컴퓨터 비젼

[Term Project - JPEG Compression]

학번	201203393
분반	00
이름	김헌겸
과제번호	Term

제출일 : 2016년 6월 22일 수요일

1. DCT 및 IDCT 구현

▼ Algorithms

The discrete cosine transform (DCT) is closely related to the discrete Fourier transform. It is a separable linear transformation; that is, the two-dimensional transform is equivalent to a one-dimensional DCT performed along a single dimension followed by a one-dimensional DCT in the other dimension. The definition of the two-dimensional DCT for an input image A and output image B is

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le p \le M-1 \\ 0 \le q \le N-1 \end{matrix}$$

where

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}}, & p = 0 \\ \sqrt{\frac{2}{M}}, & 1 \le p \le M-1 \end{cases}$$

and

$$\alpha_q = \begin{cases} \frac{1}{\sqrt{N}}, & q = 0 \\ \sqrt{\frac{2}{N}}, & 1 \le q \le N-1 \end{cases}$$

M and *N* are the row and column size of A, respectively. If you apply the DCT to real data, the result is also real. The DCT tends to concentrate information, making it useful for image compression applications.

This transform can be inverted using `idct2`.

Mathworks의 Document에서 DCT에 관련한 알고리즘을 찾아보면 위와 같다.

```
function result = DCT_1D(im)

im_size = size(im);
result = zeros(im_size);

for u=0 : im_size(2)-1
    sum = 0;
    for x=0 : im_size(2)-1
        sum = sum + (cos((((2 * x) + 1) * u * pi)/(im_size(2)*2)) * im(1,x+1));
    end

    if u == 0
        C = (1 / sqrt(im_size(2)));
    else
        C = sqrt(2/im_size(2));
    end

    result(1,u+1) = C * sum;
end
```

```
function result = DCT_2D(im)

im_size = size(im);
result = zeros(im_size(1), im_size(2));

for i=1:im_size(2)
    temp = im(:,i);
    result(:,i) = DCT_1D(temp');
end

for i=1:im_size(1)
    temp = result(i,:);
    result(i,:) = DCT_1D(temp);
end
```

이전 실습에서 작성했던 DFT를 참고하여 수식만 바꾸어 DCT를 구현하였다. 그리고 separable하게 사용하기 위해서 1D와 2D를 나누어 구현하였다.

▼ Algorithms

idct2 computes the two-dimensional inverse DCT using:

$$A_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q B_{pq} \cos \frac{\pi(2m+1)p}{2M} \cos \frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le m \le M-1 \\ 0 \le n \le N-1 \end{matrix},$$

where

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}}, & p = 0 \\ \sqrt{\frac{2}{M}}, & 1 \le p \le M-1 \end{cases}$$

and

$$\alpha_q = \begin{cases} \frac{1}{\sqrt{N}}, & q = 0 \\ \sqrt{\frac{2}{N}}, & 1 \le q \le N-1 \end{cases}.$$

Inverse DCT 또한 위와 같은 수식을 가지며 DCT와 약간의 차이만 있을 뿐이다.

```
function result = IDCT_1D(im)

im_size = size(im);
result = zeros(im_size);

for u=0 : im_size(2)-1
     sum = 0;

    for x=0 : im_size(2)-1

        if x == 0
            C = (1 / sqrt(im_size(2)));
        else
            C = sqrt(2 / im_size(2));
        end

        sum = sum + ((cos((((2 * u) + 1) * x * pi)/(im_size(2)*2)) * im(1,x+1) * C));
    end
    result(1,u+1) = sum;
end
```

```
function result = IDCT_2D(im)

im_size = size(im);
result = zeros(im_size(1), im_size(2));

for i=1:im_size(2)
     temp = im(:,i);
     result(:,i) = IDCT_1D(temp');
end

for i=1:im_size(1)
     temp = result(i,:);
     result(i,:) = IDCT_1D(temp);
end
```

IDCT도 역시 동일한 방법으로 구현 할 수 있다.

