Outline

Contents

1 Image Compression

Image (or video) compression is a process by which we take the raw bits of picture data and reduce the number of bits while minimizing any "perceivable" artifacts. This is only possible if there is some redundancy in the data. This redundancy can take various forms, as we will discuss in the following. Before getting there, we will need to review some basic information theory concepts that allows us to quantify what we mean by information bits.

The flow of this lecture is as follows: after a discussion on information quantification in terms of entropy–which allows us to measure how many "bits" are needed to convey a certain "information", we will then introduce the noiseless coding theorem (thanks to Shannon). This tells us precisely the lower bound on the number of bits required to transmit a given data. However, the theorem itself does not say how you can get achieve that lower bound. Then we will discuss a simple coding strategy, called Huffman coding, that gives a prescription to have an optimal code – a code that works with minimum number of bits per symbol – if each symbol is coded one at a time. This code length will always be greater than or equal to what the coding theorem states.

The noiseless – or also known as error-free– coding strategies will not give you much compression when it comes to images. There are, however, other redundancies one can exploit here, given that much of this data is consumed by humans. This leads us to the lossy compression strategies, lossyness mostly due to quantization of the values. You have already experimented with this in your KLT/DFT homeworks.

The last part of this would be a review of the standard, basic JPEG compression. That will be your final homework for the class.

1.1 Basics

Basics

- Data Redundancy
- Self-information and Entropy
- Error-free and lossy compression
- Huffman coding
- Predictive coding
- Transform coding

Lossy vs Loss-less compression

Loss-less (information preserving)
Images can be compressed and restored without any loss. Many medical imaging and GIS (geographic information systems) require that data is compressed without loss. You dont get much compression out of this, typially a factor of 2-3 is considered significant.

Lossy compression
In this case perfect recovery is not possible; however, you get a significant compression factor (10-100). Example applications include almost all consumer entertainment (movies, DVDs), teleconferencing, etc.

Data Redundancy

To get compression, there must be some redundancy in the data. In image processing, we consider three broad types of redundancy:

- **Coding**: exploit the fact that fewer bits to represent frequent symbols is a good strategy
- **Interpixel/Interframe**: neighboring pixels in an image or neighboring frames in a video are highly correlated – have similar values.
- **Psychovisual**: human visual system can not simultaneously distinguish all colors/gray levels.

Coding redundancy

- A reasonable strategy is to assign fewer bits to frequently occurring symbols. Let r_k denote the k-th gray level in an image, $k = 0, 1, \cdots, L-1$ where L is the total number of gray levels.
- Let the discrete probability of the intensity level r_k be $p_r(r_k) = \frac{n_k}{n}$, where n_k is the number of pixels with intensity r_k, and n is the total number of pixels in the image.

- Let l_k be the number of bits used to represent r_k. Hence the average number of bits required to represent each pixel in the image is

$$L_{\text{avg}} = \sum_{k=0}^{L-1} l(r_k) p_r(r_k)$$

- If $l(r_k) = mbits = constant$, then $L_{\text{avg}} = m$.

From above discussion, it makes sense to assign fewer bits to those r_k for which $p_r(r_k)$ are large so that the average L is reduced:

$$L_{\text{avg}} = \sum_{k=0}^{L-1} l(r_k) p_r(r_k)$$

This achieves compression and results in a variable length code.

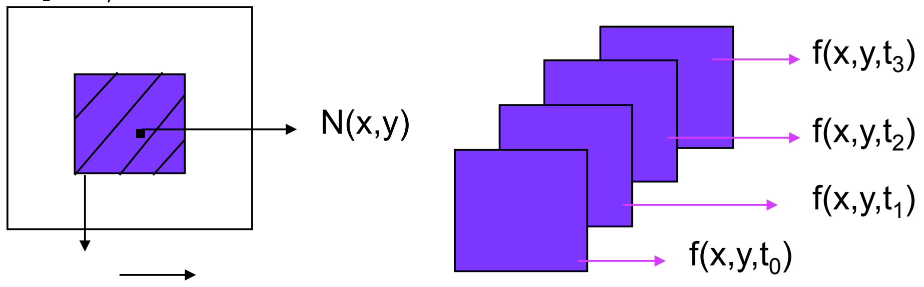
bottom line: More probable gray levels will have fewer number of bits.

