

MATLAB 大作业之图像处理

无 03 王与进 2020010708

目录

1	文件列表	2
2	练习题	3
2.1	基础知识	3
2.1.1	图像处理工具箱	3
2.1.2	简单的图像处理	3
2.2	图像压缩编码	4
2.2.1	图像的预处理	4
2.2.2	编程实现二维 DCT	5
2.2.3	DCT 系数置零	6
2.2.4	DCT 系数的转置和旋转	7
2.2.5	差分编码的频率响应	9
2.2.6	DC error 和 category	9
2.2.7	zigzag 算法的实现	10
2.2.8	测试图像 DCT、分块和量化	10
2.2.9	JPEG 编码	11
2.2.10	计算压缩比	14
2.2.11	JPEG 解码	14
2.2.12	将量化步长减小为原来的一半之后重新编解码	19
2.2.13	雪花图像的编解码	19
2.3	信息隐藏	20
2.3.1	空域隐藏和提取	20
2.3.2	变换域隐藏和提取	20
2.4	人脸检测	24
2.4.1	人脸识别模型的训练	24
2.4.2	人脸检测算法	24
2.4.3	图像变换后的人脸检测	28
2.4.4	重新选择人脸样本训练标准	30

1 文件列表

```
1 .
2 |_ code
3 |   |_ data % 测试用数据
4 |   |   |_ Faces/
5 |   |   |_ hall.mat
6 |   |   |_ JpegCoeff.mat
7 |   |   |_ snow.mat
8 |   |   |_ test_1.bmp
9 |   |   |_ test_2.bmp
10 |   |   |_ test_3.bmp
11 |   |   |_ test_4.bmp
12 |   |   |_ test_5.bmp
13 |   |_ ac_decode.m % ac解码 % --- 以下为函数 ---
14 |   |_ ac_encode.m % ac编码
15 |   |_ ac_huffman_tree_dict.m % 获取ac编码的huffman tree字典
16 |   |_ binary_decode.m % 解码二进制编码
17 |   |_ binary_encode.m % 编码二进制编码
18 |   |_ calculate_compress_ratio.m % 计算压缩比
19 |   |_ calculate_correct_ratio.m % 计算解码信息正确率
20 |   |_ concat_rgb.m % 拼接rgb值
21 |   |_ dc_decode.m % dc解码
22 |   |_ dc_encode.m % dc编码
23 |   |_ dc_huffman_tree_dict.m % 获取dc编码的huffman tree字典
24 |   |_ extract_face_feature.m % 提取训练集特征
25 |   |_ get_dct_matrix.m % 获取dct值
26 |   |_ get_dct_matrix_with_params.m
27 |   |_ get_dct_params.m
28 |   |_ get_face_area.m % 识别脸部方框
29 |   |_ get_probability_density.m % 获取图片颜色概率密度
30 |   |_ huffman_encode.m % huffman编码
31 |   |_ inv_zig_zag.m % 反zigzag
32 |   |_ jpeg_decode.m % jpeg解码
33 |   |_ jpeg_decode_with_info.m % jpeg解码 (带有隐藏信息)
34 |   |_ jpeg_decode_with_params.m
35 |   |_ jpeg_encode.m % jpeg编码
36 |   |_ jpeg_encode_with_info.m % jpeg编码 (带有隐藏信息)
37 |   |_ jpeg_encode_with_params.m
38 |   |_ my_dct.m % dct算法实现
39 |   |_ psnr.m % 计算PSNR
40 |   |_ run_size_encode.m % run/size编码
41 |   |_ test_ac_encode.m % ac编码测试
42 |   |_ zig_zag.m % zig-zag扫描
43 |   |_ hw_1_1.m % --- 以下为作业脚本 ---
44 |   |_ hw_1_2.m
45 |   |_ hw_2_1.m
46 |   |_ hw_2_2.m
47 |   |_ hw_2_3.m
48 |   |_ hw_2_4.m
49 |   |_ hw_2_5.m
50 |   |_ hw_2_7.m
51 |   |_ hw_2_8.m
```

```

52 | hw_2_9.m
53 | hw_2_11.m
54 | hw_2_12.m
55 | hw_2_13.m
56 | hw_3_1.m
57 | hw_3_2.m
58 | hw_4_1.m
59 | hw_4_2_extra.m
60 | hw_4_2.m % 4-2分为2个文件
61 | hw_4_3_1.m
62 | hw_4_3_2.m
63 | hw_4_3_3.m % 4-3分为3个文件

```

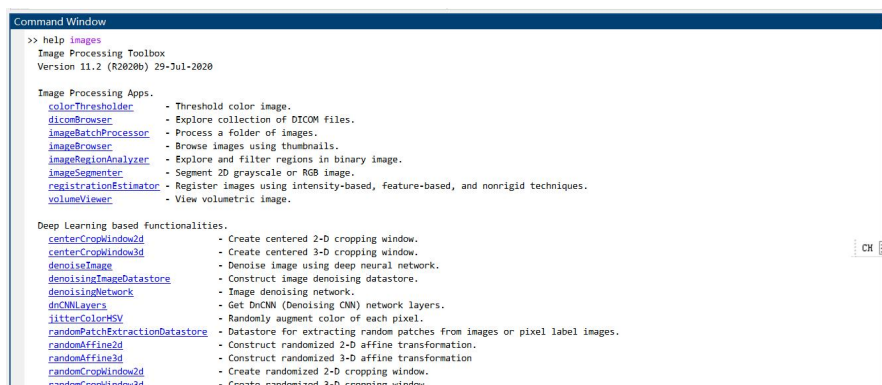
Listing 1: 文件列表

2 练习题

2.1 基础知识

2.1.1 图像处理工具箱

在 MATLAB 终端中输入 help images:



```

Command Window
>> help images
Image Processing Toolbox
Version 11.2 (R2020b) 29-Jul-2020

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

Deep Learning based functionalities.
centerCropWindow2d - Create centered 2-D cropping window.
centerCropWindow3d - Create centered 3-D cropping window.
denoiseImage - Denoise image using deep neural network.
denoisingImageDatastore - Construct image denoising datastore.
denoisingNetwork - Image denoising network.
dncnnLayers - Get DnCNN (Denoising CNN) network layers.
jitterColorHSV - Randomly augment color of each pixel.
randomPatchExtractionDatastore - Datastore for extracting random patches from images or pixel label images.
randomAffine2d - Construct randomized 2-D affine transformation.
randomAffine3d - Construct randomized 3-D affine transformation.
randomCropWindow2d - Create randomized 2-D cropping window.
randomCropWindow3d - Create randomized 3-D cropping window.

```

图 1: help images

2.1.2 简单的图像处理

绘制红色圆形 在本例中, 我使用了 matlab 自带的 rectangle 函数。其中 width/2-r 和 height/2-r 表示圆外切正方形的左下端的坐标, 2*r 表示圆的直径。

```

1 load('data/hall.mat')
2 hall_with_circle = hall_color;
3 imshow(hall_with_circle);
4
5 % get size
6 [height, width, ~] = size(hall_color);
7 r = min(height, width)/2;
8 hold on;
9
10 % draw circle

```

```

11     rectangle('position',[width/2-r,height/2-r,2*r,2*r],'curvature',[1,1],'EdgeColor',
12         'r');

```

Listing 2: 绘制红色圆形

绘制黑白格 在本例中，我们需要计算出每个像素点所在的区间。如果区间序号模 2 得到的结果不同，则将其涂黑。

```

1     block_width = 8;
2     height_blocks = height/block_width;
3     width_blocks = width/block_width;
4     for i = 1:height
5         for j = 1:width
6             if (xor(mod(floor(i/block_width), 2), mod(floor(j/block_width),2) == 1))
7                 for k = 1:3
8                     hall_with_chessboard(i,j,k) = 0;
9                 end
10            end
11        end
12    end
13