2. zigzag Coding

zigzag의 경우 오픈소스로 제공되고 있는 소스를 사용하였다.

```
function output = zigZag(im)
%Zigzag scanning
%By Ketul Shah and Sagar Shah
%Nirma University
t=0;
last_index = 0;
l=size(im);
sum=l(2)*l(1);  %calculating the M*N
new = zeros(1,(l(1) * l(2) + 1));
for d=2:sum
    c=rem(d,2);  %checking whether even or odd
    for i=1:l(1)
        for j=1:l(2)
            if((i+j)==d)
                t=t+1;
                if(c==0)
                    new(t)=im(j,d-j);
                    if(im(j,d-j) ~= 0)
                        %last_index = t+1;
                        last_index = t;
                    end
                else
                    new(t)=im(d-j,j);
                    if(im(d-j,j) ~= 0)
                        %last_index = t+1;
                        last_index = t;
                    end
                end
            end
        end
    end
end
%new(1, last_index+1) = eob.EOB;
%output = new(1, 1:last_index+1);
output = new;
```

```
function out = izigZag(input)
%Zigzag scanning
%By Ketul Shah and Sagar Shah
%Nirma University
im = zeros(8,8);
t=0;
l=size(im);
sum=l(2)*l(1);  %calculating the M*N
for d=2:sum
 c=rem(d,2);  %checking whether even or odd
    for i=1:l(1)
        for j=1:l(2)
            if((i+j)==d)
                t=t+1;
                if(c==0)
                    im(j,d-j) = input(t);
                else
                    im(d-j,j) = input(t);
                end
            end
        end
    end
end
out = im;
end
```

간단한 알고리즘을 통하여 1차원 매트릭스로 변환 할 수 있다.

3. Huffman Encoding

JPEG 압축을 하면서 가장 중요하다고 할 수 있는 Huffman Encoding과 Decoding에 대한 부분이다. 이전 과제에서 사용을 했었지만 이번에 전체 이미지를 블록 단위로 자르고, 각각에 대해서 Encoding 및 Decoding 하는 과정에 시간 소요가 크고, encoding 할 값들이 정확한 화소 값을 지칭하지 않으므로 약간 변형된 Huffman table을 만들어 연산속도를 높였다.

Generate Huffman Table

```
function output = huffman_table(im)

[x,y] = imhist(im);
x_size = size(x);
im_size = size(im);
intensity = x./(im_size(1) * im_size(2));

sorted = sort(intensity, 'descend');
arr_size = 0;
for i = 1 : x_size(1)
    if sorted(i,1) == 0
        break;
    end
    arr_size = arr_size + 1;
end

huffarray = dlnode();

arr_count = 1;
arr_set = [];
for i=1 : im_size(2)
    isDuplicated = 0;
    for j = 1 : length(arr_set)
        if arr_set(j) == (im(1,i)+1);
            isDuplicated = 1;
            break;
        end
    end
    if isDuplicated == 0
        huffarray(arr_count) = dlnode(intensity(im(1,i)+1), im(1,i));
        arr_set(arr_count) = (im(1,i)+1);

        arr_count = arr_count + 1;
    end
end

table = cell(1,arr_size);

if length(im) == 0
    output = table;
elseif length(im) == 1
    table{1} = {im+1, '1'};
    output = table;
else
```