An example of a variable length code

r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
$r_{23} = 23$	0.25	01010111	8	01	2
$r_{100} = 100$	0.5	10000000	8	1	1
$r_{123} = 123$	0.25	11000100	8	000	3
$r_{255} = 255$	0.05	11111111	8	001	3
r_k for $k \neq 23, 100, 123, 255$	0	-	8	-	0

Table 1: Variable length code

Inter-pixel/Inter-frame



(**left**):$f(x, y)$ at location (x, y) in an image depends on a local neighborhood $N(x, y)$ of pixels. (**right**) A time-sequence of temporal frames are related to each other, and this dependency is exploited for video compression.

Psychovisual

Human visual system has limitations! good example is quantization of a image/video signal: convey the necessary information but using significantly fewer bits.

Think of all the web images, streaming videos, facebook and other online multimedia data: they all work because you are able to compress the data and still dont feel the effect of this compression.

Question

How do we take advantage of these redundancies? What is a typical compression workflow?

1.2 A General framework for compression

Image Compression: A Schematic

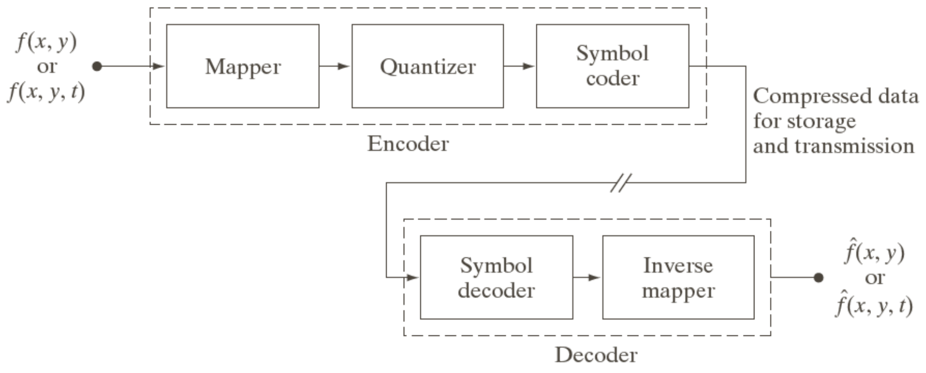

$f(x, y)$ or $f(x, y, t)$
Mapper
Quantizer
Symbol coder
Encoder
Compressed data for storage and transmission
Symbol decoder
Inverse mapper
Decoder
$\hat{f}(x, y)$ or $\hat{f}(x, y, t)$

Figure: courtesy Gonzalez and Woods Book.

Source Encoder

Mapper

Mapper is designed to reduce interpixel redundancy. Examples include

- run-length encoder
- transform to another domain where the coefficients are less correlated than the original (e.g., DCT, Fourier, Wavelets)

Quantizer

Quantizer reduces psychovisual redundancies in the image. This block should be left out if err-free encoding is desired.

Symbol Encoder

This creates a fixed/variable length code that reduces coding redundancies.

Source Decoder

*Quantization is **NOT** reversible*

You follow an inverse sequence of operations. First with the symbol decoder followed by the inverse mapper.

Questions

- Is there a minimum amount of data that is sufficient to completely describe an image without any loss of information?
- How do you measure information?

Aside: Self-information and Entropy of a signal

- Suppose an event E occurs with probability $P(E)$. Then it is said to contain $I(E) = -\log P(E)$ units of information.
- $P(e) = 1$ (always happens) $\implies I(e) = 0$ (conveys no information).
- If the base of the logarithm is 2, then the unit of information is a bit.
- If $P(e) = 1/2$, $I(e) = -\log_2(1/2) = 1\text{bit}$.
  - e.g., flipping of a coin. Output of this experiment requires 1 bit to convey the information.

Entropy of a source

Assume an information source generating L symbols $\{a_1, a_2, \cdots, a_L\}$ with probability $p(a_i)$; $\sum_i p(a_i) = 1$. Then,

$$I(a_i) = -\log_2 p(a_i)\text{bits}$$

The average information per source output, H. is then defined as its Entropy.

$$H = -\sum_{i=0}^{L} p(a_i)\log_2 p(a_i) \quad \text{bits/symbol} \tag{1}$$

H is also referred to as the uncertainty of the source.

Prove: If all the source symbols are equally probable, then the source has maximum entropy.

