Tidbits on Image Compression

(Above, Lena, unwitting data compression spokeswoman)

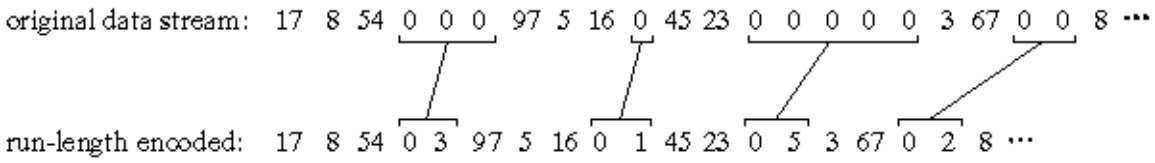
In CS203 you probably saw how to create Huffman codes with greedy algorithms. Let's examine some other methods of compressing data (run-length, arithmetic, and LZW) and also look briefly at the format used in JPEG images.

Run-Length Encoding

Data files often contain the same character repeated in sequence. For example, text files contain multiple spaces for formatting tables or columns. Images may contain multiple 0's or 1's for all black or all white. Digitized signals often contain long runs of zeros, for example quiet time between notes or times when the signal is not changing.

The idea behind run length encoding is to represent long runs of zeros. Each run of zeros (or some other repeating character) is represented by a flag indicating that run-length compression is beginning. The flag is followed by the number of zeros. The flag could be expanded to include the actual repeating character if we wish to use this technique for more than just zeros.

The process on a sample datastream is indicated below:



The PackBits program on the Macintosh used a generalized RLE scheme for data compression.

Arithmetic Encoding

In arithmetic encoding, we turn an entire datastream into a single number! The more data we have, the greater precision we will need in representing the number.

There are two fundamentals in arithmetic coding: the probability of a symbol and its encoding interval range. The probabilities of source symbols determine the compression efficiency. They also determine the interval ranges of source symbols for the encoding process. These interval ranges are contained within the interval from zero to one. The

interval ranges for the encoding process determine the compression output. This is best demonstrated by an example.

Let us assume that the source symbols are { 00, 01, 10, 11 } and the probabilities of these symbols are { 0.1, 0.4, 0.2, 0.3 }, respectively. Then, based on these probabilities, the interval [0,1) can be divided as four sub-intervals: [0,0.1), [0.1,0.5), [0.5,0.7), [0.7,1), where [x,y) denotes a half open interval, which includes x but excludes y. The above information can be summarized in table:

Symbols	00	01	10	11
Probabilities	0.1	0.4	0.2	0.3
Initial Encoding Intervals	[0,0.1)	[0.1,0.5)	[0.5,0.7)	[0.7,1)

To encode a message of a binary sequence 10 00 11 00:

We take the first symbol 10 from the message and find its encoding range is [0.5,0.7). Since the range of the second symbol 00 from the message is [0,1), it is encoded by taking the first 10th of interval [0.5,0.7) as the new interval [0.5,0.52). In other words, we treat the range from [0.5, 0.7) as an entire interval, and then use the first 0.1 of this interval for the second symbol. This process is repeated for the rest of the symbols. To encode the third symbol 11, we have a new interval [0.514,0.52). After encoding the fourth symbol 00, the new interval is [0.514,0.5146). The compression output of this message can be any number in the last interval. For example, we could pick 0.5145.

To decode the data, we apply the process in reverse. We need to know the probability ranges, presumably transmitted with the data unencoded:

Given 0.5145, the value is in the range [0.5, 0.7) so it is symbol 10.
Given 0.5145, the value is in the 1^{st} 10^{th} of the interval [0.5, 0.7) so it is symbol 00.
Given 0.5145, the value is in the 7^{th} 10^{th} of the interval [0.5, 0.52) so it is symbol 11.
Given 0.5145, the value is in the 1^{st} 10^{th} of the interval [0.514, 0.52) so it is symbol 00.

The resulting sequence of symbols is now 10, 00, 11, 00.