```

Listing 3: 绘制棋盘

最终结果如下：

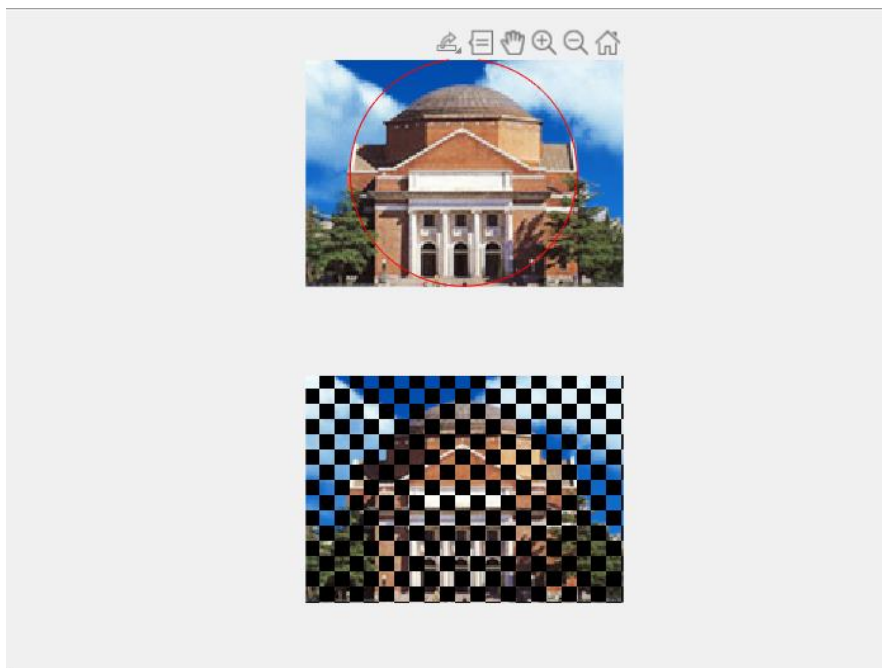


图 2: result 1

2.2 图像压缩编码

2.2.1 图像的预处理

图像的预处理是将每个像素的灰度值减去 128。这一变换可以在变换域中进行，理由如下：

考虑二维 DCT 的矩阵形式：

$$\mathbf{C} = \mathbf{D}\mathbf{P}\mathbf{D}^T \quad (1)$$

显然我们有：

$$\mathbf{D}(\mathbf{P}_1 + \mathbf{P}_2)\mathbf{D}^T = \mathbf{D}\mathbf{P}_1\mathbf{D}^T + \mathbf{D}\mathbf{P}_2\mathbf{D}^T \quad (2)$$

即二维 DCT 是一个线性变换。我们不妨设 $\mathbf{P}_2 = 128\mathbf{J}$ ，其中 \mathbf{J} 为全 1 矩阵，大小为 $N \times N$ ，则 $\mathbf{J} = \vec{l}\vec{l}^T$ ，其中 \vec{l} 为长度为 1 的全 1 向量。

由 DCT 的定义可知：

$$\mathbf{D} = \sqrt{\frac{2}{N}} \begin{pmatrix} \sqrt{\frac{1}{2}} & \cdots & \sqrt{\frac{1}{2}} \\ \vdots & \ddots & \vdots \\ \cos \frac{(N-1)\pi}{2N} & \cdots & \cos \frac{(N-1)(2N-1)\pi}{2N} \end{pmatrix} \quad (3)$$

考虑到 $\sum_{i=0}^N \cos \frac{k(2i-1)}{2N} = 0$ ，显然 $\mathbf{D}\vec{l} = [\sqrt{N}, 0, \dots, 0]^T$ ，则我们可以得出：

$$\mathbf{D}\mathbf{P}_2\mathbf{D}^T = 128\mathbf{D}\vec{l}\vec{l}^T\mathbf{D}^T = 128N \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{pmatrix} \quad (4)$$

即 \mathbf{P}_2 的 DCT 如上式所示，记为 \mathbf{K} 。

根据题意，我们只需要先将图像进行 DCT 变换，再将左上角位置的元素减去 $128N$ 即可。为验证这一结论，我们从测试图片中截取 8×8 的一块进行计算，其中用到了 MATLAB 中内置的 `dct2` 函数：

```
1 load('data/hall.mat')
2 test_img = double(hall_gray(1:8,1:8));
3
4 result_1 = dct2(test_img - 128);
5 result_2 = dct2(test_img);
6 result_2(1,1) = result_2(1,1) - 128*8;
7
8 disp(sum(sum(abs(result_1 - result_2))));
```

Listing 4: 两种 DCT 变换形式比较

可以得到两者差的绝对值之和为 9.0705×10^{-13} ，这说明在误差范围之内，两种计算方法是等效的。

2.2.2 编程实现二维 DCT

二维 DCT 的代码见 `my_dct.m`。我们将 DCT 变换扩充到了非方阵情形：

```
1 function [C] = my_dct(P)
2     [h, w] = size(P);
3     D_1 = zeros(h,h);
4     D_1(1,:) = 0.5^0.5;
5     disp(h);
6     for i = 2:h
7         D_1(i,:) = cos(pi*(i-1)*(1:2:(2*h-1))/(2*h));
8     end
9     D_1 = D_1 * (2/h) ^ 0.5;
10
11     D_2 = zeros(w,w);
```

```

12 D_2(1,:) = 0.5^0.5;
13 for i = 2:w
14     D_2(i,:) = cos(pi*(i-1)*(1:2:(2*w-1))/(2*w));
15 end
16 D_2 = D_2 * (2/w) ^ 0.5;
17
18 C = D_1 * double(P) * D_2';
19 end

```

Listing 5: 自定义 DCT

测试如下：

```

1 load('data/hall.mat')
2 test_img = double(hall_gray(1:7,1:8));
3
4 result_1 = dct2(test_img);
5 result_2 = my_dct(test_img);
6
7 disp(result_1);
8 disp(result_2);
9
10 disp(sum(sum(abs(result_1 - result_2))));

```

Listing 6: 两种 DCT 变换形式比较

可以得到两者差的绝对值之和为 4.7719×10^{-12} ，这说明在误差范围之内，两种计算方法是等效的。

2.2.3 DCT 系数置零

为了方便起见，我们将生成 DCT 系数的过程单独抽取为一个函数 `dct_param.m`。

在正式进行实验之前，我们需要探讨 DCT 系数的意义：在 DCT 系数中，右侧表示高频分量，左侧表示低频分量。更加具体地讲，左上侧系数表示图像的整体亮度，左下侧系数表示纵向变化的纹理的强度，右上侧系数表示横向变化的纹理的强度，右下侧系数表示横竖两个方向均迅速变化的成分。由于常见的图片以缓慢变化为主，所以 DCT 系数左方取值较大，右方取值较小。我们猜测：去除右侧 4 列，对图像呈现效果影响不大，但去除左侧 4 列影响会较大。

验证代码如下：

```

1 load('data/hall.mat')
2 P = double(hall_gray);
3 subplot(3,1,1);
4 imshow(uint8(P));
5 title('original');
6
7 % DCT with right
8 [height, width] = size(P);
9 D_1 = get_dct_params(height);
10 D_2 = get_dct_params(width);
11
12 C = D_1 * P * D_2';
13 C(:,end-3:end) = 0;
14
15 % IDCT with right
16
17 P1 = D_1' * C * D_2;

```

```

18 subplot(3,1,2);
19 imshow(uint8(P1));
20 title('right 0');
21
22 % DCT with left
23 C = D_1 * P * D_2';
24 C(:,1:4) = 0;
25
26 % IDCT with left
27
28 P2 = D_1' * C * D_2;
29 subplot(3,1,3);
30 imshow(uint8(P2));
31 title('left 0');

```

Listing 7: DCT 置零方式比较

结果如下：

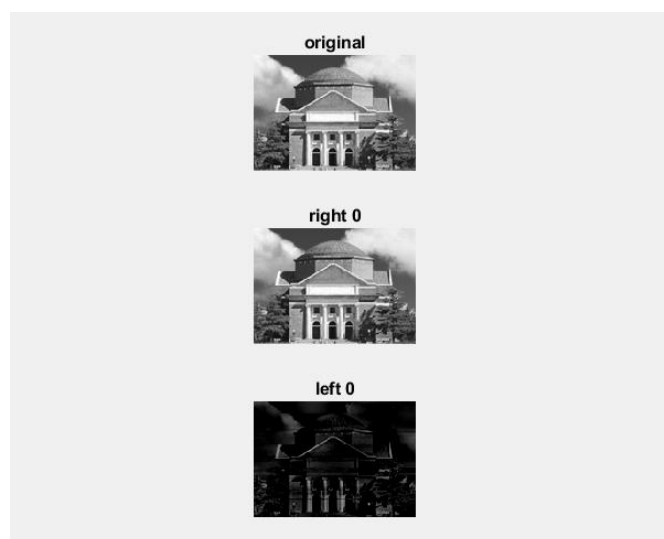


图 3: result 2

可以看出，与原始图像相比，若将左侧系数置零，则图像受损严重且较暗；而将右侧系数置零，则图像几乎可以完全恢复，与预期结果一致。

为了进一步分析两种变换对图像的影响，我们截取一部分进行分析：

可以得出以下结论：

- (1) 若删除左侧系数，我们只能看到纵向的高频分量，而低频分量接近 0。
- (2) 若删除右侧系数，则纵向的高频分量被清除，纵向颜色变化较为平缓，但亮度几乎不变。

2.2.4 DCT 系数的转置和旋转

代码如下：

```

1 load('data/hall.mat')
2 P = double(hall_gray(1:120,1:120));
3 subplot(1,4,1);
4 imshow(uint8(P));
5 title('original');