Noiseless Coding Theorem

(**Shannon**) It is possible to code, without any loss of information, a source signal with entropy H bits/symbol, using $H + \epsilon$ bits/symbol where ϵ is an arbitray small quantity.

ϵ can be made arbitrarily small by considering increasingly larger blocks of symbols to be coded.

Histograms and Image Entropy

Consider a grey scale image with intensity values in [0, 255]. The histogram of this image gives you a discrete distribution of pixel values. The normalized values could then be used to approximate the probability distribution. The entropy of the image, which is the minimum number of bits required to code the image without any error, can then be calculated using these probabilities.

Lower entropy means fewer bits to convey the information.

1.3 Huffman Codes

Error-free coding: Huffman codes

Huffman coding yields the smallest possible number of code symbols per source symbol when symbols are coded one at a time.

a_2	$0.4^{(1)}$	$0.4^{(1)}$	$0.4^{(1)}$	$0.4^{(1)}$	$0.6(0)$
a_6	$0.3^{(00)}$	$0.3^{(00)}$	$0.3^{(00)}$	$0.3^{(00)}$	$0.4(1)$
a_1	$0.1^{(011)}$	$0.1^{(011)}$	$0.2^{(010)}$	$0.3_{(01)}$	
a_4	$0.1^{(0100)}$	$0.1_{(0100)}$	$0.1_{(011)}$		
a_3	$0.06^{(01010)}$	$0.1_{(0101)}$			
a_5	$0.04_{(01011)}$				

Average Length: 2.2 bits/symbol. **Entropy:** 2.14 bits/symbol.

Huffman coding: steps summary

Steps

1. Arrange symbol probabilities in decreasing order
2. while there is more than one node
   (a) merge the two nodes with the smallest probabilities to form a new node with the probability equal to their sum
   (b) Arbitrarily assign 1 and 0 to each pair of branches merging into a node

   Read sequentially from the root node to the leaf node where the symbol is located

Huffman Coding: Properties

- Lossless code: you can recover the original symbols without loss.

- Uniquely decodable: there is a one-to-one mapping from the code words to symbols.
- Instantaneous: no future referencing needed to decode. This means as soon as the code corresponding to a symbol is received, you can decode. There is no waiting to get the next code word before decoding the current code. Note that loss-less and unique decodability does not always guaranteee instantaneous decodability. This is possible because no complete code word forms a prefix to any other code word in the case of Huffman codes.

2 Transform Coding

So far we have reviewed generic error-free coding. The Shannon coding theorem gives you a lower bound on what is possible but does not give you an algorithm to achieve that bound. The Huffman coding method gives you an error free coding scheme that is optimal if one symbol is coded at a time.

Huffman codes: further notes

Optimality of Huffman codes

- Huffman codes are optimal when one symbol is coded at a time.
- Suppose you have N symbols, say $a_1, a_2, \cdots, a_N$. You construct a mapping from a_i to the corresponding H_i representing the Huffman code. Then you compute the resulting entropy of this constructed code set H_i. This gives you the minimal average length code that is possible for any such mapping.
- There could be multiple Huffman codes that can achieve the same lower bound.

How to achieve the Shannon bound?

One possibility is to start coding groups of symbols at a time.

For example, instead of coding a_i, we start coding the pairs a_i, a_j. You can follow the exact same procedure as before for constructing the Huffman codes, but your new symbol set would include all possible pairwise combinations. Note that this will also change the underlying probability distribution.

You can keep coding longer and longer sequences, and this will, in the limit, approach the lower bound as specified by Shannon.

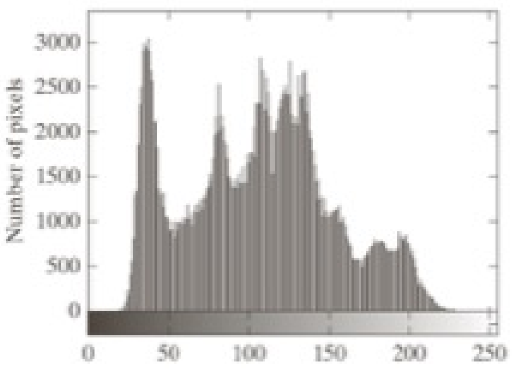
Another alternatives is Arithmetic Coding. You can refer to the book by Gonzalez & Woods for a more detailed discussion.

In the context of image compression, Huffman-type coding is often used to encode symbols that are to be preserved loss-less, as we will see next in transform coding.

