Xi'an Jiaotong-Liverpool University
西交利物浦大学

EEE330 - Image Processing
Assessment 4

Student Name: Wang Xiaoyang
Student ID: 1405888

May 15, 2018

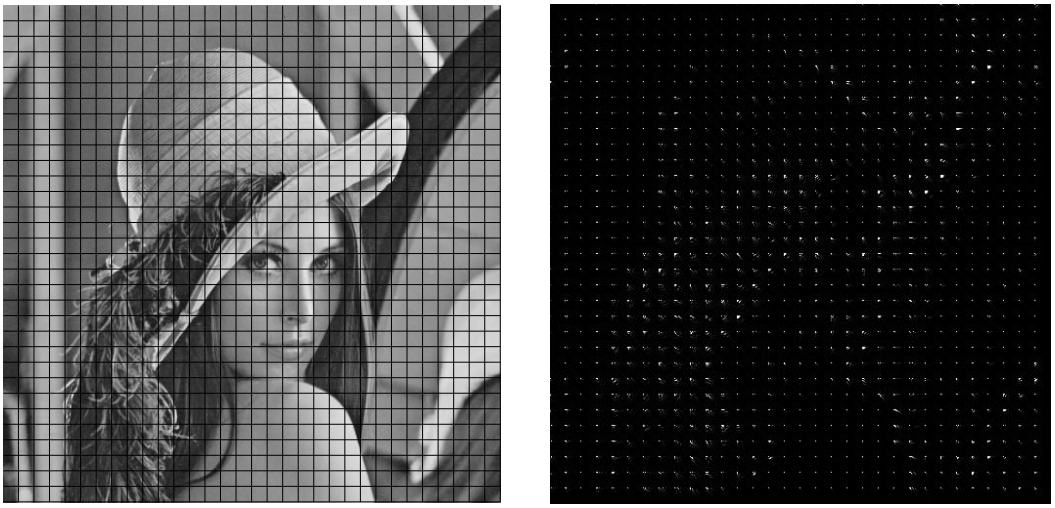
Contents

1 Image Compression

1.1 Image Segmentation and Transformation

In this assignment, we will write an algorithm to do image compression. This algorithm applies a general transform coding scheme that the image is transformed from spatial to frequency domain by 2D-DCT, and then experiences a quantization stage and finally be encoded. By doing so, the image information can be compressed to a large extent.

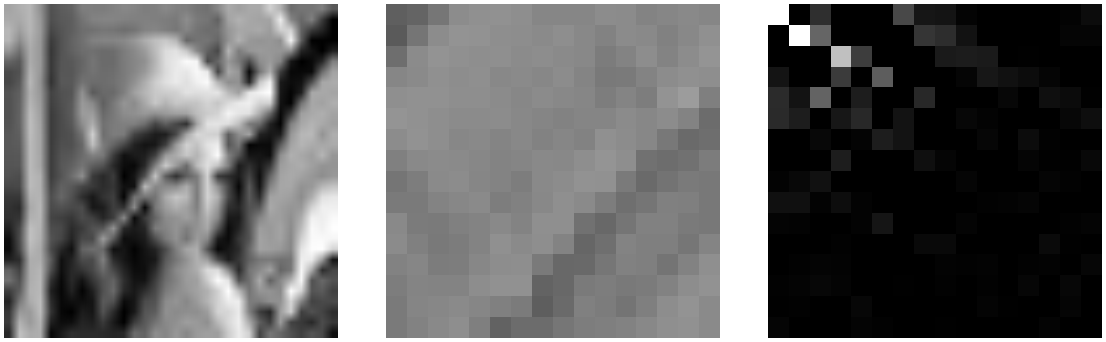
Theoretically, it is more efficient to compress information with low entropy because the distributions of data are focus on some range or say the datasets are unbalanced. For an image, the information in spacial domain can be large because every pixel is mapped to certain color directly and contributes to the image as a whole. However, after we transform it by DCT, the result usually contains less information. The reason is that there are large smooth areas in most images so low frequency components contributes most to the result or say few coefficients concentrate most of the signal energy. Therefore, it is easy to get a low-entropy symbol source in transformed domain.



(a) Segmented Image (b) DCT block by block

Figure 1: Image Segmentation and its DCT result ($N = 16$ on Lenna512)

In implementation, the first step is to segment the image to several $N \times N$ blocks. The 2D-DCT is not applied directly on the image but the little pieces separately. Here the matlab built-in function *blockporc* can be called to process specified size of blocks in an image. Then, 2D-DCT is applied on every $N \times N$ block. One example for this process is shown in Fig. 1. To have a more clear observation, Fig. 2 shows the block-level transformation. In Fig. 2(b), there are bright pixels in only up-left conner and most area are dark. It means that only DC and low frequency components have large coefficients in spectrum and high frequency parts have little effect on image. They can be removed



(a) DC values reordering (b) Single image block (c) Single block after DCT

Figure 2: Detail illustration on single block

in the quantization stage.