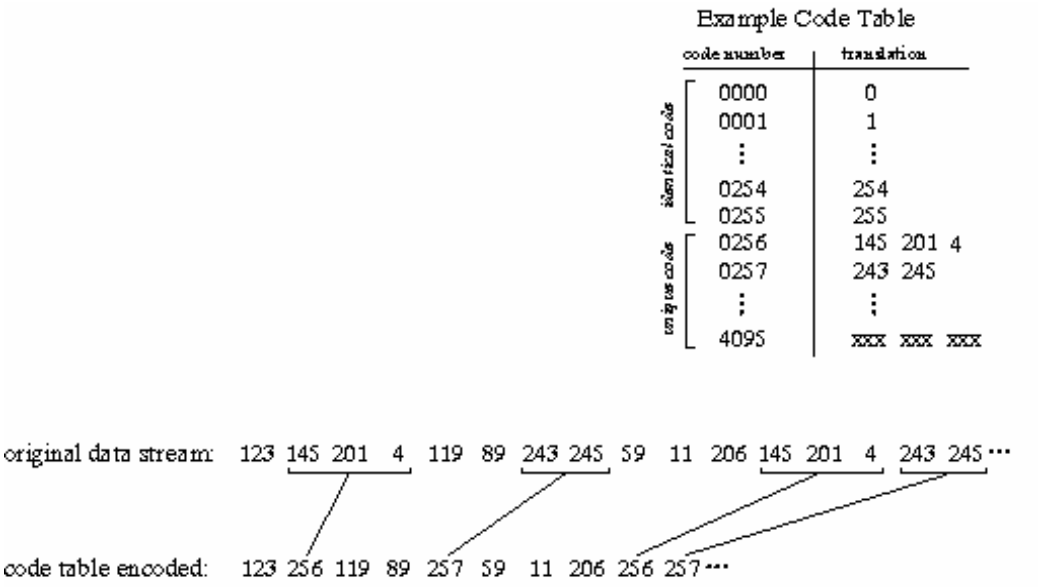
Some issues to note:

1) Since no single machine exists with an infinite precision, "underflow" and "overflow" are the obvious problems for the real world machines. We can use multiple bytes or words to represent higher precision if necessary, at additional expense in decoding these bytes and mapping them to the precision of the machine (e.g. 32, 64 bits).
2) An arithmetic coder produces only one codeword, a real number in interval [0,1), for the entire message to be transmitted. We cannot perform decoding process until we received all bits representing this real number.
3) Arithmetic coding is an error sensitive compression scheme. A single bit error can corrupt the entire message.

LZW Compression

This is a very popular form of compression that is used as the basis for many commercial and non-commercial compression programs (gzip, pkzip, GIF, compressed postscript, "disk doublers"). LZW compression is named after its developers, A. Lempel and J. Ziv, with later modifications by Terry A. Welch. It is the foremost technique for general purpose data compression due to its simplicity and versatility. Typically, you can expect LZW to compress text, executable code, and similar data files to about one-half their original size.

LZW is similar to Huffman encoding, except it uses a code table for sequences of characters. Consider the example below:



LZW compression uses a code table. A common choice is to provide 4096 entries in the table. In this case, the LZW encoded data consists entirely of 12 bit codes, each referring to one of the entries in the code table. Uncompression is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents. Codes 0-255 in the code table are always assigned to represent single bytes from the input file. For example, if only these first 256 codes were used, each byte in the original file would be converted into 12 bits in the LZW encoded file, resulting in a 50% larger file size. During uncompression, each 12 bit code would be translated via the code table back into the single bytes. Of course, this wouldn't be a useful situation.

The LZW method achieves compression by using codes 256 through 4095 to represent sequences of bytes. For example, code 523 may represent the sequence of three bytes: 231 124 234. Each time the compression algorithm encounters this sequence in the input file,

code 523 is placed in the encoded file. During uncompression, code 523 is translated via the code table to recreate the true 3 byte sequence. The longer the sequence assigned to a single code, and the more often the sequence is repeated, the higher the compression achieved.

Although this is a simple approach, there are two major obstacles that need to be overcome: (1) how to determine what sequences should be in the code table, and (2) how to efficiently determine if a sequence of data has an entry in the code table.