Case for Transform Coding

- From the previous discussion we see that concatenating the symbols help.
- In doing so, you are taking advantage of the fact that not all combinations are equi-probable.
- we can take this one step further by looking *predictive coding*, where one predicts what comes next based on what you have seen so far. This is exploited in video coding where prediction is done from one frame to the next in a sequence of frames.
- Another option is decorrelate the symbols: we explored this in great detail in our discussions on KLT, DFT and DCT. We know that coding the decorrelated set is the best one can hope for in terms of compression.

Lena Image and its histogram



Image

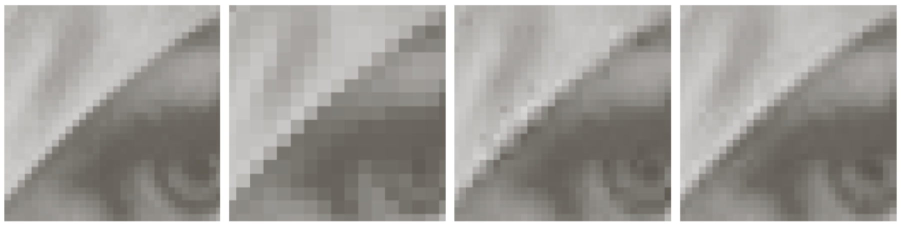
courtesy: Gonzalez & Woods Chapter 8

Case for transform coding: fewer coefficients will do!

(left) Fourier (RMSE 2.32), (middle) Walsh-Hadamard (RMSE 1.78) and (right) Cosine (RMSE 1.13). In each case the reconstructed 512×512 image used only 32 coefficients. Error images in the bottom row

2.1 Block Transform Coding

Case for block transform coding: smaller blocks?

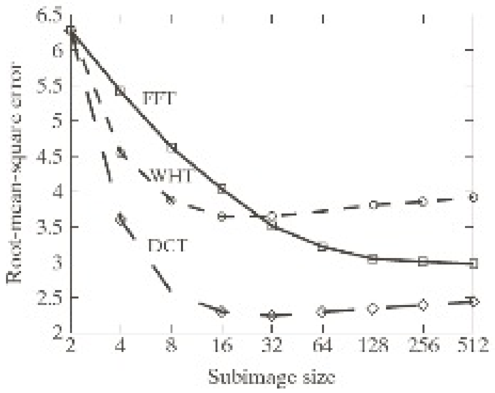


from left to right: a) Using DCT (left) 25% whole image, b) 2×2, c) 4×4, d) 8×8 subimages (zoomed).

Smaller subimages are good

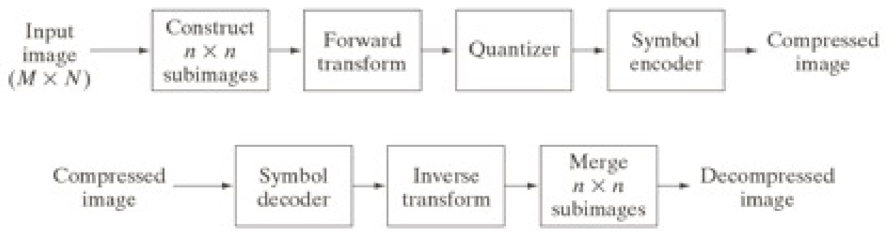
- computationally simpler (think of 1980s when you can implement a 8×8 transform in hardware
- local correlation: going to larger sub-image sizes does not really help that much

Block transform coding: case for DCT



Reconstruction error vs sub-image size for the different transforms. Image courtesy: Gonzalez & Woods Chapter 8

Block Transform coding-decoding: schematic



Key components

- Forward transform: use DCT
- Quantizer: this step is non-reversible, you quantize the coefficients and set most of them to zero
- symbol encoder: encode the remaining coefficients loss-less (e.g. use Huffman)

3 Quantization of coefficients

Quantization Strategies

Key Questions (all related to each other)

- Which coefficients to keep?
- How to quantize them?
- How many bits to allocate?

Option 1: Threshold coding

- Code each subimage adaptively
- For each subimage
  - Arrange the transform coefficients in decreasing order of magnitude
  - Keep only the top N coefficients and discard the rest (set to zero)
  - code the retained coefficients with a variable length code

Threshold coding is adaptive to each sub-image, so results in less error.

