

# 图像处理

## Matlab 大作业报告

陈子熠

2024 年 8 月 3 日

## 目录

<b>1 基础知识</b>	<b>2</b>
1.1 Image 工具箱 . . . . .	2
1.2 Image file I/O 作图 . . . . .	2
<b>2 图像压缩编码</b>	<b>4</b>
2.1 DCT 域预处理 . . . . .	4
2.2 二维 DCT 实现 . . . . .	4
2.3 DCT 系数置零 . . . . .	6
2.4 DCT 系数矩阵变换 . . . . .	7
2.5 差分编码系统 . . . . .	7
2.6 DC 预测误差与 Category 的关系 . . . . .	8
2.7 Zig-Zag 扫描算法 . . . . .	8
2.8 分块、DCT 与量化 . . . . .	11
2.9 JPEG 编码 . . . . .	11
2.10 计算压缩比 . . . . .	12
2.11 JPEG 解码 . . . . .	13
2.12 量化步长减小一半 . . . . .	16
2.13 雪花图像 JPEG 编解码 . . . . .	17
<b>3 信息隐藏</b>	<b>17</b>
3.1 空域隐藏 . . . . .	17
3.2 DCT 域隐藏 . . . . .	18
3.3 新的信息隐藏方法 . . . . .	20
<b>4 人脸检测</b>	<b>21</b>
4.1 训练人脸标准 . . . . .	21
4.2 人脸检测 . . . . .	21
4.3 图像变换后的人脸检测 . . . . .	24
4.4 重新选择人脸样本训练标准 . . . . .	25
<b>5 实验总结</b>	<b>25</b>

# 1 基础知识

本实验所有测试程序入口均为 `src/main.m`, 并标有相应的编号, 如 1.1、2.2 等。实现的函数位于 `src` 文件夹下其它文件。

## 1.1 Image 工具箱

在命令窗口输入 `help images` 即可查看该工具箱内的所有函数, 如图1。可以了解到该工具箱包含了图像处理的基本操作, 如读取、显示、保存图像, 以及图像的基本处理, 如缩放、旋转、裁剪、滤波等。

在之后的实验中, 将会使用该工具箱内的部分函数, 如 `imread`、`imshow`、`imwrite`、`imresize` 等。

```
>> help images;
Image Processing Toolbox
Version 23.2 (R2023b) 01-Aug-2023

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
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 image.
randomAffine2d              - Construct randomized 2-D affine transformation.
randomAffine3d              - Construct randomized 3-D affine transformation
```

图 1: `help images` 查看 Image 工具箱内的函数

## 1.2 Image file I/O 作图

首先, 通过 `load('./resource/hall.mat')`; 读取图像数据。

值得注意的是, 图像的数据类型是 `uint8`, 即无符号 8 位整数, 取值范围为  $[0, 255]$ 。但是在进行数据处理时, 通常需要将其转换为 `double` 类型, 即双精度浮点数, 否则可能导致不确定的结果; 在显示或保存图像时, 则需要将其转换为 `uint8` 类型。这一点在之后的压缩编码等实验中尤为重要。

以图像的中心点为圆心画圆, 思路是以图像的中心点为圆心, 半径约为图像宽度的一半处画一个圆环。只需要将距中心点距离在外环半径和内环半径之间的像素点的 `rgb` 值设为 `[255, 0, 0]` 即可。关键代码如下:

```
1 [rows, cols, ~] = size(img);
2
3 center = [cols / 2, rows / 2];
4 radius_outer = min(rows, cols) / 2; % outer radius
5 radius_inner = radius_outer - linewidth; % inner radius
6
```

```

7 [x, y] = meshgrid(1:cols, 1:rows); % generate meshgrid
8 mask = (x - center(1)).^2 + (y - center(2)).^2 <= radius_outer^2 & (x
    ↪ - center(1)).^2 + (y - center(2)).^2 >= radius_inner^2;
9 circle = double(img);
10 circle(:,:,:,1) = rgb(1) * mask + circle(:,:,:,1) .* ~mask; % set rgb
    ↪ value
11 circle(:,:,:,2) = rgb(2) * mask + circle(:,:,:,2) .* ~mask;
12 circle(:,:,:,3) = rgb(3) * mask + circle(:,:,:,3) .* ~mask;
13
14 imshow(uint8(circle));

```

其中, `mask` 为一个逻辑矩阵, 表示距中心点距离在外环半径和内环半径之间的像素点。`rgb` 为设定的 `rgb` 值, `linewidth` 为环的宽度。该方法封装在 `plot_circle` 中。

绘制位图如图 2a。

用类似的方法, 可以绘制棋盘。思路是将每个像素的横纵坐标分别整除棋盘格子的边长, 相加后对 2 取模, 即可得到棋盘格子的颜色。关键代码如下:

```

1 [rows, cols, channels] = size(img);
2
3 [x, y] = meshgrid(1:cols, 1:rows); % generate meshgrid
4 checkerboard = mod(floor((x - 0.5) / length) + floor((y - 0.5) /
    ↪ length), 2); % 0.5 for better visualization
5 mask = repmat(checkerboard, [1, 1, channels]) = 0; % set rgb to
    ↪ black
6 chess = double(img);
7 chess(mask) = 0;
8
9 imshow(uint8(chess));

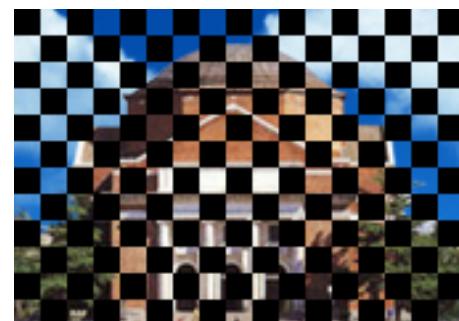
```

其中, `checkerboard` 为一个逻辑矩阵, 表示棋盘格子的颜色。`length` 为棋盘格子的边长。该方法封装在 `plot_chess` 中。

绘制位图如图 2b。



(a) 以图像的中心点为圆心画圆



(b) 绘制棋盘

图 2: 图像作图

## 2 图像压缩编码

### 2.1 DCT 域预处理

注意到, DCT 同 FFT 类似, 满足线性性质, 即  $DCT(a \cdot x + b \cdot y) = a \cdot DCT(x) + b \cdot DCT(y)$ 。因此, 先将图像灰度值减去 128, 再进行 DCT 变换, 等价于先进行 DCT 变换, 再将 DCT 变换后的系数减去全为 128 的矩阵的 DCT 变换。由于全为 128 的矩阵只包含直流分量, 因此其 DCT 变换后只有左上角的系数非零, 故只需将左上角的系数减去全为 128 的矩阵的 DCT 变换的直流分量。同时, 在二维 DCT 变换公式中, 令  $i = 0, j = 0$ , 可得:

$$\begin{aligned} F(0, 0) &= \frac{1}{\sqrt{MN}} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \cos\left(\frac{(2x+1)0\pi}{2M}\right) \cos\left(\frac{(2y+1)0\pi}{2N}\right) \\ &= \frac{1}{\sqrt{MN}} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \cos(0) \cos(0) \\ &= \frac{1}{\sqrt{MN}} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \end{aligned}$$

故 DCT 直流分量可以通过计算原图像的均值, 并乘以  $\frac{1}{\sqrt{MN}}$  得到。在本题中, 即  $128 * 8 = 1024$ 。

因此 DCT 在变换域的预处理可以用以下代码实现并验证:

```
1 patch00 = double(hall_gray(1:8, 1:8));
2 c_patch00_trans = dct2(patch00);
3 c_patch00_trans(1, 1) = c_patch00_trans(1, 1) - 128 * 8;
4 c_patch00_ori = dct2(patch00 - 128);
5 disp("SSE: " + sum((c_patch00_trans - c_patch00_ori).^2, 'all'));
```

得到 SSE 为  $3.5715e-25$ , 即两种方法得到的 DCT 系数完全一致, 验证了此变换域预处理方法的正确性。

### 2.2 二维 DCT 实现

首先计算变换矩阵 D, 代码如下:

```
1 function D = dct_D(N)
2 % Create the 2D DCT matrix
3 % N [int]: the size of the matrix
4 % return D [2D double]: the DCT matrix
5 D = [1 : N - 1]' * [1: 2: 2 * N - 1];
6 D = cos(D * pi / (2 * N));
7 D = [repmat(sqrt(0.5), [1, N]); D];
8 D = D * sqrt(2 / N);
9 end
```

