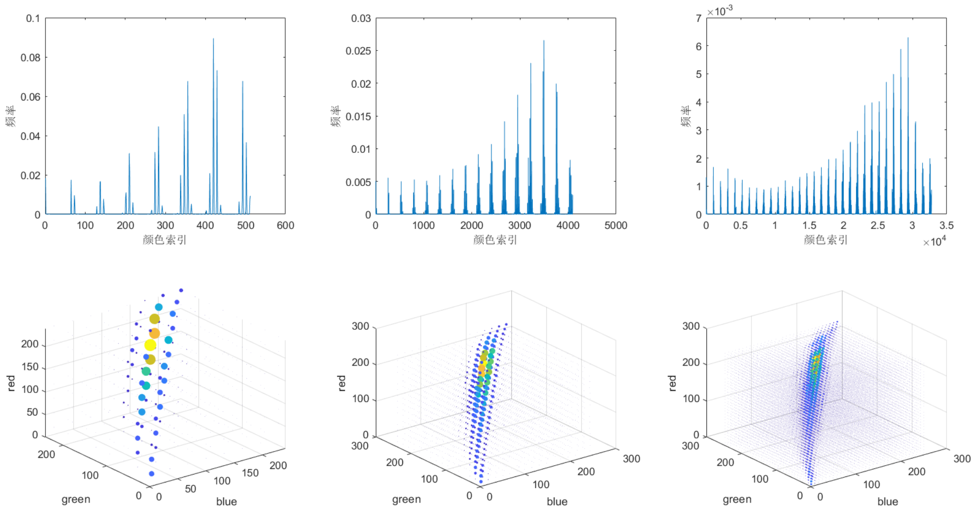


图 11: 颜色分布直方图与颜色空间密度分布

4.2 使用循环进行人脸检测

下面尝试使用循环从图片中检测出人脸，为方便复用，构建一个函数：

```
function candidates = ...
  detect_face(img, color_hist, L, blk_size, stride, threshold, k)
```

此函数接受一个图片矩阵（$H \times W \times C$），根据给定的颜色直方图标准 color_hist 和参数 L，对图片中的若干个区域（大小由 blk_size 给定，数量由步长 stride 和图片尺寸给出）进行“特征距离”的计算，再返回小于阈值的前 k 个矩形位置经去除相交后的结果。流程大致如图12所示，具体细节见随附代码。同时使用阈值和 topK 的目的是尽量限制去除相交区域时的候选矩形框数目，使其甚至可以少于 k。实际使用时可以将阈值设为 1，在全部的结果中选取 k 个，便可以规避因为阈值不合适得不到结果的问题。

初步实验结果表明，由于 L=5 时直方图细节较多，距离普遍较大，需要适当调大阈值以得到相似的结果，否则可能会得不到任何结果。选用了一张阿森纳球队的合影，适当调整参数尽量检测出所有人，见图13。

可以看到，阈值和 k 足够大时，L 并不太影响检测结果。由于函数中是先选出距离前 k 小的，再使用`rectint`和循环进行 $O(n^2)$ 的相交区域去除，故 L=3 的结果可能由于此原因相比

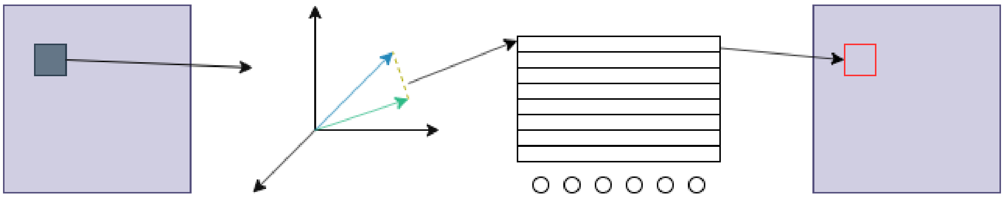


图 12: 人脸检测流程

(a) 阿森纳合影原图

(b) L=3 阈值 0.4 k=50

(c) L=4 阈值 0.4 k=100

(d) L=5 阈值 0.6 k=100

图 13: 人脸检测结果

$$\text{blk\_size} = [30, 25]; \text{stride} = [5, 5];$$

L=4 多遗失了两名球员的结果。由于训练使用的样本肤色较浅，当前参数均没有识别出右上角肤色较深的球员，表明此方法有相当大的局限性。

4.3 颜色分布直方图检测的稳定性

在上一节中，L=3 的效果已经不错，下面仍然使用 L=3 进行实验。使用以下函数对图像进行调整和检测：

```
img_rotate = imrotate(img, 90);
img_resize = imresize(img, [size(img, 1), size(img, 2) * 2]);
img_adjust = imadjust(img, [.1 .15 0; .85 .9 1]);

candidates_rotate = ...
    detect_face(img_rotate, color_hist.vec_3, ...
        3, [25, 30], stride, 0.4, 100);
candidates_resize = ...
    detect_face(img_resize, color_hist.vec_3, ...
        3, [30, 50], stride, 0.4, 100);
```

```
candidates_adjust = ...
    detect_face(img_adjust, color_hist.vec_3, ...
        3, [30, 25], stride, 0.4, 100);
```

(a) 拉伸

(b) 旋转

(c) 颜色调整

图 14: 人脸检测结果稳定性实验

从颜色分布直方图检测法的原理来看，旋转必然不会影响检测结果，而拉伸可能会由于插值方式的不同带来些微的影响，颜色调整则必然会带来影响。在图14中可以看到，旋转后的结果较上一节略好，这应是调大了 k 的结果。与之相比，拉伸少检测出一个人，调整颜色则多检测出一个人。这可能就是颜色的些微变化，让球员的脸部颜色直方图在所有候选中排位往前（或往后）移动，导致最后检测结果出现变化。

4.4 人脸样本训练标准的选取

从前面的实验结果来看，这种基于颜色分布直方图的方法虽然有受旋转、拉伸影响小的优点，但是非常受训练样本的影响，导致本实验中没有检测出非裔球员。虽然可能颜色分布与标准样本距离小于阈值，但仍会因为过于接近阈值，而在选取 topK 的步骤中被舍弃。如需进行更为准确的人脸检测，应当考虑针对不同人种训练不同的人脸标准，并在应用时将检测区域与所有样本进行比对，使得不同人种的脸部都能被识别出来。但随着标准的增多，误判的概率会有所上升，应考虑进一步限制阈值以保证准确性。

然而，由于颜色分布直方图的原理局限于颜色，不能捕获边缘特征。实际应用于人脸检测时应考虑引入边缘特征，与颜色分布一起计算一个“置信度”，再根据这个“置信度”决定是否是人脸。

5 后记