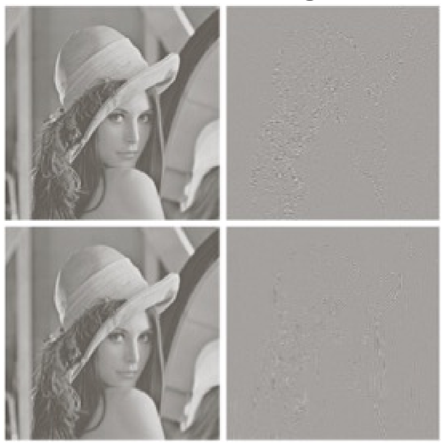
However, you will need to let the decoder know which coefficients are kept, so increases the overhead on the metadata needed to transmit an image.

Option 2: Zonal coding

- Using all of the sub-image blocks compute the variance of each of the transform coefficients.
- Sort the coefficients by magnitude and identify the top N coefficients with the largest magnitude
- In each subimage keep the corresponding N coefficients identified by theier variance magnitude in the above step. Discard the rest.
- Bit allocation: Suppose B bits are available to code each block. Let N be the number of retained coefficients and let v_i be the variance of the i-th coefficient. Then the number of bits b_i allocated to the i-th coefficient is:

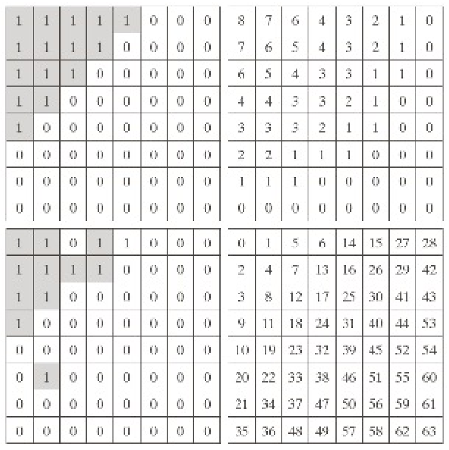
$$b_i = \frac{B}{M} + \frac{1}{2}\log_2 v_i - \frac{1}{2M}\sum_{i=1}^{M}\log_2 v_i$$

Zonal vs Threshold coding



Top row: Threshold coding, 8 × 8 block, 8 largest coefficients per block. Bottom row: zonal coding, 8 top coefficients with the largest variance kept for each block. Error images on the right.

Zonal mask vs zig-zag scanning for threshold coding



1	1	1	1	1	0	0	0
1	1	1	1	0	0	0	0
1	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

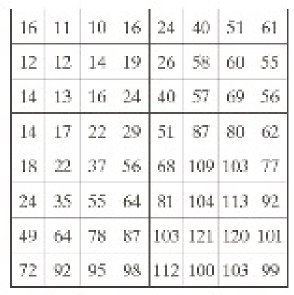
8	7	6	4	3	2	1	0
7	6	5	4	3	2	1	0
6	5	4	3	3	1	1	0
4	4	3	3	2	1	0	0
3	3	3	2	1	1	0	0
2	2	1	1	1	0	0	0
1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0

1	1	0	1	1	0	0	0
1	1	1	1	0	0	0	0
1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Top row: example of a Zonal coding mask and corresponding zonal bit allocation. Bottom row: Example of threshold coding mask with zig-zag coding (right side).