#1 is the subject of many algorithms (the authors of LZW propose one way to determine the codes using a state table). For example, one could devise any algorithm to find repeat strings and use that to determine the code table sequences. The longest, most repeating strings could be given a code assignment.

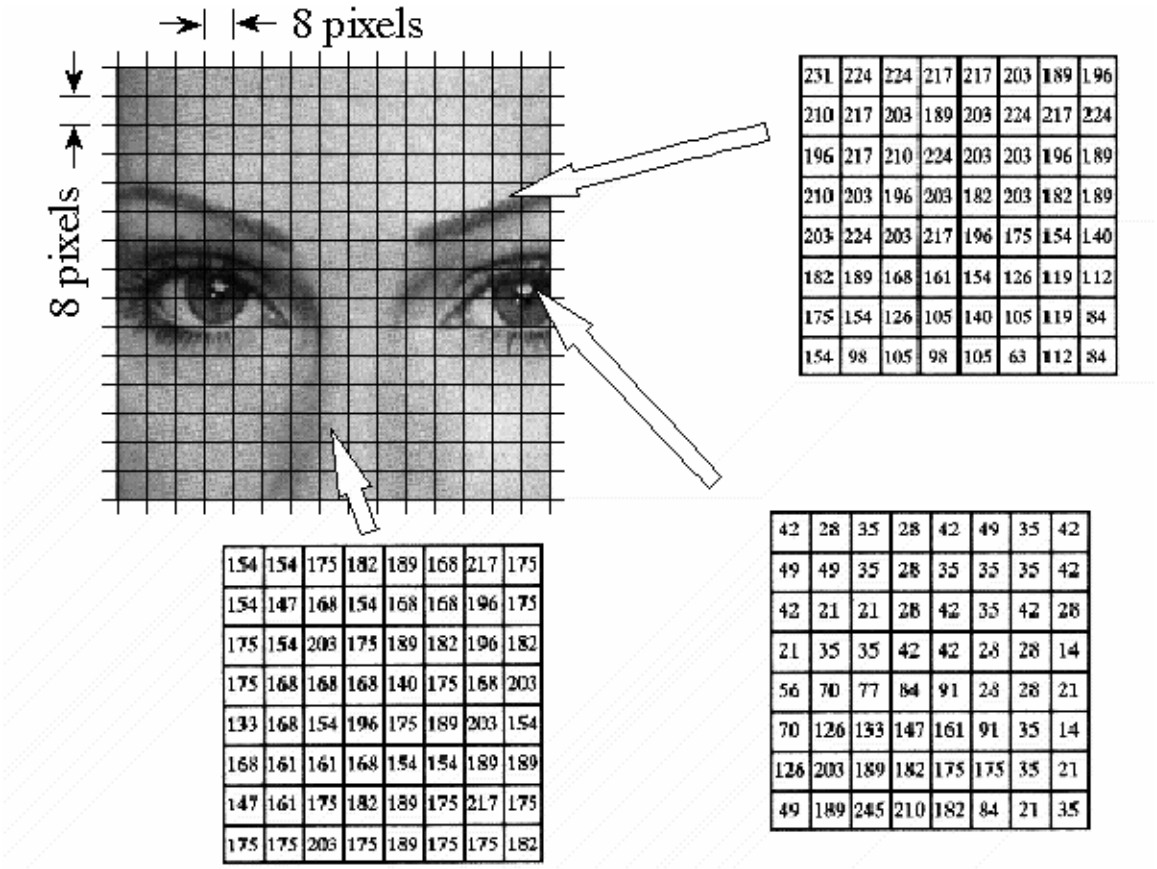
#2 could be implemented in a simple or complicated manner. However this implementation will greatly affect the execution time of the compression algorithm because it must search the code table to determine if a match is present. As an analogy, imagine you want to find if a friend's name is listed in the telephone directory. The catch is, the only directory you have is arranged by telephone number, not alphabetical order. This requires you to search page after page trying to find the name you want. This inefficient situation is exactly the same as searching all 4096 codes for a match to a specific character string. The answer: organize the code table so that what you are looking for tells you where to look (like a partially alphabetized telephone directory). In other words, don't assign the 4096 codes to sequential locations in memory. Rather, divide the memory into sections based on what sequences will be stored there. For example, suppose we want to find if the sequence: code 329 + x, is in the code table. The code table should be organized so that the "x" indicates where to starting looking. There are other schemes using hash tables, additional indices, and other techniques to help speed up the lookup process. There are many schemes for this type of code table management, and they can become quite complicated.