通过矩阵运算, 避免了显式的循环, 提高了运算效率。

接着实现二维 DCT 变换，代码如下：

```
1 function C = dct_2(P, s)
2 % Perform 2D DCT on the input matrix
3 % P [2D double]: the input matrix
4 % s [1D int][optional]: the size of the matrix
5 % return C [2D double]: the DCT matrix
6 [rows, cols] = size(P);
7
8 if nargin < 2
9     if rows == cols
10        D = dct_D(rows);
11        C = D * P * D';
12    else
13        D = dct_D(rows);
14        D_T = dct_D(cols)';
15        C = D * P * D_T;
16    end
17 else
18    P_padding = zeros(s);
19    min_rows = min(rows, s(1));
20    min_cols = min(cols, s(2));
21    P_padding(1:min_rows, 1:min_cols) = P(1:min_rows,
22    ↳ 1:min_cols);
23    if s(1) == s(2)
24        D = dct_D(s(1));
25        C = D * P_padding * D';
26    else
27        D = dct_D(s(1));
28        D_T = dct_D(s(2))';
29        C = D * P_padding * D_T;
30    end
31 end
```

注意到，MATLAB 库提供的 `dct2` 函数可以处理任意形状的矩阵（并不局限于方阵），并且可以指定变换域的大小，视情况对输入矩阵进行 padding 或 truncation。因此，我实现的 `dct_2` 函数也具有这两个功能，且参数数目可变，输入输出格式与 `dct2` 完全一致。

特别的，我实现的 `dct_2` 函数对方阵进行了优化，避免了两次计算 D 矩阵，提高了运算效率。

下面，我分别对方阵和非方阵的矩阵进行了功能验证，验证代码如下：

```
1 patch00 = double(hall_gray(1:8, 1:8));
2 c_patch00 = dct2(patch00);
3 c_patch00_ = dct_2(patch00);
4 disp("SSE: " + sum((c_patch00 - c_patch00_).^2, 'all'));
5 c_patch00 = dct2(patch00, [6, 10]);
```

```

6 c_patch00_ = dct_2(patch00, [6, 10]);
7 disp("SSE: " + sum((c_patch00 - c_patch00_).^2, 'all'));

```

SSE 截图如图 3，验证了 `dct_2` 函数的正确性。

```

SSE: 1.8696e-24
SSE: 2.0446e-24

```

图 3: DCT 正确性验证

我还对 `dct_2` 函数的运行时间与原 MATLAB 库提供的 `dct2` 函数进行了比较，两者各运行 100000 次，结果如图 4。

```

Time of matlab dct2, square: 0.97318
Time of my dct2, square: 0.55977
Time of matlab dct2, rectangle: 1.0083
Time of my dct2, rectangle: 0.95861

```

图 4: DCT 运行时间比较

可见，我实现的 `dct_2` 函数比 MATLAB 库提供的 `dct2` 函数更高效，尤其在方阵的情况下，速度提高接近 2 倍。

### 2.3 DCT 系数置零

理论上，DCT 变换得到的系数矩阵左上角代表直流分量，左下代表纵向分量变化的高频分量，右上代表横向变化的高频分量，右下代表横向和纵向变化的高频分量。因此，若将右侧 4 列置零，主要会损失横向变化的高频分量，横向亮度变化会变模糊。但由于图像中高频分量的能量较低，因此这种损失对图像的影响较小，视觉效果不会有太大变化。

若将左侧 4 列置零，主要会损失直流分量和纵向分量变化的高频分量。一方面，纵向亮度变化会变模糊。另一方面，图像亮度会趋于相同。由于处理前直流分量减去了 128，恢复时要加上 128，故亮度会趋于 128，呈现灰色。由于人眼对低频分量更敏感，因此这种损失对图像的影响较大，视觉效果上看图像严重受损。

为了验证猜想，我将图片按横纵 8\*8 分块，对每个块进行 DCT 变换，并将对应列置零。效果如图 5。



图 5: 系数置零整体效果

可见，将左侧 4 列置零后，图像亮度整体受损严重，成为一片灰色。唯一保留的是横向变化的分量。由于这部分能量较低，呈现零星的纵向纹路。而将右侧 4 列置零后，图像变

化不明显，与理论分析一致。

局部效果如图 6。可以看到，将左侧 4 列置零后，直流分量损失严重，局部变成灰色块。而将右侧 4 列置零后，直接和纵向分量几乎不受影响，但横向变得模糊，呈现条带状。

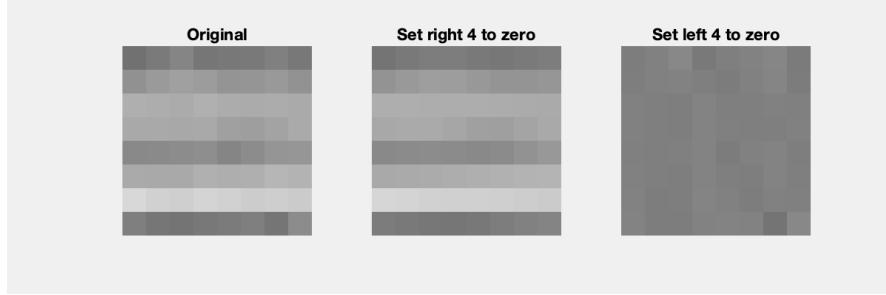


图 6：系数置零局部效果

## 2.4 DCT 系数矩阵变换

转置：横向分量与纵向分量交换，直流分量和横纵高频分量不变。注意到：

$$C^T = (DPD^T)^T = DP^T D^T = \text{DCT}(P^T)$$

因此，还原出来的图像是原图形的转置，即图像绕左上-右下对角线翻转。

旋转 90 度：左上变左下，低频分量变纵向高频分量；右上变左上，横向高频分量变低频分量。由于原图像低频能量较高，高频能量较低，因此旋转 90 度后，图像的纵向变化会很剧烈，但整体亮度趋于灰色，严重受损。

旋转 180 度：左上与右下互换，低频分量与横纵高频分量互换，横纵高频分量能量大大增加，直流受损，图像亮度应呈棋盘状剧烈变化。

局部效果如图 7。

可见，转置后恢复的图像确实是原图像的转置，旋转 90 度后图像的纵向变化剧烈，旋转 180 度后图像呈棋盘状，与理论一致。

整体效果如图 8。

由于图像被分成  $8 \times 8$  的块，因此转置后图像是大量局部转置按原顺序拼接而成的结果，凌乱中还看得出原图像的大致轮廓，颇有一种抽象油画的感觉，也与受损老照片有几分形似。而旋转 90 度后，则几乎只能看出纵向的剧烈变化，整体严重受损。旋转 180 度后，图像像多个棋盘拼接而成，形态类似纱布。这些现象均与理论分析一致。

## 2.5 差分编码系统

对于直流分量，通过差分编码后输出为零；对于剧烈变化的高频分量，通过差分编码后能量可能会增加，例如  $1, -1, 1, -1, \dots$  将成为  $2, -2, 2, -2, \dots$ 。因此猜想差分编码系统是高通滤波器。

绘制频率响应如图 9。可见确实是一个高通滤波器。

DC 系数先进行差分编码再进行熵编码，说明 DC 系数的低频分量更多。这样做的好处是，通过差分编码后能有效降低能量，编码时可以节省比特数，在保持信息不丢失的前提下提高压缩率。

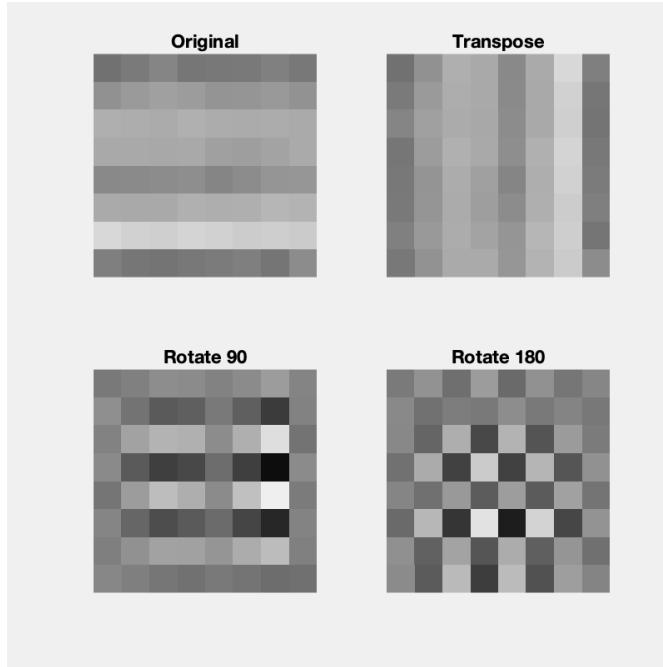


图 7: DCT 系数矩阵变换局部效果

## 2.6 DC 预测误差与 Category 的关系

记 DC 预测误差为  $\delta$ , 类别为  $C$ , 容易发现,  $C = \min(\text{ceil}(\log_2(|\delta| + 1)), 11)$ 。这里不用 floor 是为了避免  $C = 0$  的情况产生算术溢出, 增加 min 是为了处理预测误差超过预定上下界的情况, 提高鲁棒性。