사진을 보면 등장하는 값들에 대해서만 table을 만들고, 각각을 cell array로 만들어 index 값 및 encode 될 허프만 코드를 저장할 수 있도록 준비하고 있음을 알 수 있다.

```
huffarray = sort_all_dlnode(huffarray);

huffarray = build_huffman_tree(huffarray);

huffman_root = huffarray(1);

global table_size;
table_size = 1;

table = build_huffman_table(huffman_root, table, '11');

output = table;
end
end

function output = delete_zero(huffarray)
    len = length(huffarray);
    index = 0;
    for i=1 : len
        if huffarray(i).Data == 0
            index = i;
        else
            if index >= 1
                huffarray(1:index) = [];
            end
            break;
        end
    end
    output = huffarray;
end

function output = build_huffman_tree(huffarray)
    while length(huffarray) >= 2
        first = huffarray(1);
        second = huffarray(2);
        newnode = dlnode(first.Data + second.Data, -1);
        if(first.Data > second.Data)
            insertLeft(newnode, first);
            insertRight(newnode, second);
        else
            insertLeft(newnode, second);
            insertRight(newnode, first);
        end
```

```
        huffarray(1) = [];
        huffarray(1) = [];
        huffarray(length(huffarray)+1) = newnode;

        huffarray = sort_other_node(huffarray);
        output = huffarray;
    end
    output = huffarray;
end

function output = build_huffman_table(huffnode, table, code)
    global table_size;
    if huffnode.Pixel ~= -1
        table{table_size} = {huffnode.Pixel+1, code};
        table_size = table_size+1;
    else
        if ~isempty(huffnode.left)
            table = build_huffman_table(huffnode.left, table, strcat(code, '0'));
        end
        if ~isempty(huffnode.right)
            table = build_huffman_table(huffnode.right, table, strcat(code, '1'));
        end
    end
    output = table;
end

function output = sort_other_node(huffarray)
    len = length(huffarray);
    i = len;
    while i ~= 1
        if huffarray(i-1).Data < huffarray(len).Data
            break;
        end
        i = i - 1;
    end

    temp = huffarray(len);
    huffarray(i+1:len) = huffarray(i:len-1);
    huffarray(i) = temp;
    output = huffarray;
end
```

```
function output = sort_all_dlnode(huffarray)
    len = length(huffarray);
    for m=1 : len
        index = m;
        for k=m : len-1
            if huffarray(index).Data > huffarray(k+1).Data
                index = k+1;
            end
        end
        temp = huffarray(m);
        huffarray(m) = huffarray(index);
        huffarray(index) = temp;
    end
    output = huffarray;


end
```

그 다음부터는 이전과제와 동일하다. 허프만 트리를 우선적으로 구성하고 해당 트리를 통해 허프만 테이블을 구성하면 된다.

Huffman Encoding

```
function output = huffman_encoding(im, huffman_table)
    encode_data = '';
    im_size = size(im);
   % index = 2;
    output = zeros(1, im_size(1));

   % output(1) = im_size(1);
    index = 1;
    for i=1 : im_size(1)
        for j=1 : im_size(2)
            if im(i,j) == 255
                encode_data = strcat(encode_data, searchInHuffmanTable(huffman_table, 256));
            else
                encode_data = strcat(encode_data, searchInHuffmanTable(huffman_table, im(i,j)+1));
            end

            while length(encode_data) > 8
                output(index) = uint8(bin2dec(encode_data(1:8)));
                index = index + 1;
                encode_data(1:8) = [];
            end
        end
    end

    if ~isempty(encode_data)
        output(index) = uint8(bin2dec(encode_data(1:length(encode_data))));
        output(index+1) = uint8(length(encode_data));
    end
end

function output = searchInHuffmanTable(table, index)
    table_size = size(table);
    for x = 1 : table_size(2)
        record = table{x};
        if record{1} == index
            output = char(record{2});
            break;
        end
    end
end
```

이전에 만든 테이블을 이용해서 bitstream의 형태로 encoding한다. 이때, 이전과는 다른 테이블 탐색 방법을 적용한다. 이전에는 pixel들에 대해 code를 유지했는데, 이제는 존재하는 값에 대해서만 테이블이 존재하므로 모든 테이블을 일일이 뒤져봐서 해당하는 code를 찾아온다. 이렇게 만든 코드를 모두 결합하여 bitstream을 만들고 반환한다.