JPEG Overview

Many methods of lossy compression have been developed; however, a family of techniques called transform compression has proven the most valuable. The most popular for images, and one of the best techniques, is the JPEG format. JPEG is named after its originating group, the Joint Photographers Experts Group. We will describe the general operation of JPEG to illustrate how lossy compression works.

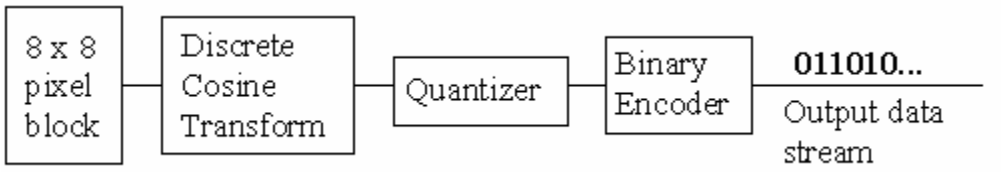
JPEG compression starts by breaking the image into 8x8 pixel groups. The full JPEG algorithm can accept a wide range of bits per pixel, including the use of color information. In this example, each pixel is a single byte, a grayscale value between 0 and 255. These 8x8 pixel groups are treated independently during compression. That is, each group is initially represented by 64 bytes. Why use 8x8 pixel groups instead of, for

instance, 16x16? The 8x8 grouping was based on the maximum size that integrated circuit technology could handle at the time the standard was developed. In any event, the 8x8 size works well, and it may or may not be changed in the future.



The 8x8 group idea is demonstrated above on a greyscale image. If we were working with a color image, we would essentially have the same process for a Red, Green, and Blue component.

Each 8x8 group is then transformed from a spatial pixel dimension to a frequency dimension. In other words, the image is treated like a signal and represented as a sum of frequencies. This is done by using the **Discrete Cosine Transform (DCT)**. A **quantizer** rounds off the DCT coefficients according to a quantization matrix. This step produces the "lossy" nature of JPEG, but allows for large compression ratios. JPEG's compression technique uses a variable length code on these coefficients, and then writes the compressed data stream to an output file (*.jpg). For decompression, JPEG recovers the quantized DCT coefficients from the compressed data stream, takes the inverse transforms and displays the image. This entire process is shown below:



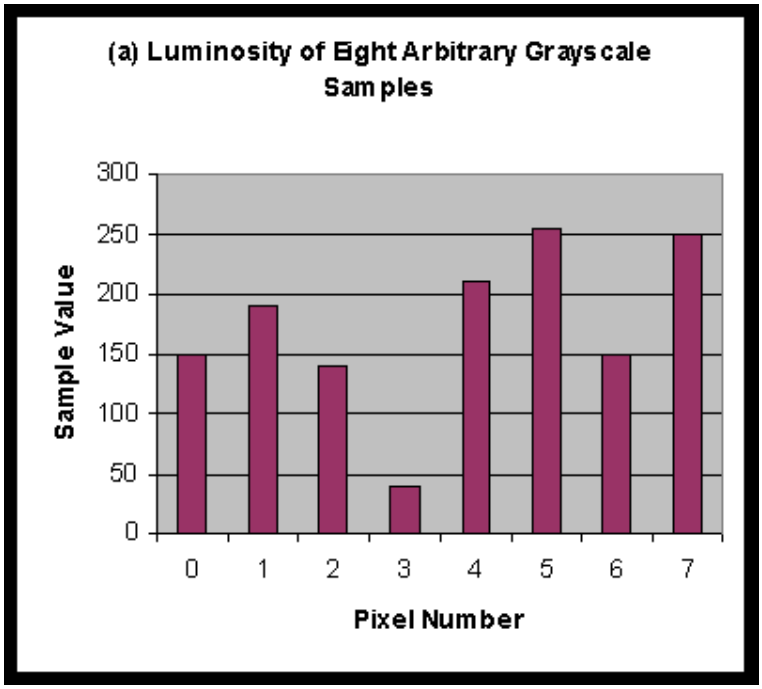
The Discrete Cosine Transform (DCT)