Threshold coding with a Quantization Matrix

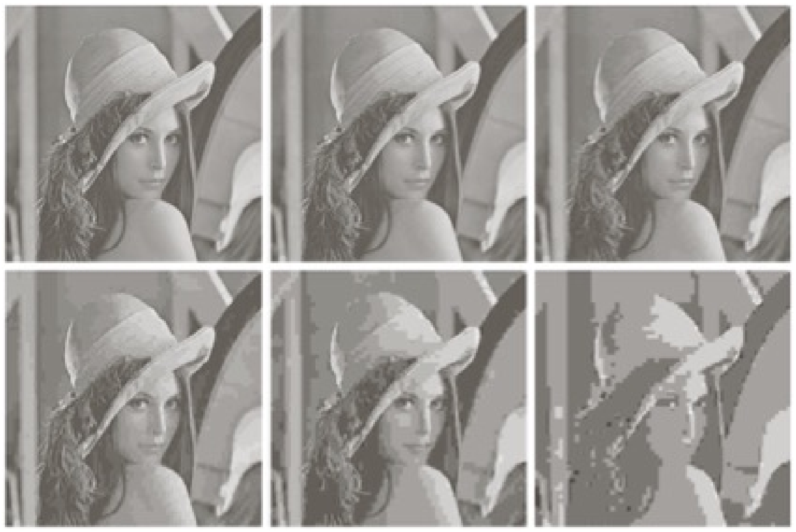


[Q] =

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

Divide each coefficient in each sub-image block by the corresponding value from the Quantization matrix shown above. Round off the result to the nearest integer value, which are then coded with a symbol encoder. Note that the low frequency coefficients are given higher priority (low values in the Q-matrix) and most of the higher frequency terms are set to zero.

Quantization matrix example



(right-left and top-bottom). Quantization by Q, 2Q, 4Q, 8Q, 16Q, 32Q

Ready for JPEG?

- JPEG is a lossy compression standard using DCT.
- Activities started in 1986 and bacame an international standard (IS) in 1992.
- Four modes of operation: Sequential (basline), hierarchical, progressive, and lossless.
- Arbitrary image sizes; DCT mode 8-12 bits/ sample. Luminance and chrominance channels are separately encoded.
- We will discuss the baseline method here

JPEG Baseline: simplified

- DCT: The image is divided into 8x8 blocks.
- Each pixel is level shifted by 2^{n-1} where 2^n is the maximum number of gray levels in the image. Thus for 8 bit images, you subtract 128.
- Then the 2-D DCT of each block is computed. For the baseline system, the input and output data precision is restricted to 8 bits and the DCT values are restricted to 11 bits.
- Quantization: the DCT coefficients are threshold coded using a quantization matrix, and then reordered using zig-zag scanning to form a 1-D sequence.

- The non-zero AC coefficients are Huffman coded. The DC coefficients of each block are (DPCM) coded relative to the DC coefficient of the previous block.

JPEG walk through (you will implement this)

The following example is from Gonzalez book.

58	64	67	64	59	62	70	78
56	55	67	89	98	88	74	69
60	50	70	119	141	116	80	64
69	51	71	128	149	115	77	68
74	53	64	105	115	84	65	72
76	57	56	74	75	57	57	74
83	69	59	60	61	61	67	78
93	81	67	62	69	80	84	84

original image block

Level shifted by 128

Next step: subtract 128 from each pixel value

−76	−73	−67	−62	−58	−67	−64	−55
−65	−69	−62	−38	−19	−43	−59	−56
−66	−69	−60	−15	16	−24	−62	−55
−65	−70	−57	−6	26	−22	−58	−59
−61	−67	−60	−24	−2	−40	−60	−58
−49	−63	−68	−58	−51	−65	−70	−53
−43	−57	−64	−69	−73	−67	−63	−45
−41	−49	−59	−60	−63	−52	−50	−34

After 2D DCT

Then take the 2D DCT of the block

−415	−29	−62	25	55	−20	−1	3
7	−21	−62	9	11	−7	−6	6
−46	8	77	−25	−30	10	7	−5
−50	13	35	−15	−9	6	0	3
11	−8	−13	−2	−1	1	−4	1
−10	1	3	−3	−1	0	2	−1
−4	−1	2	−1	2	−3	1	−2
−1	−1	−1	−2	−1	−1	0	−1

After divide by Q matrix

Recall the Q matrix earlier; divide by the Q matrix, element by element. JPEG allows you to choose your own Q matrix and embed that as part of the JPEG file.

$$\begin{matrix}
-26 & -3 & -6 & 2 & 2 & 0 & 0 & 0 \\
1 & -2 & -4 & 0 & 0 & 0 & 0 & 0 \\
-3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\
-4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{matrix}$$

Zig-zag scanning

now scan the coefficients in the zig-zag manner; this scanning prioritizes the low frequency coefficients first. The EOB indicates a special code for the end-of-block.

[−26 −3 1 −3 −2 −6 2 −4 1 −4 1 1 5 0 2 0 0 −1 2 0 0 0 0 0 −1 −1 EOB]

At this point, the DC value = (0,0)-th coefficient – is DPCM coded, i.e., the difference with respect to the previous block's DC value is encoded.

The AC values are coded separately, and the code includes the number of zero-valued coefficients preceding the non-zero coefficient.

For example, the first AC value '-3' has no zero coefficient preceding it. So the code should include both the magnitude information as well as the number of zero-valued coefficients (which is 0 in this case) preceding.