1.2 Quantization and Encoding

In this section, a quantization stage is implemented. For this task, the 2D-DCT result as shown in Fig. 2(b) is quantized by preset matrix, mainly to drive small coefficients in spectrum to zeros. Then, the result becomes easier to encode. The quantization matrix used in this task follows a pattern as in Matrix 1. There are 2^n number of 2^n in the first row, first column and diagonal lines. Other elements are just set in the range limited either by first row or first column. Notice that matrix values increase with their indexes which means the bottom-right values are larger than those in up-left conner. This is to set high frequency coefficients to zero after *round* operation. One example of quantization result is illustrated in M_{dct} where there are only few nonzero values in low frequency area.

$$Q_{Mat} = \begin{bmatrix} 1 & 2 & 2 & 4 & 4 & 4 & 4 & 8 \\ 2 & 2 & 2 & 4 & 4 & 4 & 4 & 8 \\ 2 & 2 & 2 & 4 & 4 & 4 & 4 & 8 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 8 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 8 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 8 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 8 \\ 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \end{bmatrix} \qquad M_{dct} = \begin{bmatrix} 14 & 6 & 0 & \cdots & 0 \\ 15 & -3 & 0 & \cdots & 0 \\ -2 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{1}$$

Then, we need to use factor S to rescale quantization matrix so we can control compression quality. S can be regarded as quantization step parameter and it is calculated by quality factor Q as shown in Eq. 3. Large Q will lead to high compression quality because the relevant quantization step S is small which means more levels can be expressed in the results. Meanwhile, in case of 0 in denominator of Eq. 2, the algorithm

has a validation step and then replaces all 0 values by 1. For this specific task, this will only happen when Q is 100 but that step can be applied not only in the relation of Eq.3.

$$M_{dct}(i,j) = round\left(\frac{M(i,j)}{S \times Q_{Mat}(i,j)}\right) \tag{2}$$

$$\begin{aligned} S &= \frac{100-Q}{50}, \quad Q > 50 \\ S &= \frac{50}{Q}, \quad Q \leqslant 50 \end{aligned} \tag{3}$$

After quantization, the matrix groups are combined together and directly encoded by the provided function *Entropy_enc*. Then, write the result to file by the function *fwrite*. By observation, all the encoded result are integers less than 256 write them them as *uint8* format. The whole compression process is summarized in Algorithm 1. The main difference between this compression method and JPEG is the encoding stage. In standard JPEG, the quantized DC values are coded by *DPCM* block by block. The AC coefficients are processed by *zig-zag reordering* and then entropy coding, which is more complex than the method in this task.

Algorithm 1 Image Compression

Input: im: Image to compress; N: Expected block size ($N \times N$); $Qmat$: Quantization matrix of size N; QP: Compression quality (integer $1 < QP < 100$);
Output: $im\_compress$: Compressed image;
1: Calculate scale factor S with QP
2: Obtain final quantization matrix $Q_S = Qmat \times S$;
3: Get im_dct by block-wise 2D-DCT on im;
4: Quantize im_dct by matrix Q_S and get im_dct_qtz;
5: Encode im_dct_qtz by function $entropy_enc$ and get $im\_compress$;
6: **return** $im\_compress$;

1.3 Image Decompression

Image decompression is a reverse process of compression.

1. Read from file to obtain entropy coding result.
2. Do entropy decoding on obtained data.
3. Recover 2D-DCT result by multiplying with quantization matrix.
4. Do inverse 2D-DCT to get the spatial image.

2 Results and Evaluation

This section is an evaluation on the implemented image compression method. The results are then compared with those done by Matlab built in JPEG method. Table 1 illustrates the performance under different mask sizes $N = 16$ and $N = 8$. For both mask sizes, rate and psnr values increase with quality factor indicating that higher image quality requires more data representation. Under the same quality factor, the rate of $N = 8$ is larger with a higher psnr compared to those by $N = 16$. However, the overall performance is better when $N = 16$ as shown in Fig. 3 where we can see the performance plots for both mask sizes. Mask of $N = 16$ always holds a higher *PSNR* under the same rate. Then, JPEG method is evaluated with the same sets of quality value but notice that its quality system is different from the implemented one so it is necessary to compare *PSNR* values of the same rate for evaluation. Three compressed images are chosen with similar rate are shown in Fig. 4. It can be seen that the performance of implemented compression method is close to JPEG by both intuitive observation and calculated data.