The key to the JPEG baseline compression process is a mathematical transformation known as the Discrete Cosine Transform (DCT). The DCT is in a class of mathematical operations that includes the well known Fast Fourier Transform (FFT), as well as many others. The basic purpose of these operations is to take a signal and transform it from one type of representation to another. For example, an image is a two-dimensional signal that is perceived by the human visual system. The DCT can be used to convert the signal (spatial information) into numeric data ("frequency" or "spectral" information) so that the image's information exists in a quantitative form that can be manipulated for compression.

The signal for a graphical image can be thought of as a three-dimensional signal. The X and Y-axes of the image's signal are the two dimensions of the screen, while the amplitude of the signal, the Z-axis, is the value of the pixel at (X, Y). This can be represented visually by a two-dimensional array where each cell contains the numerical value of the pixel at that location. As the specifics of a two-dimensional DCT matrix are rather complex, we will simplify the problem by first considering the derivation and intentions of a one-dimensional DCT matrix.

The One-Dimensional DCT

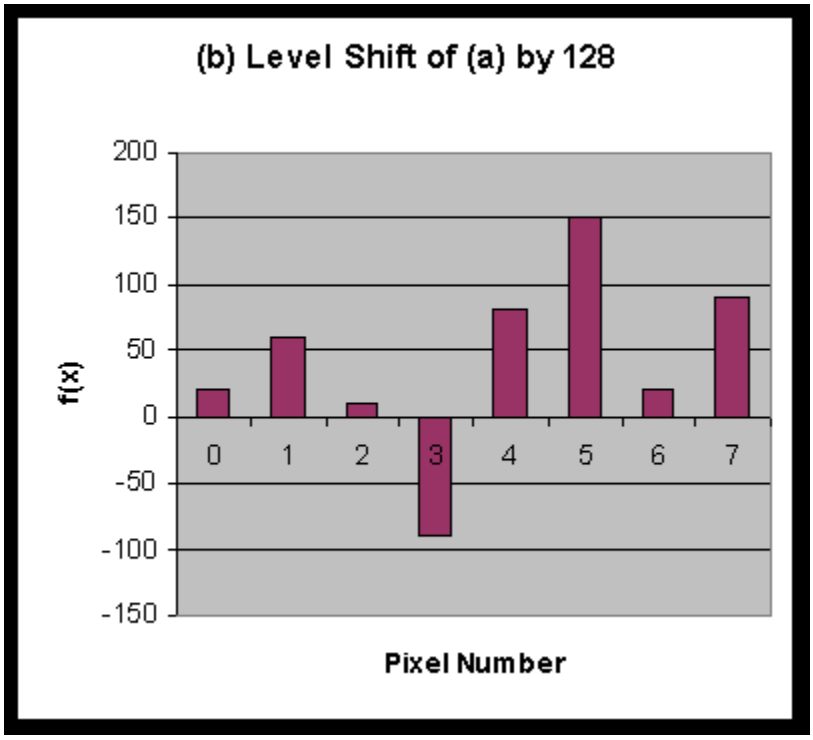
We start with a set of eight arbitrary grayscale samples as charted below, where each bar represents the luminosity of a single pixel.



These values contain all the information necessary to define the eight pixels. Thus the ultimate goal is to compress this data so it can be stored or transmitted and later decompressed to re-form the original image. However, as explained above, simple