## 2.7 Zig-Zag 扫描算法

一种通用的方法是按 Zig-Zag 扫描顺序将矩阵展开为向量, 封装如下:

```

1 function o = zig_zag(i)
2     [N, M] = size(i);
3     o = zeros(1, N * M);
4     o(1) = i(1, 1);
5     row = 1;
6     col = 1;
7     flag = 0; % 1 means row increases and column decreases
8
9     for idx = 2 : N * M
10        if col == N && flag == 0
11            row = row + 1; flag = 1;
12        elseif row == N && flag == 1
13            col = col + 1; flag = 0;
14        elseif row == 1 && flag == 0
15            col = col + 1; flag = 1;
16        elseif col == 1 && flag == 1
17            row = row + 1; flag = 0;

```

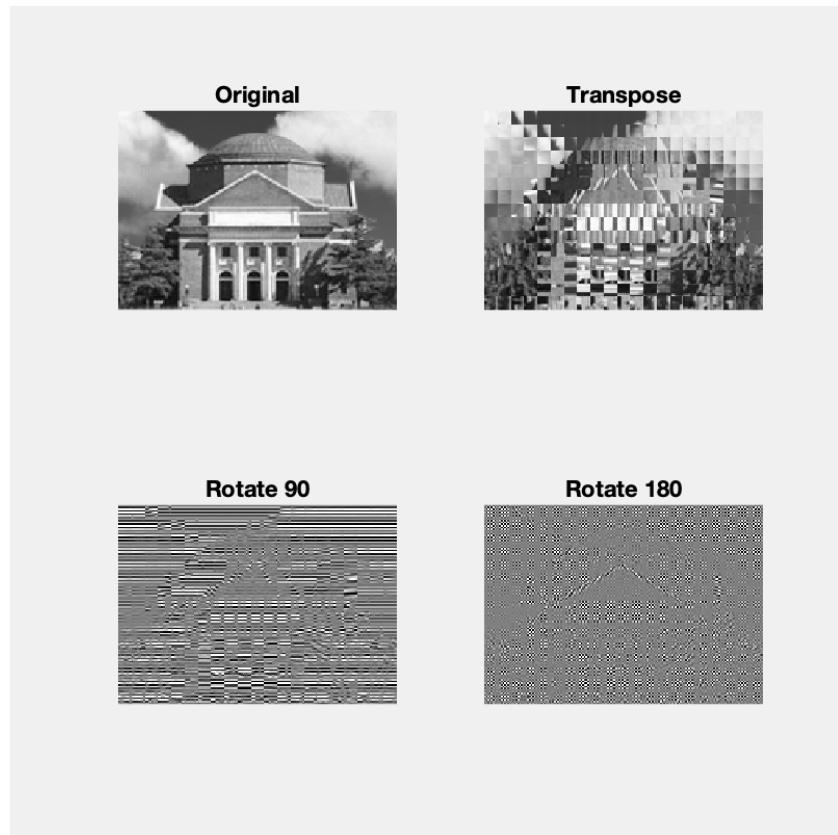


图 8: DCT 系数矩阵变换整体效果

```

18     elseif flag == 1
19         row = row + 1; col = col - 1;
20     else
21         row = row - 1; col = col + 1;
22     end
23     o(idx) = i(row, col);
24 end
25 end

```

上述算法具备很好的通用性，可以处理任意形状的矩阵，但需要二重循环，效率较低。本题中，由于矩阵是固定的 8\*8 的方阵，可以利用查表法，配合矩阵索引，对输入输出索引建立双射关系，提高效率，代码如下：

```

1 function o = zig_zag_8_8(i)
2 % Perform zig-zag scan on the input matrix, assuming the input is
3 % 8x8
4 % i [2D double]: the input matrix
5 % return o [1D double]: the zig-zag scanned matrix
6 idx_matrix =
7     1, 9, 2, 3, 10, 17, 25, 18, ...
8     11, 4, 5, 12, 19, 26, 33, 41, ...
9     34, 27, 20, 13, 6, 7, 14, 21, ...

```

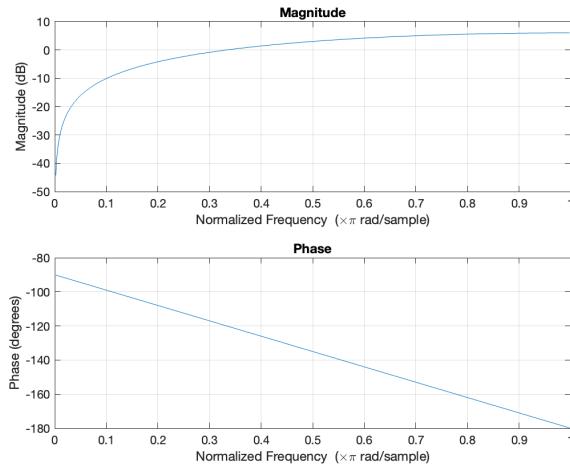


图 9: 差分编码系统频率响应

```

9      28, 35, 42, 49, 57, 50, 43, 36, ...
10     29, 22, 15, 8, 16, 23, 30, 37, ...
11     44, 51, 58, 59, 52, 45, 38, 31, ...
12     24, 32, 39, 46, 53, 60, 61, 54, ...
13     47, 40, 48, 55, 62, 63, 56, 64, ...
14 ];
15 o = reshape(i, [1, 64]);
16 o = o(idx_matrix);
17 end

```

下面对 Zig-Zag 扫描算法进行了功能验证，分别选取 8\*8 和 9\*9 的矩阵对两种算法进行了验证，测试矩阵如下：

```

1 input_8_8 = [
2   1, 2, 6, 7, 15, 16, 28, 29;
3   3, 5, 8, 14, 17, 27, 30, 43;
4   4, 9, 13, 18, 26, 31, 42, 44;
5   10, 12, 19, 25, 32, 41, 45, 54;
6   11, 20, 24, 33, 40, 46, 53, 55;
7   21, 23, 34, 39, 47, 52, 56, 61;
8   22, 35, 38, 48, 51, 57, 60, 62;
9   36, 37, 49, 50, 58, 59, 63, 64;
10 ];
11 input_9_9 = [
12   1, 2, 6, 7, 15, 16, 28, 29, 45;
13   3, 5, 8, 14, 17, 27, 30, 44, 46;
14   4, 9, 13, 18, 26, 31, 43, 47, 60;
15   10, 12, 19, 25, 32, 42, 48, 59, 61;
16   11, 20, 24, 33, 41, 49, 58, 62, 71;
17   21, 23, 34, 40, 50, 57, 63, 70, 72;
18   22, 35, 39, 51, 56, 64, 69, 73, 78;

```

```

19     36, 38, 52, 55, 65, 68, 74, 77, 79;
20     37, 53, 54, 66, 67, 75, 76, 80, 81;
21 ];

```

预期输出为按 1, 2, 3, ... 的顺序排列的向量。实际输出符合预期 (8\*8 的如图 10, 受限于篇幅, 9\*9 的不再展示, 感兴趣的可以自行验证)。

```

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

```

图 10: Zig-Zag 扫描算法功能验证

## 2.8 分块、DCT 与量化

可以利用 MATLAB 的矩阵运算功能和 `blockproc` 方法高效实现, 关键代码如下:

```

1 function o = blockproc_8_8(i, func, varargin)
2     o = blockproc(i, [8, 8], @block) func(block.data, varargin{:}),
3     ↵     'PadPartialBlocks', true);
4 end
5
6 [rows, cols] = size(i);
7 C = blockproc_8_8(double(i) - 128, @dct2); % DCT
8 Q = blockproc_8_8(C, @(x) round(x ./ QTAB)); % Quantization
9 Z = blockproc_8_8(Q, @(x) zig_zag_8_8(x)); % Zig-zag scan
10 Z = reshape(z.', [64, ceil(rows / 8) * ceil(cols / 8)]); % Reshape

```

其中, `blockproc_8_8` 函数是对 `blockproc` 针对 8\*8 分块的进一步封装, 将在之后被多次调用, 提高了代码的简洁性。

## 2.9 JPEG 编码

首先, 定义十进制数转二进制数的函数, 其中负数用 1 补码表示:

```

1 function o = dec2bin_arr(i)
2     if i ~= 0
3         o = dec2bin(abs(i)) - '0';
4         if i < 0
5             o = ~o;
6         end
7     else
8         o = [];
9     end