```

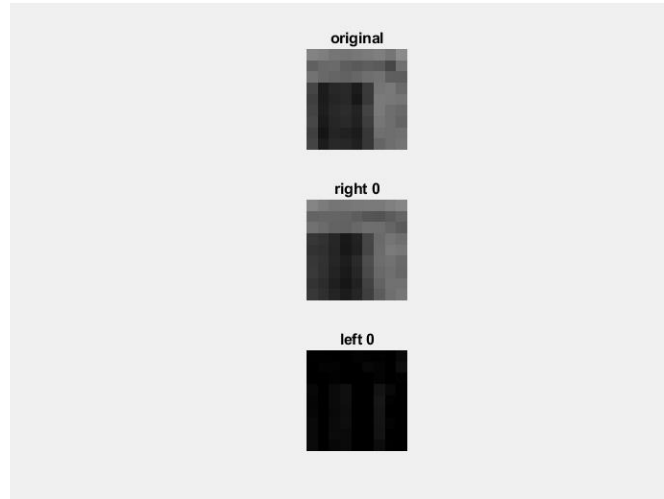


图 4: result 3

```

6
7 [height, width] = size(P);
8 D_1 = get_dct_params(height);
9 D_2 = get_dct_params(width);
10
11 % DCT with transpose
12 C = D_1 * P * D_2';
13
14 % IDCT with transpose
15 P1 = D_1' * C * D_2;
16 subplot(1,4,2);
17 imshow(uint8(P1));
18 title('transpose');
19
20 % IDCT with rot 90
21 P2 = D_1' * rot90(C) * D_2;
22 subplot(1,4,3);
23 imshow(uint8(P2));
24 title('rot 90');
25
26 % IDCT with rot 180
27 P2 = D_1' * rot90(C,2) * D_2;
28 subplot(1,4,4);
29 imshow(uint8(P2));
30 title('rot 180');

```

Listing 8: DCT 系数的转置和旋转

结果如下：

据此，我们可以得出以下结论：

(1) DCT 系数转置后的逆变换即为转置后的图像，证明如下：

$$\mathbf{D}^T \mathbf{C}^T \mathbf{D} = (\mathbf{D}^T \mathbf{C} \mathbf{D})^T = \mathbf{P}^T \quad (5)$$

(2) DCT 系数旋转 90 度后，原先右上角的系数将会移动到左上角，而原先左下角的系数将会移到右下角。这意味着，图片的横向纹理较强。

(3) DCT 系数旋转 180 度后，原先右下角的系数将会移动到左上角。这意味着，图片的纹理会呈现出斜向变化。

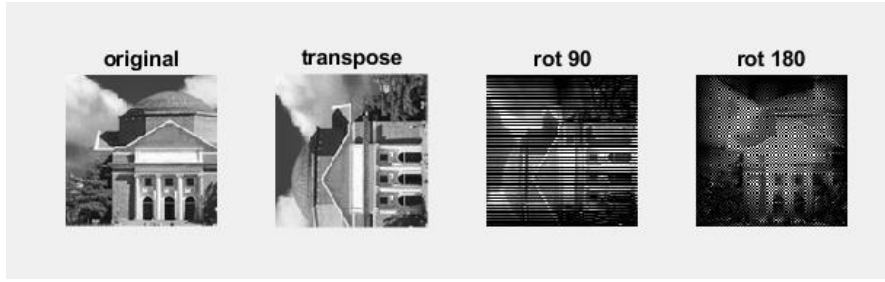


图 5: result 4

2.2.5 差分编码的频率响应

由于差分编码的表达式可以写作：

$$y(n) = x(n-1) - x(n) \quad (6)$$

可以用以下代码绘制其频率响应：

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

Listing 9: 差分编码频响

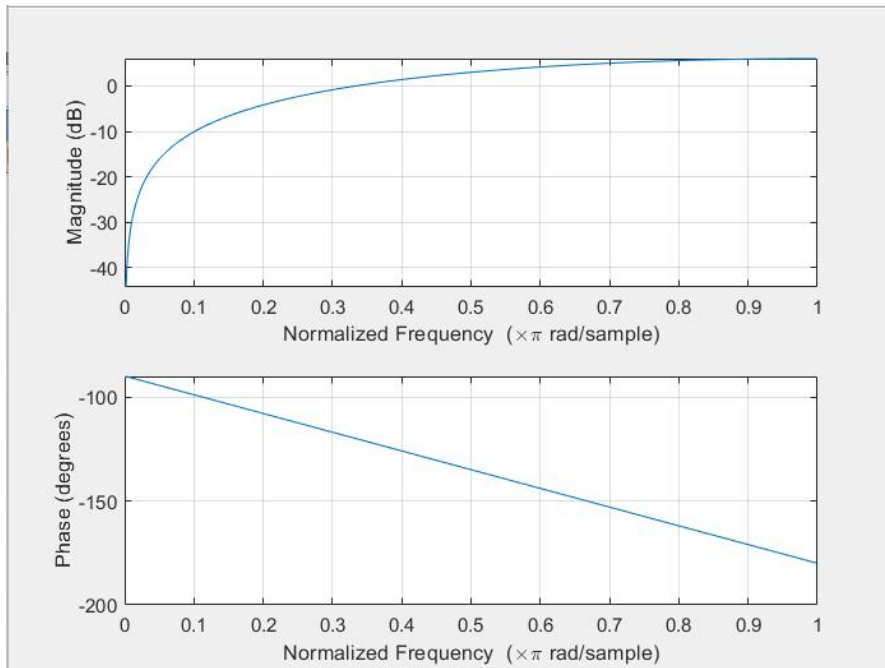


图 6: 频率响应

由图可知，这是一个高通滤波器。DC 系数的高频分量更多。

2.2.6 DC error 和 category

$$\begin{cases} Category = \lfloor \log_2(|error|) \rfloor + 1, (error \neq 0) \\ Category = 0, (error = 0) \end{cases} \quad (7)$$

2.2.7 zigzag 算法的实现

观察发现 zigzag 算法的特点：

- (1) 横纵坐标之和单调递增
- (2) 扫描方向与横纵坐标的奇偶性有关

据此可以设计 zigzag 算法。zigzag 算法见 zig_zag.m：

```
1 function [idx] = zig_zag(mat_size)
2     idx = [];
3     y = reshape(1:mat_size*mat_size, mat_size, mat_size);
4
5     for i = 2:2*mat_size
6         for j = max(i - mat_size, 1):min(mat_size, i-1)
7             if mod(i, 2) == 0
8                 idx = [idx, y(i-j, j)];
9             else
10                idx = [idx, y(j, i-j)];
11            end
12        end
13    end
14 end
```

Listing 10: zigzag 算法

测试结果如下，可见算法正确：

```
>> hw_2_6
 1   2   3   4   5   6   7   8
 9  10  11  12  13  14  15  16
17  18  19  20  21  22  23  24
25  26  27  28  29  30  31  32
33  34  35  36  37  38  39  40
41  42  43  44  45  46  47  48
49  50  51  52  53  54  55  56
57  58  59  60  61  62  63  64

Columns 1 through 22
 1   2   9  17  10   3   4  11  18  25  33  26  19  12   5   6  13  20  27  34  41  49

Columns 23 through 44
42  35  28  21  14   7   8  15  22  29  36  43  50  57  58  51  44  37  30  23  16  24

Columns 45 through 64
31  38  45  52  59  60  53  46  39  32  40  47  54  61  62  55  48  56  63  64
```

图 7: zigzag 测试结果

2.2.8 测试图像 DCT、分块和量化

DCT、分块和量化的步骤如下：

- (1) 将图片分为 8x8 的若干小块。若图片的长和宽不能被 8 整除，则将其补全。
- (2) 将每个小块预处理（-128）之后进行 DCT 变换。
- (3) 将 DCT 变换后的矩阵进行 zigzag 扫描，并进行拼接。

```
1 load('data/hall.mat');
2 load('data/JpegCoeff');
3 P = double(hall_gray);
4
5 % fill the picture to block it
6 [h, w] = size(P);
7 if(mod(h, 8) ~= 0)
8     P(h+1:(floor(h/8)+1)*8,:) = 0;
```

```

9 end
10 if(mod(w, 8) ~=0)
11     P(:,w+1:(floor(w/8)+1)*8) = 0;
12 end
13
14 % block
15 [h, w] = size(P);
16 block_h = h/8;
17 block_w = w/8;
18 k = 1;
19 result = [];
20 for i = 0:block_h-1
21     for j = 0:block_w-1
22         block = P(8*i + 1: 8*(i+1),8*j+1:8*(j+1)) - 128; % get the block
23         D_block = dct2(block);
24         D_block = round(D_block./QTAB); % quantation
25         zigzag_D_idx = zigzag(8);
26         result(:,k) = D_block(zigzag_D_idx); % zigzag
27         k = k + 1;
28     end
29 end