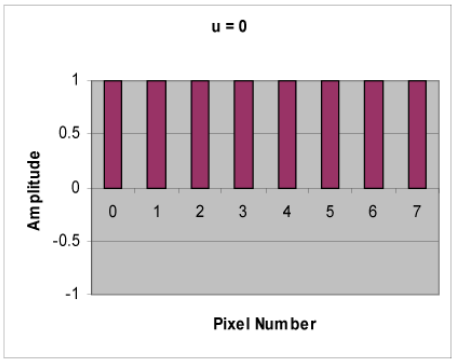
entropy or statistical encoding of this data will not be extremely effective because in continuous tone images, the levels of luminosity have equal probabilities of occurring. As a more effective alternative, the DCT can manipulate this data, separating information crucial to the definition of the image from information that's presence is not perceivable by the human eye. The insignificant information can then be "discarded" through the quantization phase of JPEG coding, thus achieving large-scale compression. Simply put, the purpose of the DCT transformation phase is to *identify* "pieces of information in the image's signal that can be effectively 'thrown away' without seriously compromising the quality of the image" (Nelson 359). No information is lost, nor is any compression achieved, in the DCT stage. This initial phase is merely a preparatory step that allows for and leads to the lossy coefficient quantization stage that follows.

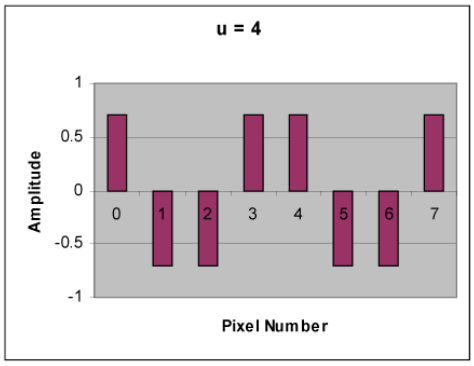
The first step involved in the "rearrangement" of the data displayed in figure (a) is to perform a level shift by 128, the result of which is shown in figure (b) below.

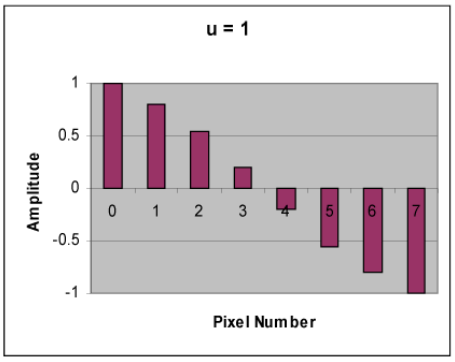


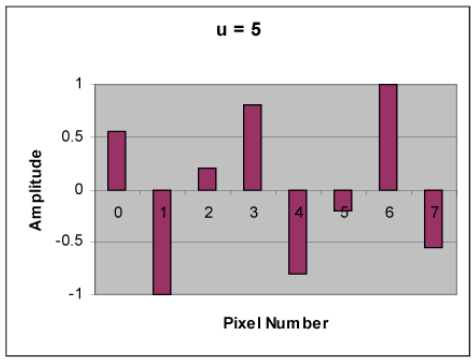
The samples have values in the range of 0 to 255. By shifting the level of their graph by 128, half their range, we obtain the values of f(x) in figure (b). Using f(x), we can decompose the eight sample values into a set of waveforms of different spatial frequencies. This decomposition is where the separation of the more significant low-frequency components from the less significant high-frequency components takes place.

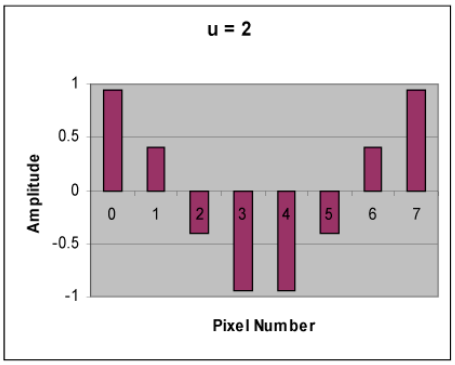
Below is a set of waveforms of eight different spatial frequencies, all of uniform amplitude and each sampled at eight points.