```

```
10 end
```

量化后的系数矩阵 Z 已由 2.8 中的代码得到，接下来对其直流分量进行差分编码和霍夫曼编码并转化为二进制码流，关键代码如下：

```
1 DC = Z(1, :);
2 DC_diff = [DC(1), DC(1 : end - 1) - DC(2 : end)];
3 DC_cata = min(ceil(log2(abs(DC_diff) + 1)), 11);
4 get_code_and_mag_dc = @(cata, diff) [DCTAB(cata + 1, 2 : DCTAB(cata +
    ↪ 1, 1) + 1), dec2bin_arr(diff)];
5 DC_stream = cell2mat(arrayfun(@(cata, diff) get_code_and_mag_dc(cata,
    ↪ diff), DC_cata, DC_diff, 'UniformOutput', false));
```

接着对交流分量进行游程编码和霍夫曼编码，关键代码如下：

```
1 AC = Z(2 : end, :);
2 AC_size = min(ceil(log2(abs(AC) + 1)), 10);
3 AC_stream = [];
4 EOB = [1 0 1 0]; % End of block
5 ZRL = [1 1 1 1 1 1 1 1 0 0 1]; % Zero run length
6 get_code_and_mag_ac = @(run, size, amp) [ACTAB(run * 10 + size, 4 :
    ↪ ACTAB(run * 10 + size, 3) + 3), dec2bin_arr(amp)];
7 for idx = 1 : size(AC, 2)
8     num_zero = 0;
9     for jdx = 1 : size(AC, 1)
10         if AC(jdx, idx) == 0
11             num_zero = num_zero + 1;
12         else
13             if num_zero > 15
14                 AC_stream = [AC_stream, repmat(ZRL, 1, floor(num_zero
                    ↪ / 16))];
15                 num_zero = mod(num_zero, 16);
16             end
17             AC_stream = [AC_stream, get_code_and_mag_ac(num_zero,
                    ↪ AC_size(jdx, idx), AC(jdx, idx))];
18             num_zero = 0;
19         end
20     end
21     AC_stream = [AC_stream, EOB];
22 end
```

最后保存 DC 码流、AC 码流、图像大小。完整代码位于 `src/jpeg_encode.m`。

## 2.10 计算压缩比

将原图像与码流长度单位统一为 bit。由于原图像数据类型为 uint8，故需将面积乘以 8。代码如下：

```
1 disp("Compression ratio: " + (8 * rows * cols) / (length(DC_stream) +
→ length(AC_stream)));
```

得到压缩比为 6.4247

## 2.11 JPEG 解码

同样定义二进制数转十进制数的函数，其中负数用 1 补码表示：

```
1 function o = bin_arr2dec(i)
2     if isempty(i)
3         o = 0;
4     else
5         if i(1) == 0
6             i = ~i;
7             o = -bin2dec(num2str(i));
8         else
9             o = bin2dec(num2str(i));
10        end
11    end
12 end
```

为对霍夫曼编码进行解码，构建霍夫曼树。霍夫曼树为一结构体，包含左右子树及对应 symbol。构造过程如下：

```
1 function tree = addSymbolToTree(tree, code, symbol)
2     if isempty(code)
3         tree.symbol = symbol;
4         return;
5     end
6     if code(1) == 0
7         if isempty(fieldnames(tree.left))
8             tree.left = struct('symbol', NaN, 'left', struct(),
9             → 'right', struct());
9         end
10        tree.left = addSymbolToTree(tree.left, code(2 : end),
11            → symbol);
11    else
12        if isempty(fieldnames(tree.right))
13            tree.right = struct('symbol', NaN, 'left', struct(),
14            → 'right', struct());
14        end
15        tree.right = addSymbolToTree(tree.right, code(2 : end),
16            → symbol);
16    end
17 end
```

对 DC 霍夫曼编码构造霍夫曼树：

```
1 DC_huffmanTree = struct('symbol', NaN, 'left', struct(), 'right',
2     ↪ struct());
3 for idx = 1:size(DCTAB, 1)
4     DC_huffmanTree = addSymbolToTree(DC_huffmanTree, DCTAB(idx, 2 :
5         ↪ DCTAB(idx, 1) + 1), idx - 1);
6 end
```

对 DC 码流进行解码并反差分，结果保存在矩阵 Z 的第一行。关键代码如下：

```
1 Z = zeros(64, ceil(rows / 8) * ceil(cols / 8));
2
3 DC = [];
4 DC_huffman_pointer = DC_huffmanTree;
5 idx = 1;
6 while idx <= size(DC_stream, 2)
7     DC_code = DC_stream(idx);
8     if DC_code == 0
9         DC_huffman_pointer = DC_huffman_pointer.left;
10    else
11        DC_huffman_pointer = DC_huffman_pointer.right;
12    end
13    idx = idx + 1;
14    if ~isnan(DC_huffman_pointer.symbol)
15        DC_cata = DC_huffman_pointer.symbol;
16        DC = [DC, bin_arr2dec(DC_stream(idx : idx + DC_cata - 1))];
17        DC_huffman_pointer = DC_huffmanTree;
18        idx = idx + DC_cata;
19    end
20 end
21
22 for idx = 2 : size(DC, 2)
23     DC(idx) = DC(idx - 1) - DC(idx);
24 end
25 Z(1, :) = DC;
```

对 AC 霍夫曼编码构造霍夫曼树，与 DC 类似，不再赘述。

对 AC 码流进行解码，结果保存在矩阵 Z 的第二行至最后，格式与编码前一致。关键代码如下：

```
1 AC = [];
2 AC_huffman_pointer = AC_huffmanTree;
3 idx = 1;
4 patch_idx = 1;
5 while idx <= size(AC_stream, 2)
6     AC_code = AC_stream(idx);
```

```

7 if AC_code == 0
8     AC_huffman_pointer = AC_huffman_pointer.left;
9 else
10    AC_huffman_pointer = AC_huffman_pointer.right;
11 end
12 idx = idx + 1;
13 if ~isnan(AC_huffman_pointer.symbol)
14     AC_run = AC_huffman_pointer.symbol(1);
15     AC_size = AC_huffman_pointer.symbol(2);
16     if AC_run == 0 && AC_size == 0
17         Z(2 : size(AC, 1) + 1, patch_idx) = AC;
18         AC = [];
19         patch_idx = patch_idx + 1;
20     else
21         AC = [AC; zeros(AC_run, 1); bin_arr2dec(AC_stream(idx :
22             ↳ idx + AC_size - 1))];
23         idx = idx + AC_size;
24     end
25     AC_huffman_pointer = AC_huffmanTree;
26 end

```

最后，对 DCT 系数矩阵进行逆量化和逆 DCT 变换，得到原图像：

```

1 Z = reshape(Z, [64 * ceil(cols / 8), ceil(rows / 8)]).';
2 Q = blockproc(Z, [1, 64], @block) zig_zig_8_8(block.data));
3 C = blockproc_8_8(Q, @x x .* QTAB);
4 o = blockproc_8_8(C, @idct2) + 128;
5 o = uint8(o(1:rows, 1:cols));

```

其中，`zig_zig_8_8` 函数是 `zig_zig_8_8` 函数的逆操作，将 Zig-Zag 扫描的结果还原为 8\*8 的矩阵，同样采用查表法，不再赘述。完整代码位于 `src/jpeg_decode.m`。

对编码、解码进行测试，结果如图 11。可以看到，解码后的图像与原图像视觉效果几乎完全一致，说明 JPEG 编码解码过程正确。

仔细观察图 11，可以发现，解码后的图像存在较明显的块状效应（特别是在白云附近），这是因为 JPEG 编码过程中对图像进行了分块处理，导致边界处的块与块之间存在不连续性。同时，图像的细节部分也受到了一定程度的模糊，例如亮度变化剧烈的地方，这是量化的结果。

下面用 PSNR 评价解码后的图像与原图像的相似度：

```

1 function PSNR = cal_PSNR(ori, img)
2     MSE = sum((double(ori) - double(img)).^2, 'all') / numel(ori);
3     PSNR = 10 * log10(255^2 / MSE);
4 end

```

得到 PSNR 为 31.1874 dB，说明解码后的图像与原图像的相似度较高。

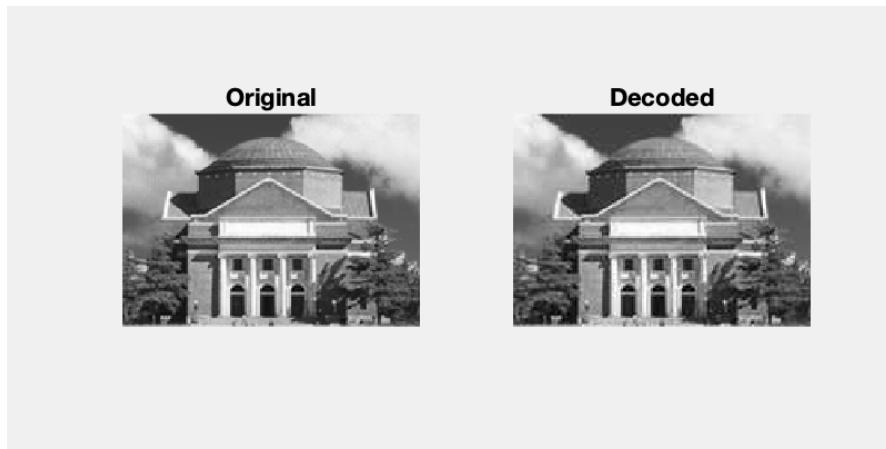


图 11: JPEG 编码解码测试

## 2.12 量化步长减小一半

将量化步长减小一半，即 QTAB 中的每个元素除以 2，重新进行 JPEG 编码解码，重新计算压缩比与 PSNR，结果如图 12。

```
Compression ratio origin: 6.4247
PSNR origin: 31.1874
Compression ratio half: 4.4097
PSNR half: 34.2067
```