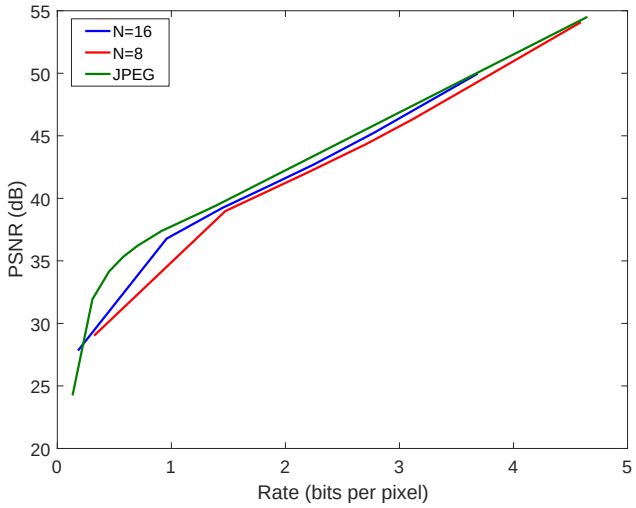


Figure 3: Rate-PSNR Comparison between $N = 16$ and $N = 8$

Table 1: Comparison on PSNR Results

Size	QP	1	15	29	43	57	71	85	99
16	rate (bpp)	0.18	0.96	1.45	1.87	2.25	2.78	3.69	7.57
	psnr (dB)	27.82	36.78	39.24	41.04	42.71	45.24	49.96	Inf
8	rate (bpp)	0.32	1.47	2.19	2.71	3.12	3.66	4.59	8.50
	psnr (dB)	29.02	38.98	42.05	44.334	46.32	49.19	54.09	Inf

Table 2: JEGP Compression Performance

JPEG-QP	1	15	29	43	57	71	85	99
rate (bpp)	0.13	0.31	0.45	0.58	0.70	0.91	1.39	4.65
psnr (dB)	24.24	31.93	34.15	35.32	36.22	37.39	39.39	54.51

(a) N=16, R=1.45, P=39.24 (b) N=8, R=1.47, P=38.98 (c) JPEG, R=1.39, P=39.39

Figure 4: Comparison of compression quality under similar rates

3 Trial on Enhancement

After implement the compression algorithm and evaluate the results, some trials are made to improve the performance. A zigzag scan is added to each block before encoded. This operation is mainly to gather zeros in the bottom right conner. As a result, most of the zeros will be placed at the end of stream. This is more efficient for encoding. However, the improvement is very limited. The bit rate only decreases by 0.002 bits per pixel for $N = 16$ and 0.02 for $N = 16$. The possible reason is that there are still a significant number of non-zero values in DCT matrix after quantization. It will prevent zigzag scan to achieve large run of zeros, making zigzag scan inefficient. The implemented algorithm with additional zigzag scan is attached in appendix.

Appendices

A Codes for Image Compression

Listing 1: base_mat_generation

```
function Qmat = base_mat_generation(N)
%-------------------------------------------------------
% This function is to generate quantization matirx
% according to provided rules
% Args:
%   N: size of suqare quantization matrix (N*N)
% Returns:
%   Qmat: generated qunatization matrix
%-------------------------------------------------------
Qmat = zeros([N,N]);
n = 0;
L = 0;
while L<N
    L = L+2^n;
    n = n+1;
end
n = n-1;
for i = 0:1:n
    low = 2^i;
    up = 2^(i+1)-1;
    if low > N
        low = N;
    end
    if up > N;
        up = N;
    end
    Qmat(low:up,:) = 2^i;
    Qmat(:,low:up) = 2^i;
end
```

Listing 2: compress_im

```
%-------------------------------------------------------
% This function is to do image compression
% Agrs:
%   im: input image to be compressed
%   Qmat: base quantization matrix
%   QP: compressiong quality factor
%   N: operation block size (Qmat size)
%   file_name: name of the file to store encoded image
% Returns:
%   rate: bit rate (bits per pixel)
%   imsize: Input image size
%   im_dct_qtz: matrix of image after dct and quantization
%-------------------------------------------------------

%% Quantization matrix generation
if QP > 50
    S = (100-QP)/50;
else
    S = 50/QP;
end
Qmat_s = S.*Qmat;
Qmat_s(Qmat_s==0) = 1;
%% Do DCT and quantization for each block
block_dct_qtz = @(block_struct) round(dct2(block_struct.data
   )./Qmat_s);
im_dct_qtz = blockproc(im, [N N], block_dct_qtz);
imsize = size(im);
%% Write file and evaluate
entropy_encode = entropy_enc(im_dct_qtz);
file = fopen(file_name,'w');
fwrite(file,entropy_encode,'uint8');
fclose(file);
rate = fsize(file_name)*8/(imsize(1)*imsize(2));
end
```