```

Listing 11: DCT、分块和量化

2.2.9 JPEG 编码

上一步计算中得到了图像的 DC 系数和 AC 系数。定义函数 `dc_params.m` 和 `ac_params.m` 分别计算 DC 编码和 AC 编码。

DC 编码可以分为两部分：huffman encode 和 binary encode。

DC 编码的流程如下：

- (1) 对 DC 系数进行差分。
- (2) 对差分后的系数进行 huffman encode 和 binary encode。

此处需要特别指出的是，0 的 binary encode 不是 '0'，而是 ""。

```

1 function [dc_code] = dc_encode(params)
2     dc_params = params(1, :);
3     dc_params_diff = [];
4     dc_params_diff(1) = dc_params(1);
5     dc_len = length(dc_params);
6
7     % diff, and code
8     dc_code = strcat(huffman_encode(dc_params_diff(1)), category_encode(dc_params_diff(1)));
9     for i = 2:dc_len
10         dc_params_diff(i) = dc_params(i-1) - dc_params(i);
11         huffman_code = huffman_encode(dc_params_diff(i));
12         binary_code = binary_encode(dc_params_diff(i));
13         dc_code = strcat(dc_code, huffman_code, binary_code);
14     end
15 end

```

Listing 12: DC code

AC 编码可以分为两部分：run/size encode 和 binary encode。

AC 编码的流程较为复杂：

- (1) 对系数进行 zigzag 扫描。

(2) 我们需要得出每一个非零系数的 Size 和 Run，查表得到 Huffman 编码，再与 Amplitude 拼接。

(3) 如果遇到 16 个连零，我们需要将其编码为 1111111001 (ZRL)；如果遇到结束，我们需要编码 1010 (EOB)。

```

1 function [ac_code] = ac_encode(params)
2     ZRL = '1111111001';
3     EOB = '1010';
4     ac_code = '';
5     ac_params = params(2:end, :);
6     for i = 1:size(ac_params, 2)
7         vec = ac_params(:,i); % params after zigzag
8         zero_num = 0;
9         for j = 1:length(vec)
10            if vec(j) ~= 0
11                while zero_num >= 16
12                    ac_code = strcat(ac_code, ZRL);
13                    zero_num = zero_num - 16;
14                end
15                ac_code = strcat(ac_code, run_size_encode(zero_num, vec(j)));
16                ac_code = strcat(ac_code, binary_encode(vec(j)));
17                zero_num = 0;
18            else
19                zero_num = zero_num + 1;
20            end
21        end
22        ac_code = strcat(ac_code, EOB);
23    end
24 end

```

Listing 13: AC code

其中辅助函数如下：

```

1 function [huffman_code] = huffman_encode(num)
2     load('data/JpegCoeff.mat');
3     if(num == 0)
4         category = 0;
5     else
6         category = floor(log2(abs(num))) + 1;
7     end
8
9     huffman_code_length = DCTAB(category+1,1);
10    huffman_code = DCTAB(category+1, 2:1 + huffman_code_length);
11    huffman_code = num2str(huffman_code);
12    huffman_code = strrep(huffman_code, ' ', '');
13
14 end

```

Listing 14: huffman encode

```

1 function [category_code] = binary_encode(num)
2     load('data/JpegCoeff.mat');
3     if(num == 0)
4         category_code = '';
5     else
6         category_code = dec2bin(abs(num));
7     end
8     if(num < 0)
9         for i = 1:length(category_code)

```

```

10         category_code(i) = num2str(~str2num(category_code(i)));
11     end
12 end
13 end

```

Listing 15: binary encode

```

1 function [run_size_code] = run_size_encode(run, size)
2     load('data/JpegCoeff.mat');
3     if(size == 0)
4         category = 0;
5     else
6         category = floor(log2(abs(size))) + 1;
7     end
8     idx = run*10 + category;
9     code_length = ACTAB(idx,3);
10    run_size_code = ACTAB(idx,4:4+code_length-1);
11    run_size_code = num2str(run_size_code);
12    run_size_code = strrep(run_size_code, ' ', '');
13 end

```

Listing 16: run/size encode

DC 编码测例如下，与说明文档中相一致，可见结果正确：

```

>> params = [10,8,60];
>> dc_encode(params)

ans =

    '10110100111101110001011'

```

图 8: DC 编码测试结果

AC 编码测例如下，与说明文档中相一致，可见结果正确：

```

Command Window
>> test_ac_encode
101110100111111110001011111111001111011111010
fx >>

```

图 9: AC 编码测试结果

为了方便起见，我们将分块、DCT、量化的算法封装为一个函数：

```

1 function [result] = get_dct_matrix(P)
2 % fill the picture to block it
3 [h, w] = size(P);
4 if(mod(h, 8) ~=0)
5     P(h+1:(floor(h/8)+1)*8,:) = 0;
6 end
7 if(mod(w, 8) ~=0)
8     P(:,w+1:(floor(w/8)+1)*8) = 0;
9 end
10
11 % block
12 [h, w] = size(P);
13 block_h = h/8;
14 block_w = w/8;

```

```

15 k = 1;
16 result = [];
17 for i = 0:block_h-1
18     for j = 0:block_w-1
19         block = P(8*i + 1: 8*(i+1),8*j+1:8*(j+1)) - 128; % get the block
20         D_block = dct2(block);
21         D_block = round(D_block./QTAB); % quantation
22         zigzag_D_idx = zigzag(8);
23         result(:,k) = D_block(zigzag_D_idx); % zigzag
24         k = k + 1;
25     end
26 end
27 end

```

Listing 17: 分块、DCT、量化

最后，我们将图片的 DC 码流、AC 码流、长度、宽度写入文件：

```

1 load('data/hall.mat');
2 load('data/JpegCoeff');
3 P = double(hall_gray);
4
5 dct_matrix = get_dct_matrix(P);
6 dc_code = dc_encode(dct_matrix);
7 disp(length(dc_code));
8 ac_code = ac_encode(dct_matrix);
9 disp(length(ac_code));
10 [h, w] = size(P);
11 save('jpegcodes.mat', 'dc_code', 'ac_code', 'h', 'w');

```

Listing 18: 写入文件

方便起见，我们定义函数实现 jpeg 编码，输入为图像数据和码流文件名称：

```

1 function [] = jpeg_encode(original_picture, stream_name)
2     load('data/JpegCoeff');
3     dct_matrix = get_dct_matrix(original_picture);
4     dc_code = dc_encode(dct_matrix);
5     ac_code = ac_encode(dct_matrix);
6     [h, w] = size(original_picture);
7     save(stream_name, 'dc_code', 'ac_code', 'h', 'w');
8 end

```

Listing 19: jpeg-encode.m

2.2.10 计算压缩比

由于原图的尺寸为 20160 bytes，变换后 DC 码流长为 2031，AC 码流长为 23072，总长度为 25103 bits (3137 bytes)，计算得出压缩比为 6.4265。

2.2.11 JPEG 解码

JPEG 解码的基本步骤如下：

(1) DC 解码和 AC 解码

考虑到 DC 码和 AC 码中的 huffman code 和 run/size code 均为 huffman 编码，我们可以构造 huffman 树。由于 huffman 编码较小，所以我们可以考虑直接构造哈希表，在解码时直接检索即可：

```

1 function [huffman_tree_dict] = dc_huffman_tree_dict()
2     load('data/JpegCoeff.mat');
3     [code_num, len] = size(DCTAB);
4     huffman_tree_dict = containers.Map();
5
6     for i = 1:code_num
7         % get huffman code
8         huffman_code_length = DCTAB(i,1);
9         huffman_code = DCTAB(i, 2:1 + huffman_code_length);
10        huffman_code = num2str(huffman_code);
11        huffman_code = strrep(huffman_code, ' ', '');
12        disp([num2str(i-1), '->', huffman_code]);
13        huffman_tree_dict(huffman_code) = i - 1;
14    end
15 end
16

```

Listing 20: dc huffman tree

```

1 function [huffman_tree_dict] = ac_huffman_tree_dict()
2     load('data/JpegCoeff.mat');
3     [code_num, len] = size(ACTAB);
4     huffman_tree_dict = containers.Map();
5
6     for i = 1:code_num
7         % get huffman code
8         huffman_code_length = ACTAB(i,3);
9         huffman_code = ACTAB(i, 4:3 + huffman_code_length);
10        huffman_code = num2str(huffman_code);
11        huffman_code = strrep(huffman_code, ' ', '');
12        disp([num2str(i-1), '->', huffman_code]);
13        huffman_code = strcat('x', huffman_code);
14        huffman_tree_dict(huffman_code) = i;
15    end
16 end
17

```

Listing 21: ac huffman tree

对于 DC 码流解码, 哈希表构造完毕之后, 我们首先将码流一一与哈希表对比, huffman code 解码成功后即可得到 binary code 的长度, 再对 binary code 进行解码即可得到原来的数值, 完毕之后跳过 binary code, 进行解码下一个 huffman code, 直到码流的末尾。最后进行反差分即可。代码如下:

```

1 function [dc_decode_data] = dc_decode(dc_stream)
2     len = length(dc_stream);
3     huffman_code_temp = '';
4     huffman_code_dict = dc_huffman_tree_dict();
5     dc_decode_data = [];
6     i = 1;
7
8     % diff decode
9     while i <= len
10        huffman_code_temp = strcat(huffman_code_temp, dc_stream(i));
11        if(isKey(huffman_code_dict, huffman_code_temp))
12            % get the decode data's length
13            dc_decode_data_length = huffman_code_dict(huffman_code_temp);
14            % get the binary code
15            i = i + 1;