图 12: 量化步长减小一半与原结果的比较

可见，压缩比由 6.4247 降低到了 4.4097，PSNR 由 31.1874 dB 提高到了 34.2067 dB。原因可能在于，量化步长减小后，量化精度提高，高频分量的细节更多地保留了下来，因此 PSNR 提高。但由于量化步长减小，量化后的系数更大，导致码流长度增加，压缩比降低。

视觉效果如图 13，与原量化步长相比，变化不明显。

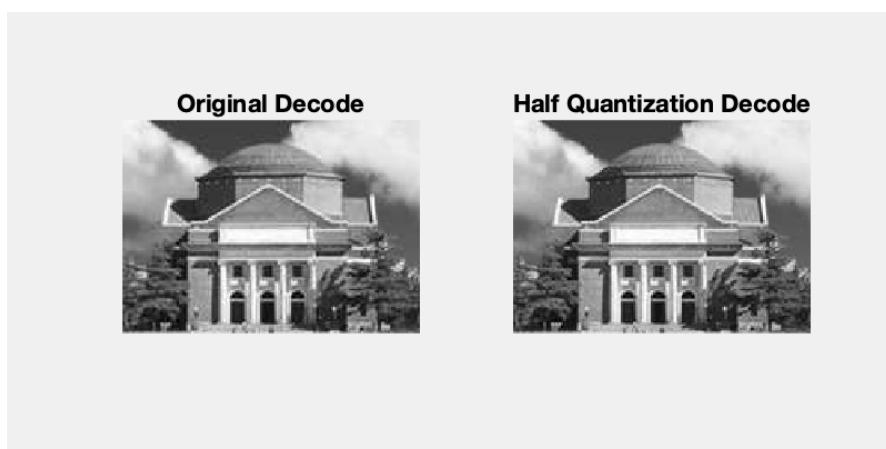


图 13: 量化步长减小一半与原结果解码图像比较

## 2.13 雪花图像 JPEG 编解码

解码图像与指标如图 14 与 15。相比大礼堂图像，雪花图像的压缩比较低、PSNR 较低，视觉效果细节上失真较多。这可能是因为雪花图像的细节更多，高频分量更多，量化后数值较大，压缩比较低；同时，高频分量量化系数较大，细节丢失较多，PSNR 较低。

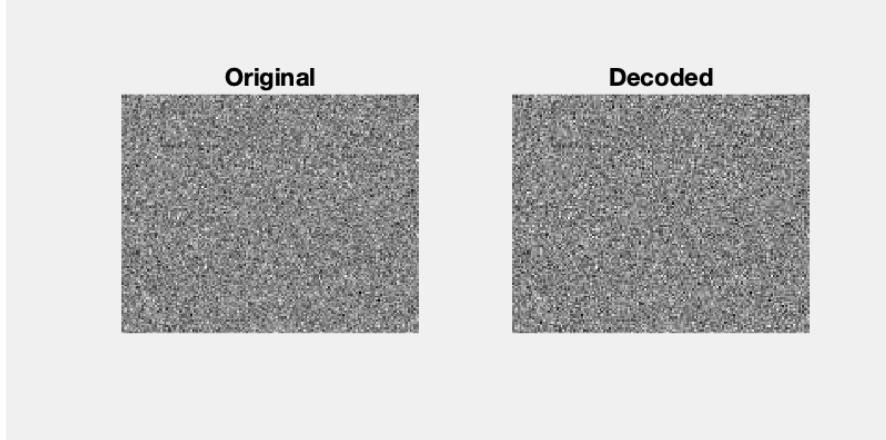


图 14: 雪花图像 JPEG 解码图像

Compression ratio: 3.645  
PSNR: 22.9244

图 15: 雪花图像 JPEG 编解码结果

## 3 信息隐藏

### 3.1 空域隐藏

将信息隐藏在图像的最低有效位，关键代码如下：

```
1 % encode
2 i = bitset(i, 1, hidden_info);
3 % decode
4 hidden_info = logical(bitget(o, 1));
```

对大礼堂图像进行信息隐藏，隐藏信息分别取 01 随机序列、全 1 序列、全 0 序列。隐藏信息后的图像如图 16，解码准确率如图 17。

结果表明，图像观感上几乎无失真。但无论隐藏信息是 01 随机序列（高频分量较多）、全 1 序列（直流）还是全 0 序列，解码准确率均在 50%，附近。这说明空域隐藏信息的方法在 JPEG 编码下几乎无法保留任何信息，不适合信息隐藏。原因在于，JPEG 编码过程中，对图像进行了量化，导致隐藏信息的最低有效位被破坏，无法准确提取。

为了方便代码复用，空域隐藏的代码写进了 JPEG 编解码方法中(`src/jpeg_encode.m`与`src/jpeg_decode.m`)，调用时传入额外的参数 `enc_method`、`dec_method` 与 `hidden_info` 即可，具体实现及用法见代码。之后的其它信息隐藏方法同样写进了 JPEG 编解码方法中，不再赘述。



图 16: 空域隐藏信息后的图像

```
Accuracy: 0.49866
Accuracy: 0.47887
Accuracy: 0.52123
```

图 17: 空域隐藏信息解码准确率

### 3.2 DCT 域隐藏

信息隐藏在每个 DCT 系数的最低有效位，关键代码如下：

```
1 % encode
2 Q = double(bitset(int64(Q), 1, hidden_info));
3 % decode
4 hidden_info = logical(bitget(int64(Q), 1));
```

信息隐藏在部分系数的最低位。与隐藏在所有系数的最低位相比，隐藏在部分系数的最低位，可以减小隐藏信息对图像的影响，提高隐藏信息的隐蔽性。为了最小化隐藏信息对图像的影响，我选取量化矩阵 QTAB 中系数最小的位置用来隐藏信息。这是因为，量化系数越小，最低位等量的变化（加减 1）所对原 DCT 系数的影响越小，失真也越小。关键代码如下。其中量化矩阵系数最小的位置由计算得到，具备很好的通用性。同时，由于各列中系数最小的位置是固定的，可以用矩阵运算处理。

```
1 % encode
2 [~, min_QTAB_idx] = min(QTAB(:));
3 idx_matrix = zeros(8, 8);
4 idx_matrix(mod(min_QTAB_idx - 1, 8) + 1, floor((min_QTAB_idx - 1) /
    ↪ 8) + 1) = 1;
5 [~, min_QTAB_idx] = max(zig_zag_8_8(idx_matrix));
6 Z(min_QTAB_idx, :) = double(bitset(int64(Z(min_QTAB_idx, :)), 1,
    ↪ hidden_info));
7 % decode
```

```

8 [~, min_QTAB_idx] = min(QTAB(:));
9 idx_matrix = zeros(8, 8);
10 idx_matrix(mod(min_QTAB_idx - 1, 8) + 1, floor((min_QTAB_idx - 1) /
→ 8) + 1) = 1;
11 [~, min_QTAB_idx] = max(zig_zag_8_8(idx_matrix));
12 hidden_info = logical(bitget(int64(Z(min_QTAB_idx, :)), 1));

```

信息隐藏在最后一个非零系数之后，用 1 表示 1, -1 表示 0。由于各列非零系数并不固定，因此需要逐列处理，关键代码如下：

```

1 % encode
2 hidden_info = double(hidden_info);
3 hidden_info(hidden_info == 0) = -1;
4 for idx = 1 : min(size(Z, 2), length(hidden_info))
5     non_zero_indices = find(Z(:, idx));
6     if ~isempty(non_zero_indices)
7         last_non_zero_idx = non_zero_indices(end);
8         if last_non_zero_idx == size(Z, 1)
9             Z(last_non_zero_idx, idx) = hidden_info(idx);
10        else
11            Z(last_non_zero_idx + 1, idx) = hidden_info(idx);
12        end
13    else
14        Z(1, idx) = hidden_info(idx);
15    end
16 end
17 % decode
18 hidden_info = logical(zeros(1, size(Z, 2)));
19 for idx = 1 : size(Z, 2)
20     non_zero_indices = find(Z(:, idx));
21     if ~isempty(non_zero_indices)
22         last_non_zero_idx = non_zero_indices(end);
23         hidden_info(idx) = Z(last_non_zero_idx, idx) == 1;
24     end
25 end