Bit coding

1010110 0100 001 0100 0101 100001 0110 100011 001 100011 001
001 100101 11100110 110110 0110 11110100 000 1010

This step is bit more(!) involved Look up tables in JPEG provide a mapping between the sign, magnitude and number of zero-coefficients preceding, to the bit representation.

This is the optional part of your HW.

Symbol decoder output and reassembly

−26	−3	−6	2	2	0	0	0
1	−2	−4	0	0	0	0	0
−3	1	5	−1	−1	0	0	0
−4	1	2	−1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

symbol decoder faithfully reproduces the coefficient values from the bit representation.

denoramlization by Q

−416	−33	−60	32	48	0	0	0
12	−24	−56	0	0	0	0	0
−42	13	80	−24	−40	0	0	0
−56	17	44	−29	0	0	0	0
18	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

here you multiply back by the quantization matrix Q, element by element.

Inverse DCT

−70	−64	−61	−64	−69	−66	−58	−50
−72	−73	−61	−39	−30	−40	−54	−59
−68	−78	−58	−9	13	−12	−48	−64
−59	−77	−57	0	22	−13	−51	−60
−54	−75	−64	−23	−13	−44	−63	−56
−52	−71	−72	−54	−54	−71	−71	−54
−45	−59	−70	−68	−67	−67	−61	−50
35	47	61	66	60	48	44	44

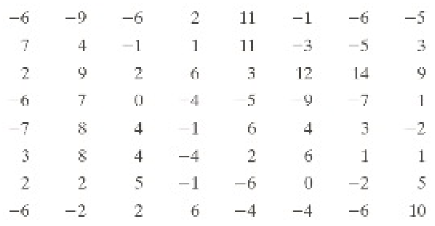
Restore back the pixel values

level shifting back

58	64	67	64	59	62	70	78
56	55	67	89	98	88	74	69
60	50	70	119	141	116	80	64
69	51	71	128	149	115	77	68
74	53	64	105	115	84	65	72
76	57	56	74	75	57	57	74
83	69	59	60	61	61	67	78
93	81	67	62	69	80	84	84

.. and add back 128, to get the reconstructed block

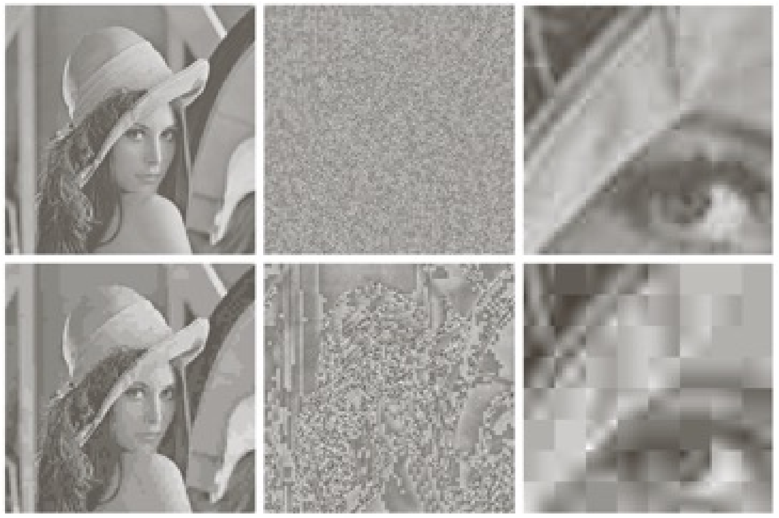
error image



−6	−9	−6	2	11	−1	−6	−5
7	4	−1	1	11	−3	−5	3
2	9	2	6	3	12	14	9
−6	7	0	−4	−5	−9	−7	1
−7	8	4	−1	6	4	3	−2
3	8	4	−4	2	6	1	1
2	2	5	−1	−6	0	−2	5
−6	−2	2	6	−4	−4	−6	10

This is the difference between the original block and the reconstructed block

Real image example



Concluding remarks

- The whole process can be extended to color images. The images are coded in the YCrCb color space where Y represents the luminance values and Cr, Cb represent the color. Y values are coded as just explained. The Cr and Cb are sub-sampled by a factor of 2 and then encoded similarly.
- The quantization matrix Q is used as knob to control the compression quality
- Blocking artifacts are visible at high compression factors (low quality).