```

```

16         if(dc_decode_data_length)
17             binary_code = dc_stream(i:i + dc_decode_data_length - 1);
18         else
19             binary_code = '';
20         end
21         % decode data
22         dc_decode_data = [dc_decode_data, binary_decode(binary_code)];
23         % reset the huffman code
24         huffman_code_temp = '';
25         % jump
26         i = i + dc_decode_data_length;
27     else
28         i = i + 1;
29     end
30 end
31
32 % sum
33 for i = 1:length(dc_decode_data) - 1
34     dc_decode_data(i+1) = dc_decode_data(i) - dc_decode_data(i+1);
35 end
36 end
37

```

Listing 22: dc decode

我们可以将编码之前的原始数据和解码之后的数据进行比较，发现我们可以完全复原其信息：

```

>> sum((dc_decode(dc_code))-dct_matrix(1,:))

ans =

     0

```

图 10: DC 解码测试结果

对于 AC 解码，我们首先针对每一个小块构建一个长为 63 的向量（注意除去了 DC 系数），将码流一一与哈希表对比，huffman code 解码成功后即可得到 run 和 size，再分别进行解码得到之前 0 的数目和数值，完毕之后跳过 binary code，进行解码下一个 huffman code，直到码流的末尾。同时要特别注意遇到 EOB 和 ZRL 的情况。

```

1 function [ac_decode_data] = ac_decode(ac_stream)
2     len = length(ac_stream);
3     huffman_code_temp = '';
4     huffman_code_dict = ac_huffman_tree_dict();
5     ac_decode_data_vec = zeros(1,63);
6     ac_decode_data = [];
7     i = 1;
8     idx = 1;
9
10    ZRL = '11111111001';
11    EOB = '1010';
12
13    % diff decode
14    while i <= len
15        huffman_code_temp = strcat(huffman_code_temp, ac_stream(i));
16        if(isKey(huffman_code_dict, huffman_code_temp))
17            % get the decode data's length
18            raw_data = huffman_code_dict(huffman_code_temp);

```



```

19     binary_data_run = floor(raw_data / 10);
20     binary_data_size = mod(raw_data, 10);
21     while(binary_data_run > 0)
22         idx = idx + 1;
23         binary_data_run = binary_data_run - 1;
24     end
25     % get the binary code
26     i = i + 1;
27     if(binary_data_size)
28         binary_code = ac_stream(i:i + binary_data_size - 1);
29     else
30         binary_code = '';
31     end
32     % decode data
33     ac_decode_data_vec(idx) = binary_decode(binary_code);
34     idx = idx + 1;
35     % reset the huffman code
36     huffman_code_temp = '';
37     % jump
38     i = i + binary_data_size;
39     elseif(strcmp(huffman_code_temp, ZRL) == 1)
40         for j = 1:16
41             idx = idx + 1;
42         end
43         i = i + 1;
44         huffman_code_temp = '';
45     elseif(strcmp(huffman_code_temp, EOB) == 1)
46         ac_decode_data = [ac_decode_data; ac_decode_data_vec];
47         ac_decode_data_vec = zeros(1,63);
48         i = i + 1;
49         idx = 1;
50         huffman_code_temp = '';
51     else
52         i = i + 1;
53     end
54 end
55 ac_decode_data = ac_decode_data';
56 end
57

```

Listing 23: ac decode

我们可以将编码之前的原始数据和解码之后的数据进行比较，发现我们可以完全复原其信息：

```

>> sum(sum(ac_decode(ac_code) - dct_matrix(2:end, :)))
ans =
    0

```

图 11: AC 解码测试结果

(2) 反 zigzag 扫描

反 zigzag 扫描的思路和 zigzag 扫描类似，只不过是将方向反过来：

```

1 function [y] = inv_zig_zag(vec)
2     mat_size = sqrt(length(vec));
3     zig_zag_idx = zig_zag(mat_size);
4     y = zeros(1, mat_size);

```

```

5     for i = 1:mat_size * mat_size
6         y(zig_zag_idx(i)) = zig_zag_idx(i);
7     end
8     y = reshape(y, mat_size, mat_size);
9     y = y';
10 end
11

```

Listing 24: inv zig zag

(3) 反量化、逆 DCT 变换和图片拼接

```

1     for i = 0:block_h - 1
2         for j = 0:block_w - 1
3             block = full_matrix(:,k);
4             block = inv_zig_zag(block);
5             block = block.*QTAB;
6             block = idct2(block);
7             block = block + 128;
8             picture(8*i + 1: 8*(i+1),8*j + 1:8*(j+1)) = uint8(block);
9             k = k + 1;
10        end
11    end
12

```

Listing 25: 反量化、逆 DCT 变换和图片拼接

方便起见，我们将解码封装为函数：

```

1 function [picture] = jpeg_decode(stream_file)
2     load(stream_file);
3     load('data/JpegCoeff.mat');
4
5     picture = zeros(h, w);
6     % decode
7     dc_matrix = dc_decode(dc_code);
8     ac_matrix = ac_decode(ac_code);
9     full_matrix = [dc_matrix; ac_matrix];
10    disp(full_matrix);
11
12    % inv zig zag & inv quat
13    block_h = h/8;
14    block_w = w/8;
15    k = 1;
16
17    for i = 0:block_h - 1
18        for j = 0:block_w - 1
19            block = full_matrix(:,k);
20            block = inv_zig_zag(block);
21            block = block.*QTAB;
22            block = idct2(block);
23            block = block + 128;
24            picture(8*i + 1: 8*(i+1),8*j + 1:8*(j+1)) = uint8(block);
25            k = k + 1;
26        end
27    end
28 end

```

Listing 26: jpeg-decode

测试代码如下：

```

1 load('data/hall.mat')
2 subplot(1,2,1);
3 imshow(uint8(hall_gray));
4 title("original");
5
6 jpeg = jpeg_decode('jpegcodes.mat');
7 subplot(1,2,2);
8 imshow(uint8(jpeg));
9 title("jpeg");
10
11 mse = sum((double(jpeg) - double(hall_gray)).^2, 'all')/(h*w);
12 psnr = 10*log10(255*255/mse);
13 disp(psnr);

```

Listing 27: jpeg 测试

jpeg 解码结果如下:

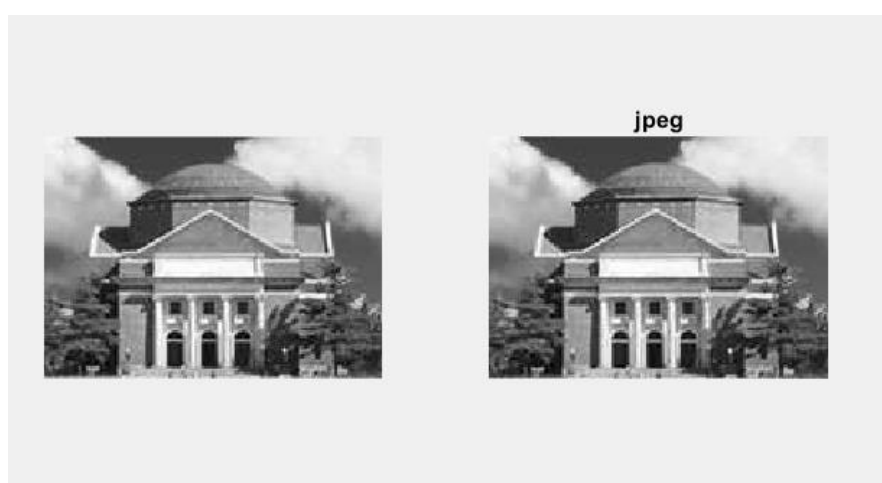


图 12: jpeg 解码测试结果

可以看出, jpeg 的恢复效果很好。计算得到图像的 PSNR 值为 31.1874dB。

2.2.12 将量化步长减小为原来的一半之后重新编解码

减小之后, PSNR 值变为 34.2067dB。DC 码流长为 2410, AC 码流长为 34164, 总长度为 36574 bits (4571 bytes) 由于原图尺寸为 20160 bytes, 压缩比为 4.4104。

由此可见, 图像 PSNR 值升高, 压缩比降低, 但仍然没有出现较大失真。

2.2.13 雪花图像的编解码

PSNR 值为 22.9244 dB。DC 码流长为 1403, AC 码流长为 43546, 总长度为 44949 bits (5618 bytes) 由于原图尺寸为 20160 bytes, 压缩比为 3.5885。图片的失真度较高。