Huffman Decoding

```
function output = huffman_decoding(huffman_encoded_data, table)
    output = zeros(1, length(huffman_encoded_data)-1);
    decode_data = '';
    min_length = 1;
    index = 1;
    while length(huffman_encoded_data) > 2
        decode_data = strcat(decode_data, dec2bin(huffman_encoded_data(1), 8));
        huffman_encoded_data(1) = [];

        bit_place = min_length;
        while length(decode_data) >= bit_place
            table_found = 0;
            table_size = size(table);
            for j=1 : table_size(2)
                record = table{j};
                if strcmp(record{2}, decode_data(1:bit_place))
                    output(index) = uint8(record{1}-1); index = index + 1;
                    decode_data(1:bit_place) = [];
                    bit_place = min_length; table_found = 1;
                    break;
                end
            end
            if ~table_found
                bit_place = bit_place + 1;
            end
        end
    end
    if ~isempty(huffman_encoded_data)
        decode_data = strcat(decode_data, dec2bin(huffman_encoded_data(1), huffman_encoded_data(2)));
    end
    bit_place = min_length;
    while length(decode_data) >= bit_place
        table_found = 0;
        for j=1 : length(table)
            if strcmp(table(j), decode_data(1 : bit_place))
                output(index) = uint8(j-1); index = index + 1;
                decode_data(1:bit_place) = [];
                bit_place = min_length; table_found = 1;
                break;
            end
        end
        if ~table_found
            bit_place = bit_place + 1;
        end
    end
```

주어진 bitstream과 허프만 테이블 정보를 통해 코드를 decoding한다. 비트는 한 개씩 차례대로 읽고, 해당 비트들을 가지고 테이블에서 찾아 원본 값으로 돌려준다. 이 때, 새로 적용한 테이블이 유용하게 쓰이는데, JPEG을 할 때, block에서 대부분의 값이 0이 되므로 쓸모없는 데이터들이 많게 되고, 그만큼 불필요한 table은 유지할 필요가 없다. 그러므로 이러한 테이블 방식을 사용하는 것이 더 유용할 것이다.

4. JPEG Encoding

```
function [result, table_list, im_x, im_y] = jpeg_in(im, N, Q)

im = double(im) - 128;

im_size = size(im);

im_x = ceil(im_size(1)/8);
im_y = ceil(im_size(2)/8);

im_temp = zeros(im_x*8, im_y*8);
im_temp(1:im_size(1),1:im_size(2)) = im;

result = zeros(1,0);
table_list = cell((im_x)*(im_y),2);
table_size = 1;

for i = 1 : 8 : im_size(1)
    for j = 1 : 8 : im_size(2)
        block = im_temp(i:i+7,j:j+7);
        dct_block = DCT_2D(block);
        dct_Q_block = round(dct_block./(N*Q));
        zig = zigZag(dct_Q_block);
        zig = uint8(zig + 128);
        table = huffman_table(zig(1,:));

        encode_data = huffman_encoding(zig(1,:),table);
        table_list{table_size,1} = table;
        table_list{table_size,2} = length(encode_data);
        table_size = table_size+1;

        result = cat(2, result, encode_data);
    end
end

im_x = im_x*8;
im_y = im_y*8;
```

encoding 순서에 맞추어 각 8x8 블록에 적용한다.

1. Block을 구분한다	block = im_temp(i : i+7, j : j+7)
2. DCT_2D를 적용한다	dct_block = DCT_2D(block)
3. DCT 블록에 N*Q로 nomalization을 한다.	dct_Q_block= round(dct_block./(N*Q))
4. 결과 값을 zigzag 코딩한다.	zig = zigZag(dct_Q_block)
5. 마지막으로 Huffman Coding을 한다.	

5. JPEG Decoding

```
function result = jpeg_out(stream, table, N, Q, x, y)

result = zeros(x,y);
table_count = 1;


for i = 1 : 8 : x
    for j = 1 : 8 : y
        cur_table = table{table_count,1};
        length = table{table_count,2};
        table_count = table_count + 1;
        data = stream(1:length);
        stream(1:length) = [];
        decode_data = huffman_decoding(data, cur_table);
        decode_data = double(decode_data - 128);
        pad = zeros(1,64);
        s = size(decode_data);
        pad(1:s(2)) = decode_data;
        zig = izigZag(pad);
        zig_Q_block = zig.*(N*Q);
        zig_IDCT = IDCT_2D(zig_Q_block);
        zig_IDCT = round(zig_IDCT + 128);
        result(i:i+7, j:j+7) = uint8(zig_IDCT);
    end
end

result = uint8(result);
```

Encoding 했던 방식을 역으로 되돌린다.