```

对大礼堂图像进行信息隐藏，隐藏信息取 01 随机序列。隐藏信息后的图像如图 18，解码准确率与 PSNR 如图 19。

可见，DCT 域三种信息隐藏的方法均能完整地保留信息，解码准确率均为 100%。

从观感上，DCT 域全部最低位隐藏失真较明显，高频分量变得很多。原因在于此方法对每个系数的最低位进行了修改，而高频分量量化系数较大，最低位的变化对原系数的影响较大。同时，原始图像的高频分量较少，量化后多为 0，此时由 0 变 1 导致图像高频分量显著增加，失真明显。

其它几种量化方法对图像的影响较小。其中，压缩率与 PSNR 最高的是第二种 DCT 域部分最低位隐藏。这可能是因为隐藏信息的位置被选取得当（即量化系数最小），对图像的影响最小。

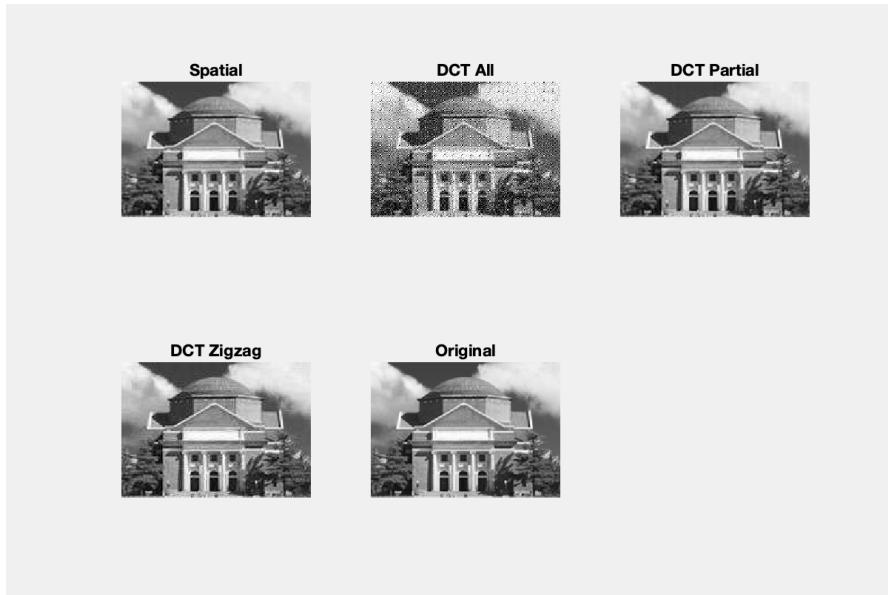


图 18: DCT 域隐藏信息后的图像

<b>Average compression ratio:</b>			
6.4136	2.8681	6.3895	6.1916
<b>Average PSNR:</b>			
31.1738	15.4804	31.1237	28.9401
<b>Average accuracy:</b>			
0.4986	1.0000	1.0000	1.0000

图 19: DCT 域隐藏信息解码准确率 & PSNR

从左到右分别为空域隐藏、DCT 域全部最低位隐藏、DCT 域部分最低位隐藏、DCT 域最后非零系数后隐藏

值得注意的是，最后一种（DCT 域最后非零系数后）隐藏的方法是可以几乎完美地恢复出原始图像的（将最低非零系数置零即可），前提是知道图像采用的是这种隐藏信息的技术。但这种恢复方法不应出现在解码器中。毕竟用于隐藏信息的图像本质上是用来给别人看的，别人大概率不知道隐藏信息的方法。因此没有太多实际意义。

### 3.3 新的信息隐藏方法

由之前的分析可知，选取量化系数最小的位置隐藏信息，可以很好地减小隐藏信息对图像的影响，提高隐藏信息的隐蔽性。但缺点在于，这种方法每 64 个系数只能隐藏 1 位信息，隐藏效率较低。同时，上述隐藏信息的方法形式过于简单，容易被攻击者破解。

因此一种折中的思路是，根据隐藏信息的长度，动态选取若干个量化系数最小的位置隐藏信息。每个分块的隐藏信息的长度由前一个分块的隐藏信息决定（例如隐藏信息的长度为前一个分块的最后两个隐藏信息所表示的数 + 3）。这种方法可以在保证隐藏信息对图像影响较小的前提下，提高隐藏效率。同时，由于后一个分块的隐藏信息选取位置依赖于前一个分块的隐藏信息，排除了攻击者对单个分块隐藏格式暴力破解的可能性，大大提高了

破解的复杂度。

## 4 人脸检测

### 4.1 训练人脸标准

由于我们采用的标准是各种色彩占图片的比例，即对图片大小进行了归一化，因此无需将图片调整为相同大小。

$L$  越大，对颜色的区分度可能更精细。下面对  $L = 3, 4, 5$  进行训练。首先定义将 RGB 值转化为对应标量的函数：

```
1 function num = rgb2scalar(rgb, L)
2     num = bitshift(rgb(1), L - 8) * (2^(2*L)) + bitshift(rgb(2),
3         ↵ L - 8) * (2^L) + bitshift(rgb(3), L - 8);
4 end
```

接着定义训练函数，返回特征向量，关键代码如下：

```
1 function v = train(imgs, L)
2     vals = zeros(1, 2^(3*L));
3     edge = -0.5 : 1 : 2^(3*L) - 0.5;
4     for idx = 1:length(imgs)
5         img = imgs{idx};
6         [h, w, ~] = size(img);
7         img = reshape(img, [h * w, 3]);
8         val = zeros(1, h * w);
9         for i = 1 : h * w
10             val(i) = rgb2scalar(int64(img(i, :)), L);
11         end
12         vals = vals + histcounts(val, edge) / (h * w);
13     end
14     v = vals / length(imgs);
15 end
```

注意这里预定义了  $val = zeros(1, h * w)$ ；而不是采用  $val = []$ ;  $val = [val, ...]$  的方法，这是因为  $val = [val, ...]$  这种写法会导致每次迭代都会重新分配内存，效率较低。这种速度的差异在之后的人脸识别程序中会较明显。

特征向量如图 20。可以看到， $L$  越大，特征向量的维度越高，对颜色的区分度越精细； $L$  较小时，更多相似的颜色被合并到了一起。可以认为，较小的  $L$  得到的特征向量是较大的  $L$  的特征向量对应周围密度值的求和。

### 4.2 人脸检测

算法如下，基本思路是：

(1) 首先选取大小合适的窗口及步长，对每个窗口进行特征提取，然后与训练得到的特征向量进行比较，若相似度超过阈值，则认为检测到人脸。

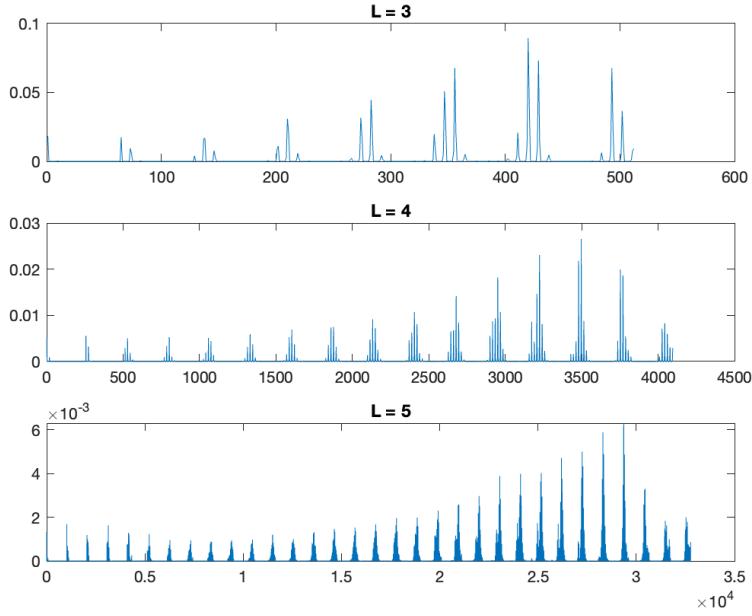


图 20:  $L = 3, 4, 5$  的特征向量

(2) 接着对小窗口进行合并。合并的准则是, 若任意两个窗口的重叠面积占较小窗口的比例超过阈值, 则认为两个窗口重叠, 将其合并。

(3) 然后, 将面积较小窗口视为噪声, 并剔除。

(4) 重复上述步骤, 直到不再有窗口合并。

(5) 在第一轮循环中, 由于存在大量可能是噪声的小窗口, 因此重叠判定的阈值较高, 为 0.5, 剔除阈值较高, 为窗口面积平均值的 1 倍; 在之后的循环中, 降低重叠判定的阈值为 0.25, 降低剔除判定的阈值为窗口面积平均值的 0.5 倍, 以更好地保留并合并人脸处的窗口。