原因如下: jpeg 编码适用于色彩变换较为平缓、高频分量较小的图片, 而雪花图片的色彩分布没有规律可言, 高频分量较大, 因此在压缩之后失真较大。



图 13: 量化步长减小后的 jpeg 解码测试结果

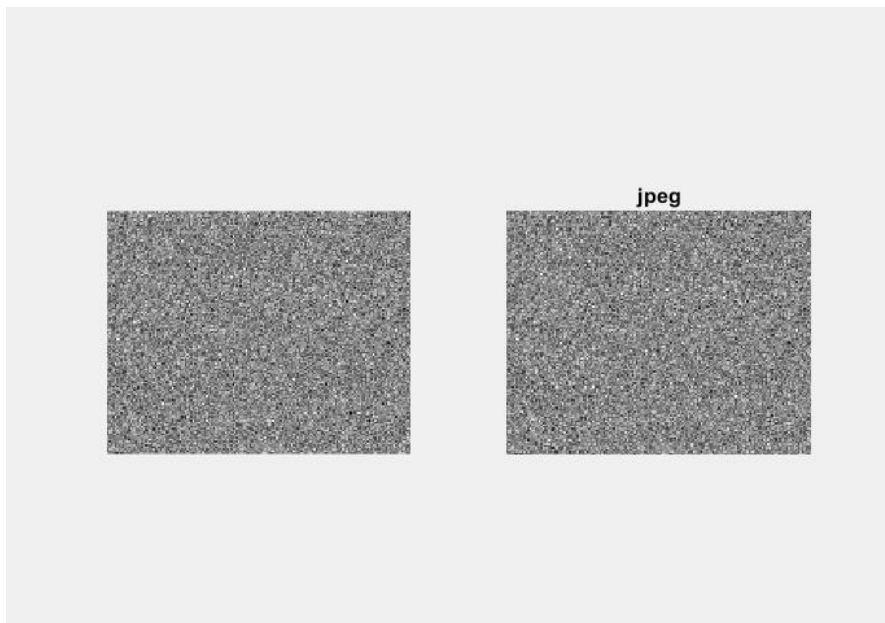


图 14: 雪花图像的编解码

2.3 信息隐藏

2.3.1 空域隐藏和提取

为了使得结果具有普适性，我们使用随机信息序列进行隐藏。我们将信息保存为二进制码流，然后储存至每个像素的最后一位。

可以看出，图片几乎无失真，但提取信息的正确率在 50% 左右，这意味着其抗 jpeg 编码能力很差。

2.3.2 变换域隐藏和提取

公平起见，我们仍然使用随机 0/1 序列进行测试。很显然可以猜测得到，方法 2 和 3 相比于方法 1 储存的信息少、图片失真率小。为了让方法 2 和方法 3 具有可比性，在方法 2 中我们在每一个块中只选取一个位进行储存信息，这样与方法 3 存储信息的密度相同。在方法 2 中为了比较不同替换策略的影响，此处我设置了三种替换策略：选取最大的量化值 (2-1)、随机选取 (2-2)、选取最小的量化值 (2-3)。

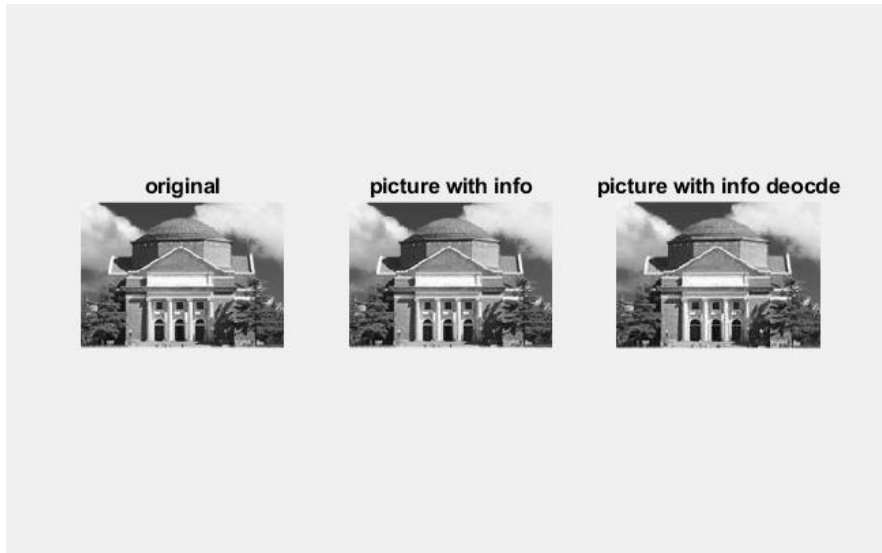


图 15: 空域隐藏和提取

```
>> hw_3_1
0.5001

>> hw_3_1
0.4945

>> hw_3_1
0.5002

>> hw_3_1
0.5007
```

图 16: 空域隐藏和提取正确率

方法 1: 用信息位逐一替换掉每一个量化之后的 DCT 系数的最低位, 再进行熵编码

方法 2: 用信息位逐一替换掉若干量化之后的 DCT 系数的最低位, 再进行熵编码

方法 3: 将信息位追加在每一个块 zig-zag 顺序的最后一个非零 DCT 系数之后

编码、解码的代码如下所示:

```
1 function [dc_code, ac_code, h, w, hidden_info] = jpeg_encode_with_info(original_picture,
2   stream_name, QTAB, DCTAB, ACTAB, method)
3   dct_matrix = get_dct_matrix_with_params(original_picture, QTAB);
4   [dct_h, dct_w] = size(dct_matrix);
5   hidden_info = uint8(randi([0 1], dct_h * dct_w, 1));
6   hidden_info = reshape(hidden_info, dct_h, dct_w);
7   [h, w] = size(original_picture);
8   zig_zag_idx = zig_zag(8);
9   m = median(QTAB, 'all');
10  switch (method)
11      case('method_1')
12          dct_with_info = double(bitset(int64(dct_matrix), 1, hidden_info));
13      case('method_2_1')
14          [~, max_idx] = max(QTAB(zig_zag_idx));
15          dct_with_info = dct_matrix;
16          dct_with_info(max_idx,:) = double(bitset(int64(hidden_info(max_idx,:)), 1,
17              hidden_info(max_idx,:)));
18          hidden_info = hidden_info(max_idx,:);
19      case('method_2_2')
20          random_idx = 32;
```

```

19         dct_with_info = dct_matrix;
20         dct_with_info(random_idx,:) = double(bitset(int64(hidden_info(random_idx,:)), 1,
hidden_info(random_idx,:)));
21         hidden_info = hidden_info(random_idx,:);
22         case('method_2_3')
23             [~, min_idx] = min(QTAB(zig_zag_idx));
24             dct_with_info = dct_matrix;
25             dct_with_info(min_idx,:) = double(bitset(int64(hidden_info(min_idx,:)), 1,
hidden_info(min_idx,:)));
26             hidden_info = hidden_info(min_idx,:);
27         case('method_3')
28             dct_with_info = dct_matrix;
29             hidden_info = int8(hidden_info(1:dct_w));
30             hidden_info_tmp = hidden_info;
31             for i = 1:length(hidden_info)
32                 if hidden_info_tmp(i) == 0
33                     hidden_info_tmp(i) = -1;
34                 end
35             end
36             idx = 1;
37             for i = 1:dct_w
38                 for j = dct_h:-1:1
39                     if(dct_with_info(j, i) ~= 0)
40                         if (j ~= dct_h)
41                             dct_with_info(j + 1, i) = hidden_info_tmp(idx);
42                         else
43                             dct_with_info(j, i) = hidden_info_tmp(idx);
44                         end
45                         break
46                     end
47                 end
48                 idx = idx + 1;
49             end
50         otherwise
51             ME = MException('available options: method_1/method_2_1/method_2_2/method_2_3/
method_3');
52             throw(ME);
53         end
54         dc_code = dc_encode(dct_with_info, DCTAB);
55         ac_code = ac_encode(dct_with_info, ACTAB);
56         save(stream_name, 'dc_code', 'ac_code', 'h', 'w');
57     end

```

Listing 28: 信息编码

```

1 function [picture, decode_info] = jpeg_decode_with_info(stream_file, QTAB, DCTAB, ACTAB,
method)
2     load(stream_file);
3
4     picture = zeros(h, w);
5     % decode
6     dc_matrix = dc_decode(dc_code, DCTAB);
7     ac_matrix = ac_decode(ac_code, ACTAB);
8     full_matrix = [dc_matrix; ac_matrix];
9     [dct_h, dct_w] = size(full_matrix);
10    m = median(QTAB, 'all');
11    zig_zag_idx = zig_zag(8);
12
13    switch (method)
14        case('method_1')