1. 주어진 table을 통해 Huffman Decoding을 하여 값 배열을 얻어온다.	
2. Decoding 결과 값에 padding을 하고 izigzag 코딩을 한다.	zig = izigZag(pad)
3. 블록에 N*Q를 곱해 denomalization을 한다.	zig_Q_block= zig.*(N*Q)
4. IDCT_2D를 적용한다	zig_IDCT = IDCT_2D(zig_Q_block)
5. 각 block에 대해 결과 값으로 합쳐준다.	result(i : I+7, j : j+7) = uint8(zig_IDCT)

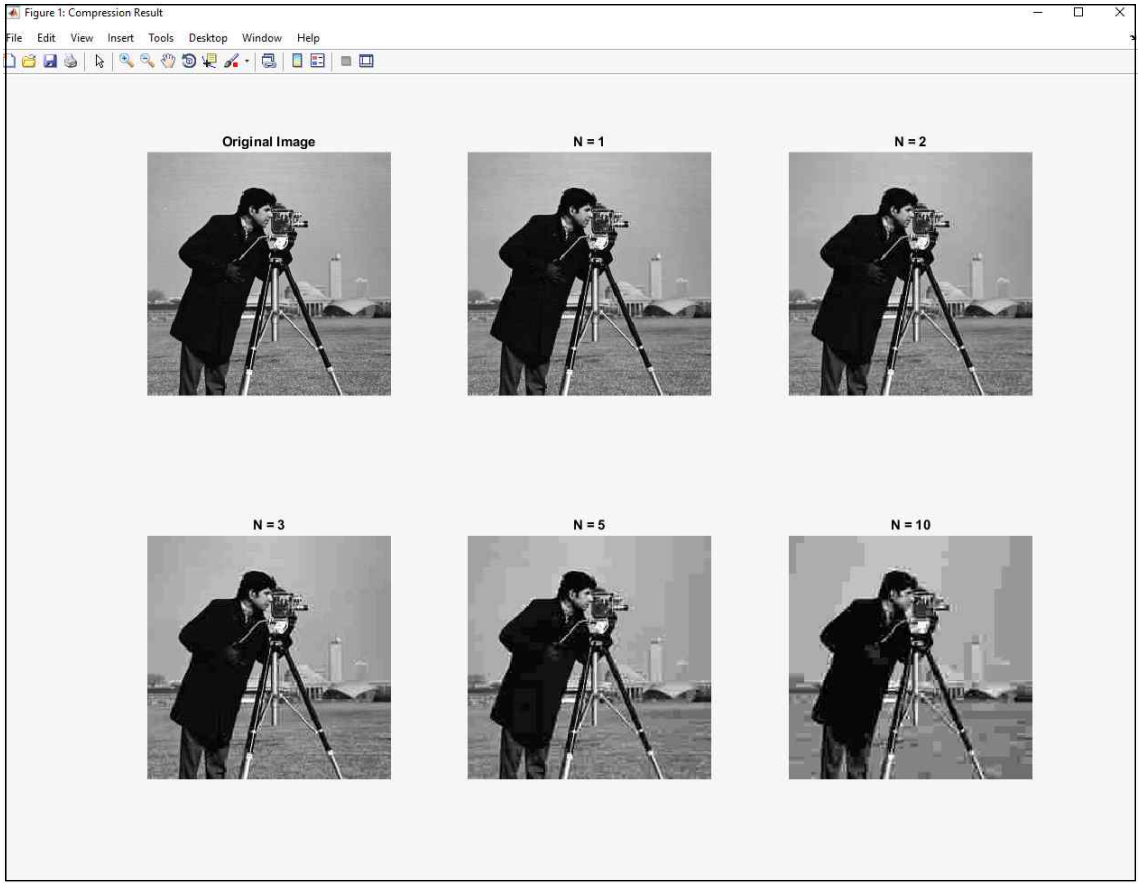
이렇게 하면 JPEG Decoding을 할 수 있다.