B Codes for Image Decompression

Listing 3: decompress_im

```
function [rate,imsize,im_dct_qtz] = compress_im(im, Qmat,QP,
    N, file_name)
%-------------------------------------------------------
% This function is to do image compression
% Agrs:
%   im: input image to be compressed
%   Qmat: base quantization matrix
%   QP: compressiong quality factor
%   N: operation block size (Qmat size)
%   file_name: name of the file to store encoded image
% Returns:
%   rate: bit rate (bits per pixel)
%   imsize: Input image size
%   im_dct_qtz: matrix of image after dct and quantization
%-------------------------------------------------------

%% Quantization matrix generation
if QP > 50
    S = (100-QP)/50;
else
    S = 50/QP;
end
Qmat_s = S.*Qmat;
Qmat_s(Qmat_s==0) = 1;
%% Do DCT and quantization for each block
block_dct_qtz = @(block_struct) round(dct2(block_struct.data
   )./Qmat_s);
im_dct_qtz = blockproc(im, [N N], block_dct_qtz);
imsize = size(im);
%% Write file and evaluate
entropy_encode = entropy_enc(im_dct_qtz);
file = fopen(file_name,'w');
fwrite(file,entropy_encode,'uint8');
fclose(file);
rate = fsize(file_name)*8/(imsize(1)*imsize(2));
end
```

C Image Compression with Zigzag Scan

Listing 4: zigzag

```
function zigzag_out = zigzag(mat_in)
%-------------------------------------------------------
% This function is to do zigzag scan on matrix
% Args:
%   mat_in: input matrix to do zigzag on
% Returns:
%   zigzag_out: array after zigzag scan
%-------------------------------------------------------
h = 1;
v = 1;
vmin = 1;
hmin = 1;
vmax = size(mat_in, 1);
hmax = size(mat_in, 2);
i = 1;
zigzag_out = zeros(1, vmax * hmax);
while ((v <= vmax) && (h <= hmax))
    if (mod(h + v, 2) == 0)
        if (v == vmin)
            zigzag_out(i) = mat_in(v, h);
            if (h == hmax)
                v = v + 1;
            else
                h = h + 1;
            end;
            i = i + 1;
        elseif ((h == hmax) && (v < vmax))
            zigzag_out(i) = mat_in(v, h);
            v = v + 1;
            i = i + 1;
        elseif ((v > vmin) && (h < hmax))
            zigzag_out(i) = mat_in(v, h);
            v = v - 1;
            h = h + 1;
            i = i + 1;
        end;
    else
        if ((v == vmax) && (h <= hmax))
            zigzag_out(i) = mat_in(v, h);
            h = h + 1;
```

```
            i = i + 1;
        elseif (h == hmin)
            zigzag_out(i) = mat_in(v, h);
            if (v == vmax)
                h = h + 1;
            else
                v = v + 1;
            end;
            i = i + 1;
        elseif ((v < vmax) && (h > hmin))
            zigzag_out(i) = mat_in(v, h);
            v = v + 1;
            h = h - 1;
            i = i + 1;
        end;
    end;
    if ((v == vmax) && (h == hmax))
        zigzag_out(i) = mat_in(v, h);
        break
    end;
end;
```

Listing 5: Inverse Zigzag

```
function mat_out = izigzag(mat_in, vmax, hmax)
%-------------------------------------------------------
% This function is to do inverse zigzag scan to
% recover the original matrix before zigzag operation
% Args:
%   mat_in: input matrix to recovered
%   vmax: expected output row
%   hmax: expected ouput column
% Returns:
%   mat_out: recovered matrix
%-------------------------------------------------------
h = 1;
v = 1;
vmin = 1;
hmin = 1;
mat_out = zeros(vmax, hmax);
i = 1;
while ((v <= vmax) && (h <= hmax))
    if (mod(h + v, 2) == 0)
        if (v == vmin)
```

```
            mat_out(v, h) = mat_in(i);
            if (h == hmax)
                v = v + 1;
            else
                h = h + 1;
            end;
            i = i + 1;
        elseif ((h == hmax) && (v < vmax))
            mat_out(v, h) = mat_in(i);
            v = v + 1;
            i = i + 1;
        elseif ((v > vmin) && (h < hmax))
            mat_out(v, h) = mat_in(i);
            v = v - 1;
            h = h + 1;
            i = i + 1;
        end;
    else
        if ((v == vmax) && (h <= hmax))
            mat_out(v, h) = mat_in(i);
            h = h + 1;
            i = i + 1;
        elseif (h == hmin)
            mat_out(v, h) = mat_in(i);
            if (v == vmax)
                h = h + 1;
            else
                v = v + 1;
            end;
            i = i + 1;
        elseif ((v < vmax) && (h > hmin))
            mat_out(v, h) = mat_in(i);
            v = v + 1;
            h = h - 1;
            i = i + 1;
        end;
    end;
    if ((v == vmax) && (h == hmax))
        mat_out(v, h) = mat_in(i);
        break
    end;
end;
```