```

```

15         decode_info = bitget(int64(full_matrix), 1);
16     case('method_2_1')
17         [~, max_idx] = max(QTAB(zig_zag_idx));
18         decode_info = bitget(int64(full_matrix(max_idx,:)), 1);
19     case('method_2_2')
20         random_idx = 32;
21         decode_info = bitget(int64(full_matrix(random_idx,:)), 1);
22     case('method_2_3')
23         [~, min_idx] = min(QTAB(zig_zag_idx));
24         decode_info = bitget(int64(full_matrix(min_idx,:)), 1);
25     case('method_3')
26         decode_info = int8(zeros(dct_w, 1));
27         for i = 1:dct_w
28             for j = dct_h:-1:1
29                 if(full_matrix(j,i) ~= 0)
30                     if(full_matrix(j,i) > 0)
31                         decode_info(i) = 1;
32                     elseif(full_matrix(j,i) < 0)
33                         decode_info(i) = 0;
34                     end
35                     break
36                 end
37             end
38         end
39     otherwise
40         ME = MException('available options: method_1/method_2_1/method_2_2/method_2_3/
41         method_3');
42         throw(ME);
43     end
44
45     % inv zig zag & inv quat
46     block_h = h/8;
47     block_w = w/8;
48     k = 1;
49     for i = 0:block_h - 1
50         for j = 0:block_w - 1
51             block = full_matrix(:,k);
52             block = inv_zig_zag(block);
53             block = block.*QTAB;
54             block = idct2(block);
55             block = block + 128;
56             picture(8*i + 1: 8*(i+1),8*j + 1:8*(j+1)) = uint8(block);
57             k = k + 1;
58         end
59     end
60 end

```

Listing 29: 信息解码

三种方法的压缩比、PSNR、准确率、主观评判质量如下表所示：

生成的图像如下所示：

由此可以得出，三种方法都能够正确解码隐藏信息。其中，方法 2-3 和方法 3 具有较好的效果：图像的 PSNR 值和压缩比均较高。

表 1: 变换域隐藏结果					
	方法 1	方法 2-1	方法 2-2	方法 2-3	方法 3
PSNR/dB	15.5001	26.0098	29.5916	26.0061	28.9016
压缩比	2.8784	5.4431	6.0461	6.5612	6.1916
正确率	1	1	1	1	1
主观评判质量	差	中	中	好	好



图 17: 变换域隐藏图像

2.4 人脸检测

2.4.1 人脸识别模型的训练

是否需要调整图片的大小 不需要，因为我们只关注人脸的颜色。

L=3、4、5 时，所得人脸特征 v 之间的关系 由于 RGB 值是一个 uint8 数值，所以 L 的值意味着提取 RGB 之中的前 L 位。L 越大，标号相同的颜色就越少、所能表示的颜色就越精确、但数据量也越大。

2.4.2 人脸检测算法

首先，我们需要将所有的 R、G、B 值拼接为一个值。同时为了权衡 L 的大小对识别的影响，我们也将提取位数 L 设置为参数：

```

1 function [val] = concat_rgb(r, g, b, l)
2     ignore_bits = 8 - l;
3     r = bitshift(r, -ignore_bits, 'uint8');
4     g = bitshift(g, -ignore_bits, 'uint8');
5     b = bitshift(b, -ignore_bits, 'uint8');
6     val = bitshift(r, l*2, 'uint32') + bitshift(g, l, 'uint32') + b;

```


7 end

Listing 30: RGB 值的拼接

之后，我们使用拼接好的值计算概率密度：

```
1 function [destiny_list] = get_probability_density(img, l)
2     [h, w, c] = size(img);
3     img = reshape(img, h*w, c);
4     destiny_list = zeros(1, 2^(3 * l));
5     for i = 1:h * w
6         feature = concat_rgb(img(i, 1), img(i, 2), img(i, 3), l);
7         destiny_list(feature + 1) = destiny_list(feature + 1) + 1;
8     end
9     destiny_list = destiny_list/(h*w);
10 end
```

Listing 31: 概率密度计算

一个概率密度提取示例如下，从上到下分别为 $l = 3, 4, 5$ 的情形：



图 18: 概率密度提取示例

可以较为容易地看出， l 越大，数据量越大，但对于颜色的分辨能力就越精确。接下来就可以准备训练特征了。

训练标准特征 计算训练集中人脸特征的平均值。由于训练集中的图片几乎全部区域均为人脸，所以我们无需手动定位。计算公式如下：

$$\mathbf{v} = \frac{1}{I} \sum_i \mathbf{u}(R_i) \quad (8)$$

分块 将图像划分为小块分别与标准特征比较。

计算与人脸标准的距离 定义待定特征和人脸标准的距离如下：

$$d(\mathbf{u}(R), \mathbf{v}) = 1 - \rho(\mathbf{u}(R), \mathbf{v}) = 1 - \sum_n \sqrt{u_n v_n} \quad (9)$$

比较 判断待定特征和人脸标准的相似程度，如果相似，则使用锚定框标记：

$$R \in Face \quad \text{if} \quad d(\mathbf{u}(R), \mathbf{v}) < \epsilon \quad (10)$$

训练代码如下：

```
1 function [destiny_list] = extract_face_feature(l)
2   destiny_list = zeros(1, 2^(3 * l));
3   for i = 1:33
4     picture = imread(strcat('data/Faces/', num2str(i), '.bmp'));
5     destiny_list_single = get_probability_density(picture, l);
6     destiny_list = destiny_list + destiny_list_single;
7   end
8   destiny_list = destiny_list/33;
9 end
```

Listing 32: 特征提取

检测代码如下：

```
1 function [] = get_face_area(file_name, ref_destiny, l, threshold, block_size)
2   % get the feature
3   subplot(2,1,1);
4   idx = 1:length(ref_destiny);
5   plot(idx, ref_destiny);
6
7   % split the picture
8   P = imread(file_name);
9
10  subplot(2,1,2);
11  imshow(P);
12
13  % block
14  [h, w, c] = size(P);
15  block_h = floor(h/block_size);
16  block_w = floor(w/block_size);
17  title(strcat('threshold=', num2str(threshold), ' l=', num2str(l), ' block-size=', num2str(
18    block_size)));
19
20  for i = 0:block_h-1
21    for j = 0:block_w-1
22      block = P(block_size * i + 1: block_size * (i + 1), block_size * j + 1: block_size
23        * (j + 1), :); % get the block
24      hypo_destiny = get_probability_density(block, l);
25      if(1 - sum(sqrt(hypo_destiny).*sqrt(ref_destiny)) < threshold)
26        rectangle('position', [block_size*j + 1, block_size*i + 1, block_size,
27          block_size], 'EdgeColor', 'r');
28      end
29    end
30  end
31 end
```

Listing 33: 人脸检测

选取若干测试图片（19、20、21），可以发现该检测方法的一些问题：

(1) 基本可以识别出人脸特征。

- (2) 容易将手或与皮肤颜色相近的背景识别为人脸。
- (3) 对于黑色人种的识别效果较差。

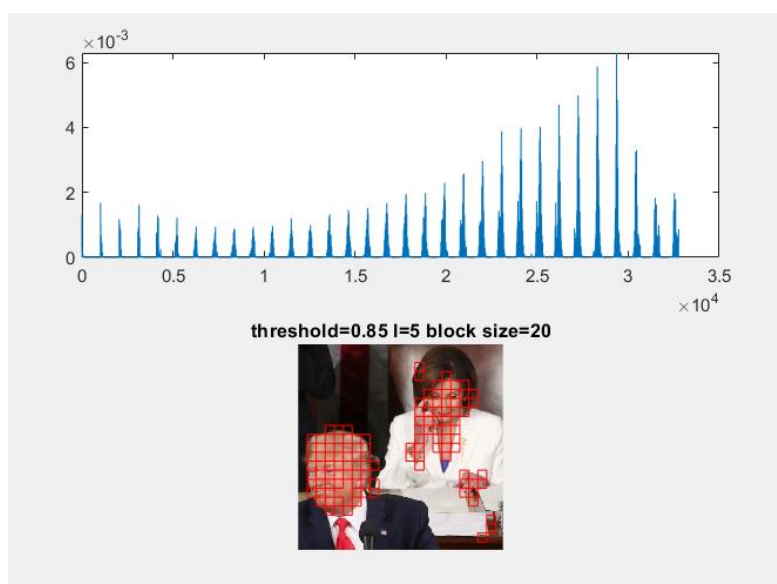


图 19: 识别样例 1

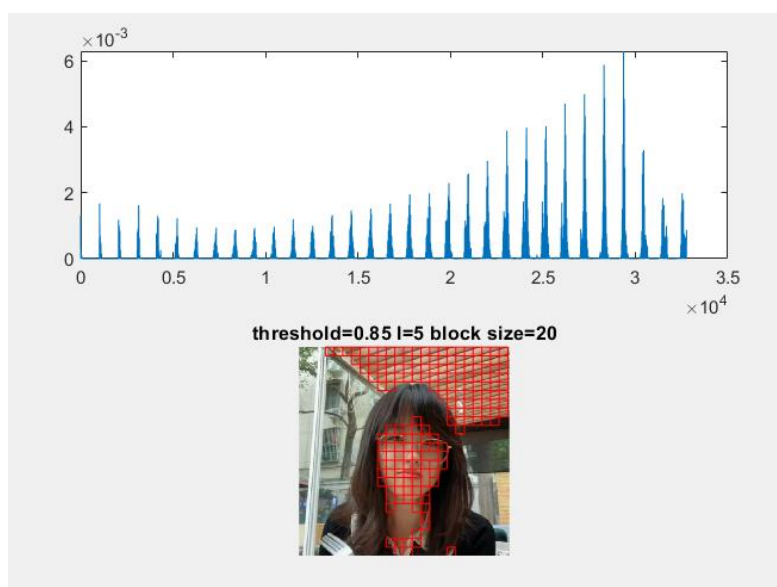


图 20: 识别样例 2

之后我们选取一张合照，并寻找最适合其的超参。对一张合照枚举 l (3、4、5)、threshold (0.6、0.7、0.8、0.9)、block size (10、20) 三个值，得到的结果如图 (22、23)：