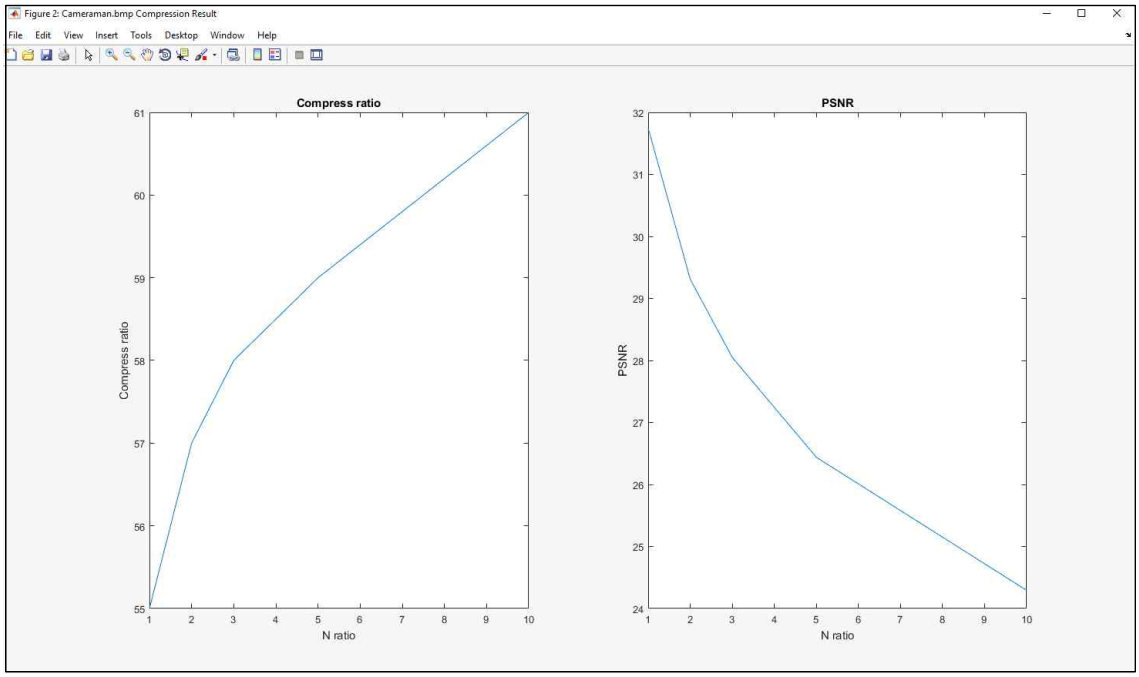
6. Result

```
function term()

Q = [16,11,10,16,24,40,51,61;12,12,14,19,26,58,60,55;
    14,13,16,24,40,57,69,56;14,17,22,29,51,87,80,62;
    18,22,37,56,68,109,103,77;24,35,55,64,81,104,113,92;
    49,64,78,87,103,121,120,101;72,92,95,98,112,100,103,99];

im = imread('Cameraman.bmp');
%im = imread('Lena.bmp');
%im = imread('caribou.tif');

im_size = size(im);

N = [1 2 3 5 10];
PSNR_value = zeros(1,5);
compress_value = zeros(1,5);
out_result = cell(1,5);

total_size = im_size(1) * im_size(2);

for i = 1 : 5
    [in, table, im_x, im_y] = jpeg_in(im, N(i), Q);
    out = jpeg_out(in, table, N(i), Q, im_x, im_y);
    out_result{1,i} = out;

    comp_size = size(in);
    comp_size = comp_size(2);

    compress_value(1,i) = round(100 - ((comp_size/total_size) * 100));
    PSNR_value(1,i) = psnr(im, out);
end

figure('Name', 'Compression Result'),
subplot(2,3,1), imshow(uint8(im)), title('Original Image')
subplot(2,3,2), imshow(uint8(out_result{1,1})), title('N = 1')
subplot(2,3,3), imshow(uint8(out_result{1,2})), title('N = 2')
subplot(2,3,4), imshow(uint8(out_result{1,3})), title('N = 3')
subplot(2,3,5), imshow(uint8(out_result{1,4})), title('N = 5')
subplot(2,3,6), imshow(uint8(out_result{1,5})), title('N = 10')

figure('Name', 'Cameraman.bmp Compression Result'),
subplot(1,2,1), plot( N, compress_value),
title('Compress ratio'), xlabel('N ratio'), ylabel('Compress ratio')
subplot(1,2,2), plot( N, PSNR_value),
title('PSNR'), xlabel('N ratio'), ylabel('PSNR')
```