关键代码如下:

```

1 function face_recognition(img, height, width, step_h, step_w, L, v,
2   threshold)
3   [h, w, ~] = size(img);
4   edge = -0.5 : 1 : 2^(3 * L) - 0.5;
5   position = [];
6   % find small windows
7   for i = 1 : step_h : h - height + 1
8     for j = 1 : step_w : w - width + 1
9       window = img(i : i + height - 1, j : j + width - 1, :);
10      [h_, w_, ~] = size(window);
11      window = reshape(window, [h_ * w_, 3]);
12      val = zeros(1, h_ * w_);
13      for k = 1 : h_ * w_
14        val(k) = rgb2scalar(int64(window(k, :)), L);
15      end
16      val = histcounts(val, edge) / (h_ * w_);
```

```

16 % Bhattacharyya distance
17 d = 1 - sum(sqrt(v .* val));
18 if d < threshold
19     position = [position; j, i, width, height];
20 end
21 end
22 disp("Progress: " + i / (h - height + 1) * 100);
23 end

24 % merge and remove small windows
25 flag = true;
26 first_epoch = true;
27 overlap_ratio = 0.5;
28 delete_ratio = 1;
29 while flag
30     idx = 1;
31     flag = false;
32     position = sortrows(position);
33     n = size(position, 1);
34     keep = true(n, 1);
35     for idx = 1 : n - 1
36         if ~keep(idx)
37             continue;
38         end
39         for idy = idx + 1 : n
40             if ~keep(idy)
41                 continue;
42             end
43             overlap = rectint(position(idx, :), position(idy,
44             ↵ :));
45             area = min(position(idx, 3) * position(idx, 4),
46             ↵ position(idy, 3) * position(idy, 4));
47             ratio = overlap / area;
48             if ratio > overlap_ratio
49                 position(idx, 1) = min(position(idx, 1),
50                 ↵ position(idy, 1));
51                 position(idx, 2) = min(position(idx, 2),
52                 ↵ position(idy, 2));
53                 position(idx, 3) = max(position(idx, 1) +
54                 ↵ position(idx, 3), position(idy, 1) +
55                 ↵ position(idy, 3)) - min(position(idx, 1),
56                 ↵ position(idy, 1));
57                 position(idx, 4) = max(position(idx, 2) +
58                 ↵ position(idx, 4), position(idy, 2) +
59                 ↵ position(idy, 4)) - min(position(idx, 2),
60                 ↵ position(idy, 2));
61                 keep(idy) = false;
62             end
63             flag = true;

```

```

54         end
55     end
56 end
57 position = position(keep, :);
58 avg_area = mean(position(:, 3) .* position(:, 4));
59 less_than_avg_area = position(:, 3) .* position(:, 4) <
    ↳ avg_area * delete_ratio;
60 position(less_than_avg_area, :) = [];
61 if any(less_than_avg_area)
62     flag = true;
63 end
64 if first_epoch
65     first_epoch = false;
66     overlap_ratio = 0.25;
67     delete_ratio = 0.5;
68 end
69 end
70 imshow(img);
71 hold on;
72 for i = 1 : size(position, 1)
73     rectangle('Position', position(i, :), 'EdgeColor', 'r',
    ↳ 'LineWidth', 2);
74 end
75 hold off;
76 end

```

我们选取窗口大小为 16\*16 的正方形，宽高步长均为 4。

以 2024 巴黎奥运会朝韩乒乓球选手世纪同框自拍作为验证图片。通过调节颜色阈值，得到识别结果如图 21。

可以看到，通过调节阈值，不同的 L 均可以较好地识别出人脸。同时，我们可以发现，这种仅依靠颜色信息的人脸检测方法无法区分贴合过近的人脸，例如图中右上角的两位选手；无法区分颜色相近、大小相近的物体，例如图中中间运动员露出的左手。

之后，我们选取 2023 年 eesast 合照作为测试图片，得到识别结果如图 22。

当 L=3 时，采用同样的阈值可以很好地识别到大部分人脸。但随着 L 的增大，采用相同的阈值无法很好地识别人脸。即使调整阈值，效果也不如 L=3 时好，容易出现漏检与误检的情况（如图中左边肉色衣服很容易被误检，而一些皮肤较为白皙的同学容易被漏检）。这可能是由于测试图片的人脸颜色细节与训练图片有较大差异，例如部分肤色偏亮、不同人脸的肤色差异较大等，导致 L 较大时需要的阈值偏高；而较高的阈值容易导致背景被误检。

### 4.3 图像变换后的人脸检测

我们选取识别效果最好的 L=3 的情况，固定阈值为 0.475。

顺时针旋转 90° 后的识别结果如图 23。

可见效果几乎不受影响。这可能是因为我们只关注了颜色信息，而不在意人脸的形状。

利用 `imresize` 函数将图片宽度增加一倍后的识别结果如图 24。

可见对识别影响也不大。这说明该方法对形状变换的鲁棒性确实较好。

将图片调亮与调暗，得到识别结果如图 25。

可见调亮后识别效果较差，而调暗后识别效果很好。这印证了之前的猜想：该测试图片相较于训练图片，亮度更低，识别难度更大。

首次启发，我用降低了亮度的图片再次进行了实验，得到识别结果如图 26。

可以看到，降低亮度后，不同的 L 值识别效果均明显得到提升，其中 L=3 时达到了相当高的准确率。这说明基于颜色信息的人脸检测确实严重依赖于亮度信息，提示我们实际应用时需要配合其它特征。

#### 4.4 重新选择人脸样本训练标准

从实验中我们可以发现，根据颜色信息的人脸识别对人脸形状鲁棒，但对肤色亮度敏感，且容易受相近颜色背景干扰，例如人的手部或肉色衣物。

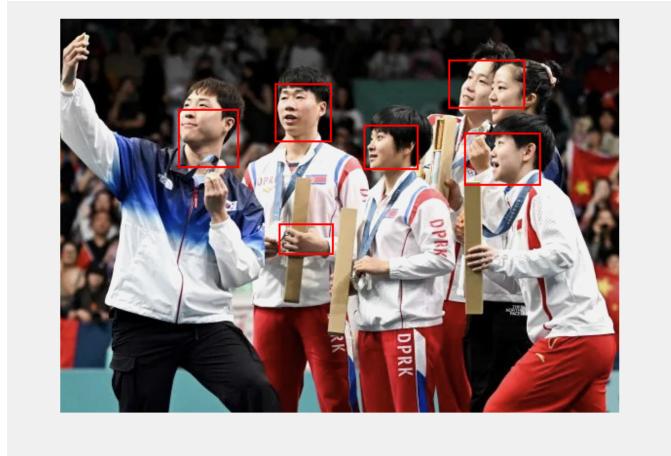
因此，我认为除了颜色，还需综合考虑其它特征，例如形状、纹理等。可能的方法是对颜色求梯度，提取梯度特征，即轮廓特征；或者对颜色进行聚类，提取聚类特征。综合多种特征，预期可以提高人脸识别的准确性，降低误检率。