可以得出以下结论：

- (1) l 越大，方块被识别为人脸的概率就越大。
- (2) threshold 越大，方块被识别为人脸的概率就越大。
- (3) block size 应该与人脸的尺寸相近或略小。

经过比对，我选出了效果较好的参数组，其中第一组的效果如图24：

- (1) threshold = 0.6, l = 4, block-size = 10

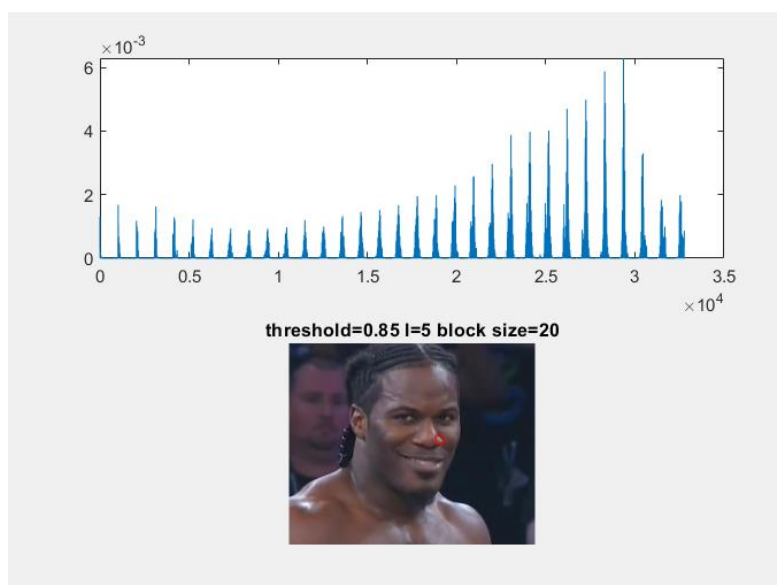


图 21: 识别样例 3

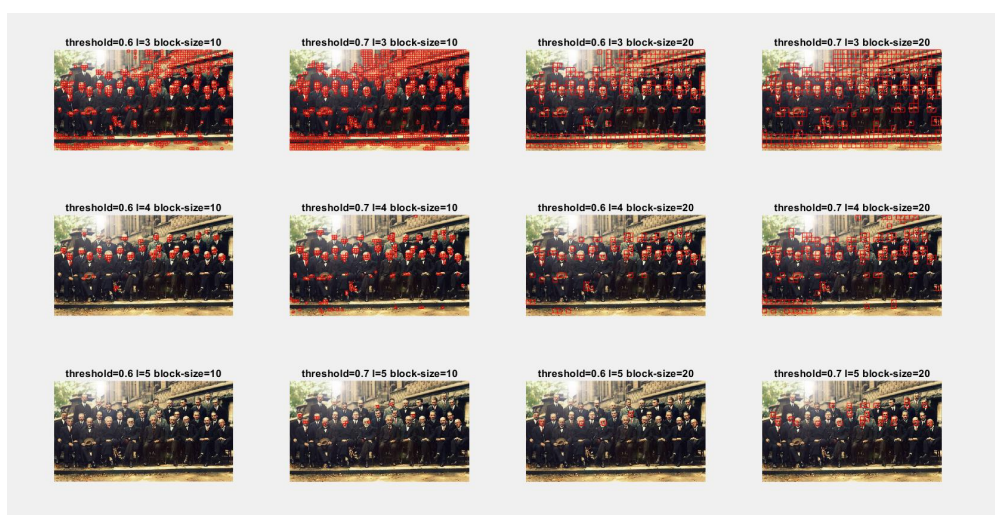


图 22: 合照人脸检测 1

(2) threshold = 0.7, l = 5, block-size = 20

(3) threshold = 0.8, l = 5, block-size = 10

2.4.3 图像变换后的人脸检测

此处仍选择之前的最佳超参: threshold = 0.6, l = 4, block-size = 10。测试代码如下:

```

1 l = 4;
2 block_size = 10;
3 threshold = 0.6;
4
5 P = imread('data/test_3.bmp');
6
7 P_2 = rot90(P);
8 get_face_area(P_2, ref_destiny, l, threshold, block_size);
9 title('rotate 90 degrees');

```



图 23: 合照人脸检测 2



图 24: threshold = 0.6, l = 4, block-size = 10

```

10
11 [h, w, c] = size(P);
12 P_3 = imresize(P, [h, w*2], 'nearest');
13 get_face_area(P_3, ref_destiny, l, threshold, block_size);
14 title('enlarge width');
15
16 P_4 = imadjust(P, [.2 .3 0; .6 .7 1], []);
17 get_face_area(P_4, ref_destiny, l, threshold, block_size);
18 title('adjust the color');

```

Listing 34: 图像变换

图片顺时针旋转 90 度 见图25

宽度拉伸为之前的 2 倍 见图26

改变图片颜色 见图27

可以得出以下结论：旋转和改变尺寸，对识别效果影响不算大，但改变图片颜色对于识

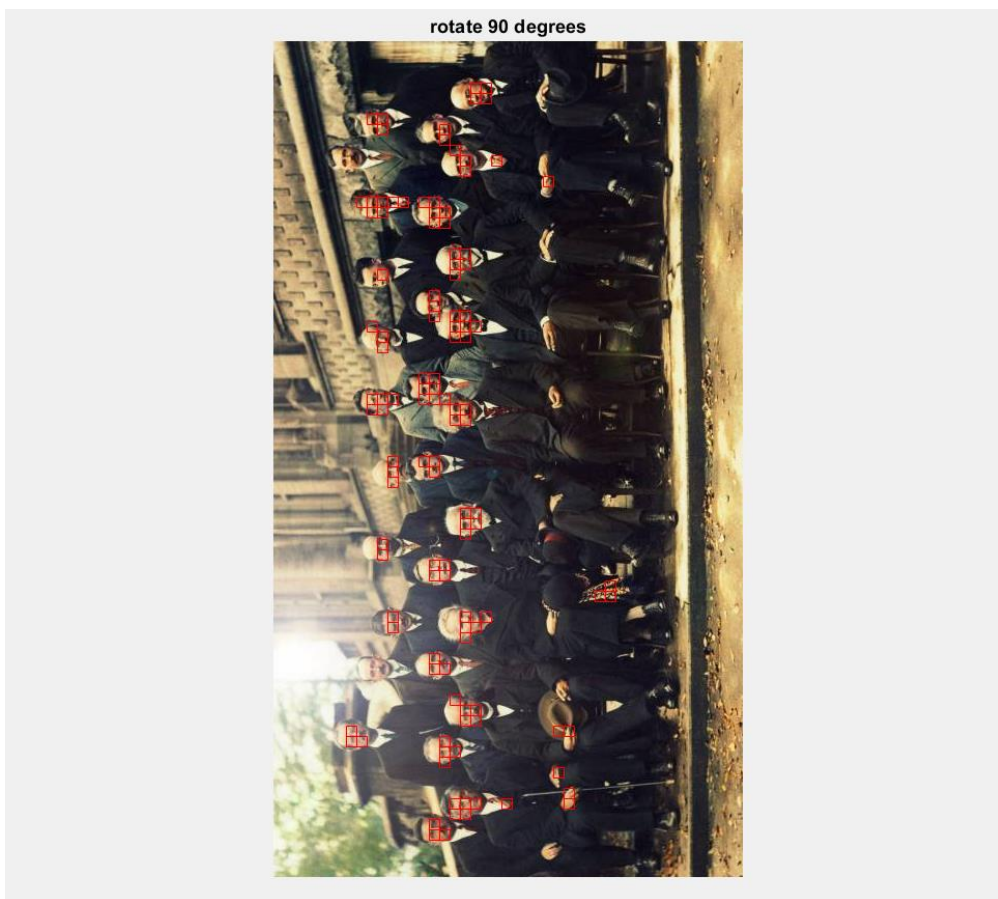


图 25: 图片顺时针旋转 90 度

别效果影响巨大。这是因为我们的判别准则是基于颜色进行的。

2.4.4 重新选择人脸样本训练标准

综合上述实验，我认为更合适的人脸样本训练标准如下：

- (1) 样本应该涵盖更多的人种。
- (2) 由于以上识别方法容易把身体的其它部位或背景颜色识别为人脸，我们应该制定一个与轮廓有关的识别算法，使用轮廓和颜色进行综合判断。
- (3) 样本应该涵盖多种光照角度下的情况，使得特征更具有鲁棒性。



图 26: 宽度拉伸为之前的 2 倍



图 27: 改变图片颜色