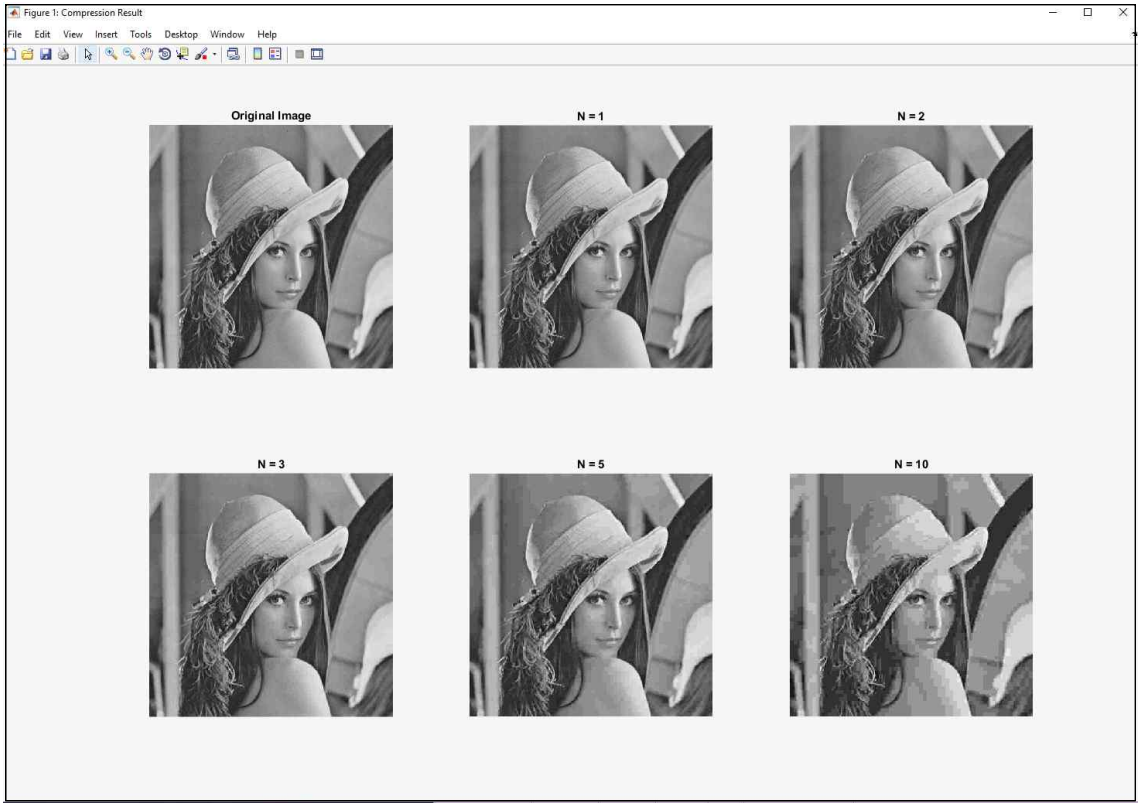
주어진 사진에 대해서 N = 1,2,3,5,10 각각으로 encoding과 decoding을 하고, 압축률과 PSNR 값을 계산하여 그래프로 출력해본다.