### 5 实验总结

本次实验主要研究了图像处理的基本原理和方法，实现了 JPEG 编码解码、基于颜色信息的人脸检测等实验。

除此以外，实验中多次尝试提高代码的效率，通过实验掌握了一些小技巧，例如避免重复分配内存、避免循环中的重复计算等，积累了一些经验。

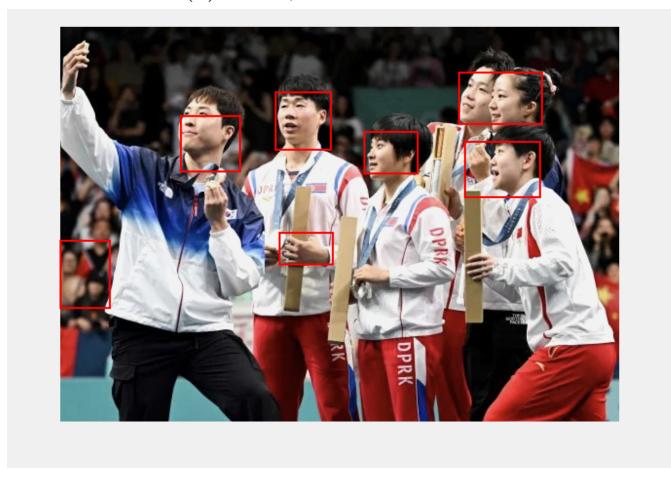
所有代码均位于 `src` 目录下，可供查阅。



(a)  $L = 3$ , threshold = 0.475



(b)  $L = 4$ , threshold = 0.625



(c)  $L = 5$ , threshold = 0.795

图 21: 2024 巴黎奥运会朝韩乒乓球选手世纪同框自拍人脸检测结果

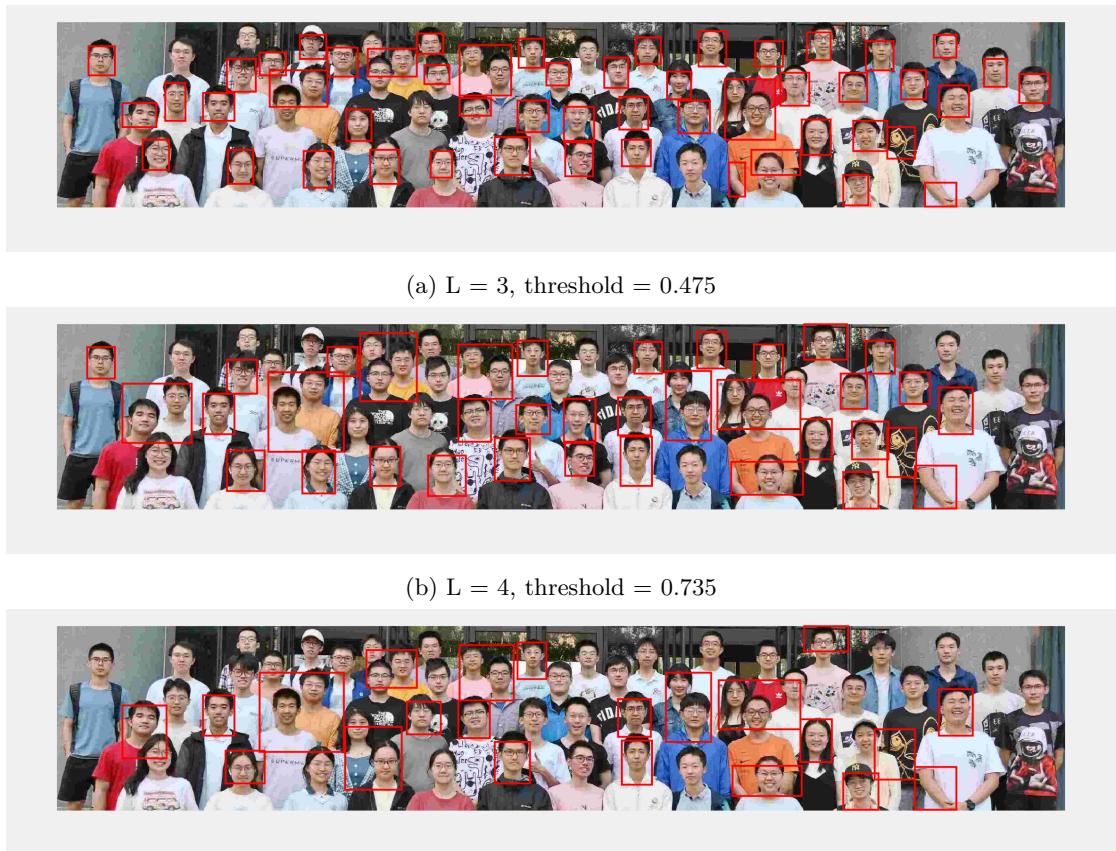


图 22: 2023 年 eesast 合照人脸检测结果 (未调整亮度)

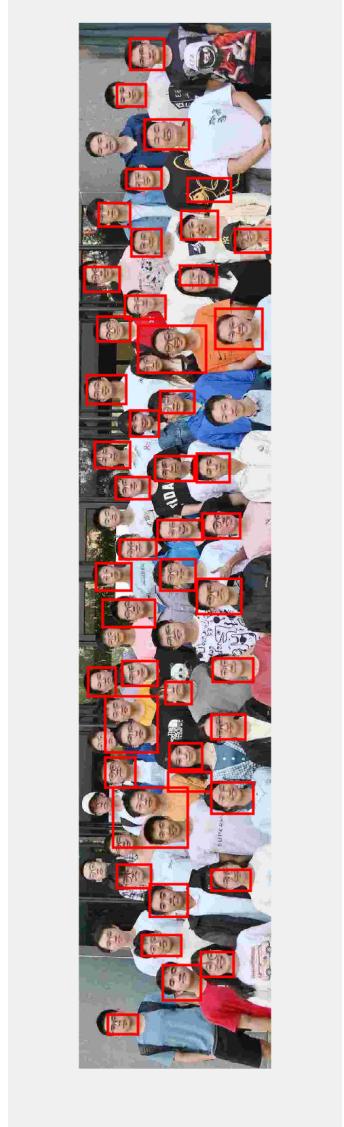


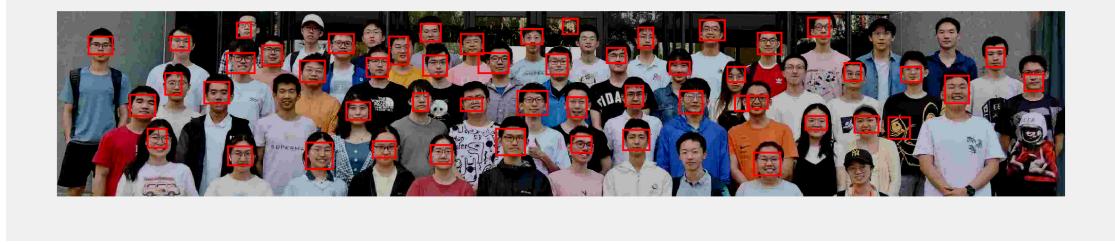
图 23: 2023 年 eesast 合照顺时针旋转 90° 后人脸检测结果



图 24: 2023 年 eesast 合照宽度增加一倍后人脸检测结果

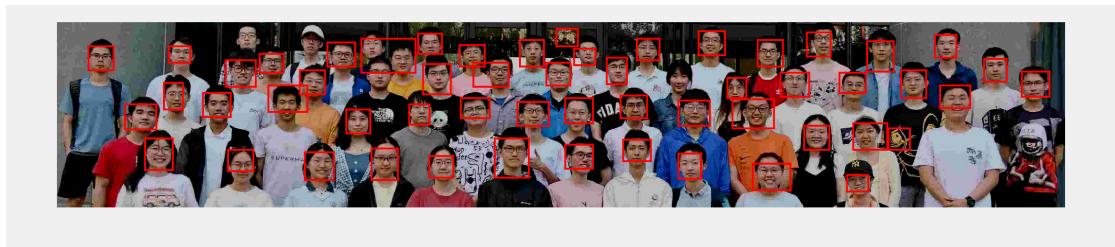


(a) 2023 年 eesast 合照调亮后人脸检测结果

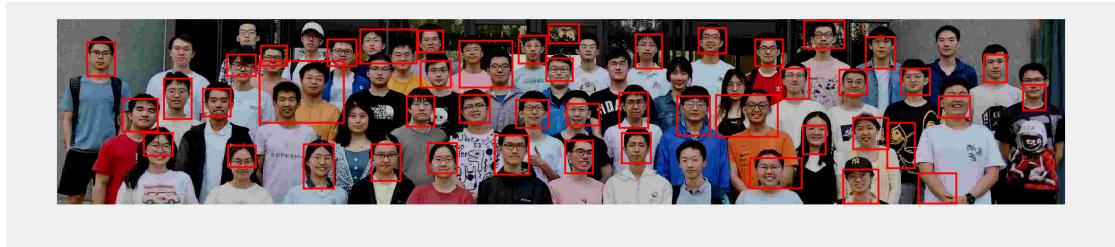


(b) 2023 年 eesast 合照调暗后人脸检测结果

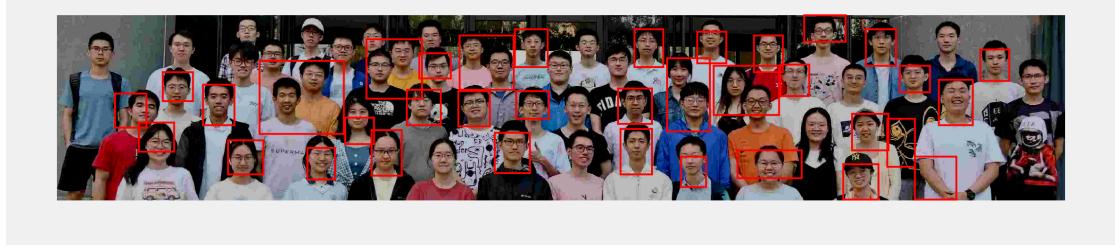
图 25: 2023 年 eesast 合照调亮与调暗后人脸检测结果



(a)  $L = 3$ , threshold = 0.525



(b)  $L = 4$ , threshold = 0.735



(c)  $L = 5$ , threshold = 0.87

图 26: 2023 年 eesast 合照人脸检测结果 (亮度调整)