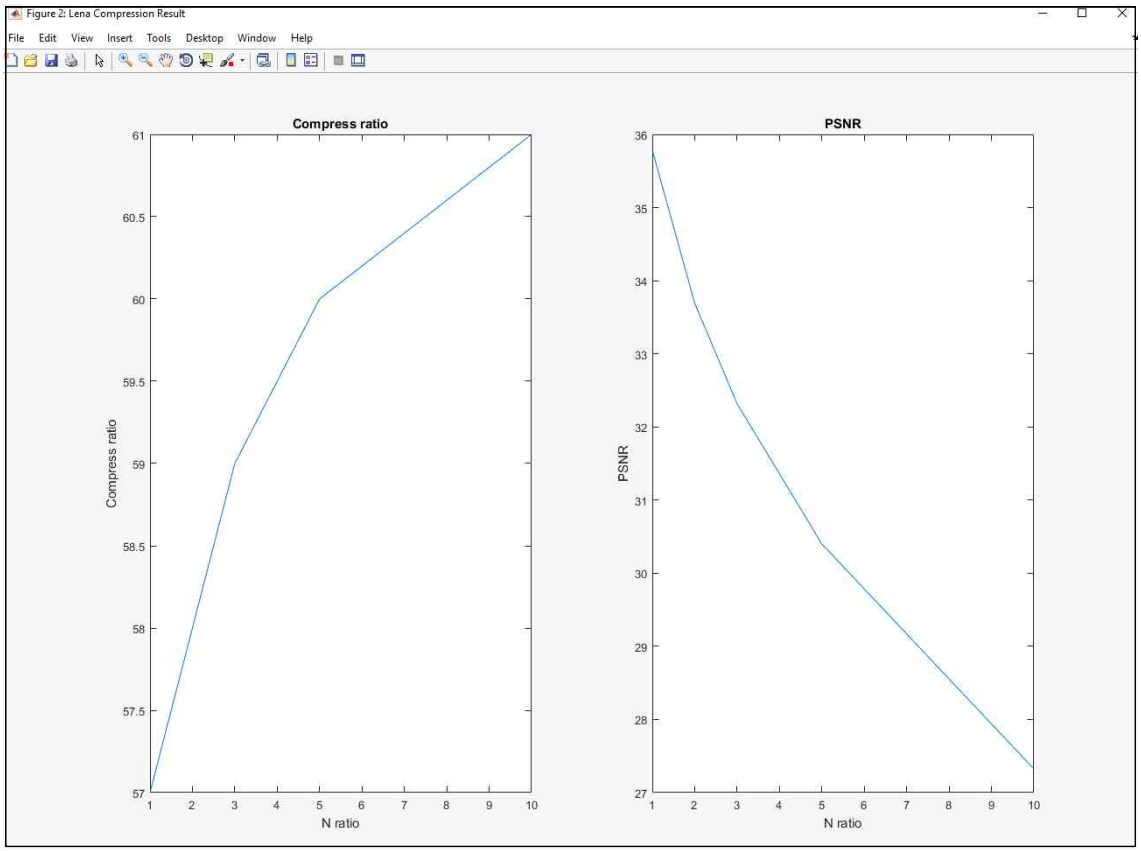
- Compress Cameraman.bmp -





- Compress Lena.bmp -





결과 값을 보면 N의 값을 증가시킬수록 압축률은 증가하지만 PSNR값은 감소하는 것을 보아 화질은 떨어지고 있음을 알 수 있다.

7. Review

JPEG 과제가 역대 급으로 어려웠다라고 자신 있게 말 할 수 있을 것 같다. 알고리즘을 적용하는데에 있어서 zigzag 코딩 하는 과정에서 생기는 데이터 제로 현상 (결과를 뽑아냈을 때, 데이터가 그냥 없어져서 검은 블록으로만 나타났다), huffman coding 중 음수 값을 coding 하는 경우 처리, DC coefficient를 따로 코딩하는 과정에서 생겼던 문제들, 알고리즘을 완성한 뒤에 실행해보니 터무니없이 느린 decoding 속도 등, 꽤 많은 문제들이 발생하였고, huffman coding을 적용하여 활용하는 것 자체에서 큰 문제들이 발생하다보니 디버그와 알고리즘 분석에 심혈을 기울일 수밖에 없었다. 지금 돌아보면 사소한 실수들이 시간을 질질 끌게 만들었지만, 그래도 그 과정에서 cell array를 적극 활용하여 데이터의 구조도 제대로 바꾸어 보고, JPEG header 파일을 만들어 보려는 시도에서 꽤 많은 공부도 할 수 있었으며 무엇보다 손으로 구현해서 완성해냈다는 뿌듯함이 많이 남는다. 헤더조차 없으니 완벽한 JPEG Encoding은 아니지만 이를 통해 꽤 많은 배움을 얻어 갈 수 있었다고 생각한다.