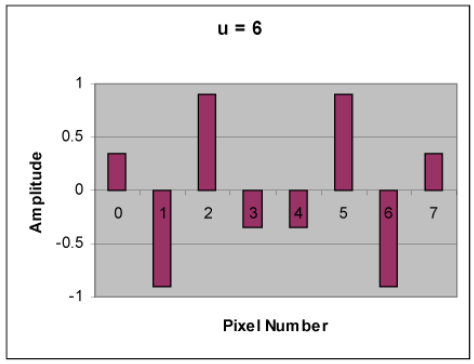

u = 0
Amplitude
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7
Pixel Number

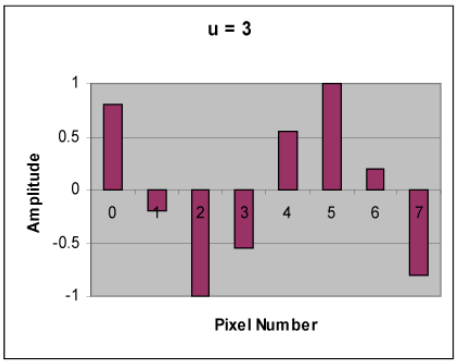

u = 4
Amplitude
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7
Pixel Number

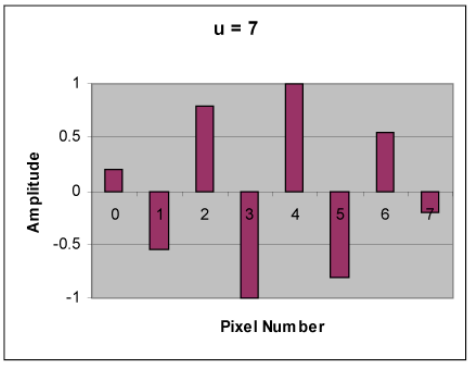

u = 1
Amplitude
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7
Pixel Number


u = 5
Amplitude
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7
Pixel Number

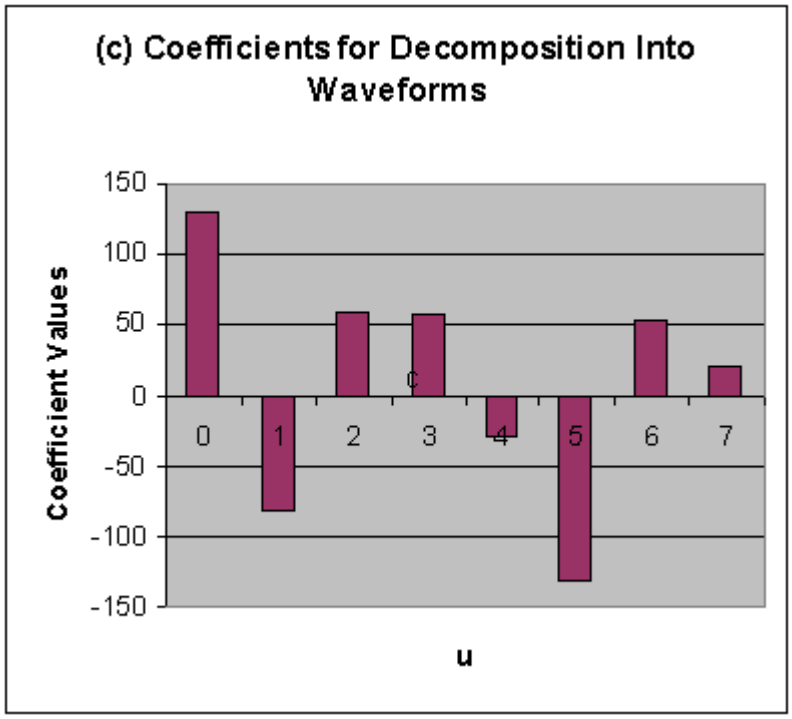

u = 2
Amplitude
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7
Pixel Number


u = 6
Amplitude
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7
Pixel Number


u = 3
Amplitude
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7
Pixel Number


u = 7
Amplitude
1
0.5
0
-0.5
-1
0 1 2 3 4 5 6 7
Pixel Number

The top left waveform (u = 0) is simply a constant, whereas the other seven waveforms (u = 1, …, 7) show an alternating behavior at progressively higher frequencies. These waveforms, which are called cosine basis functions, are independent, meaning that there is no way that a given waveform can be represented by any combination of the other waveforms. However, the complete set of eight waveforms, when scaled by coefficients and added together, can be used to represent any eight sample values such as those in figure (b). The intention is to use the Discrete Cosine Transform to determine the values of the coefficients. The coefficients plotted in figure (c) below are the output of an 8-point DCT for the eight sample values in figure (b).

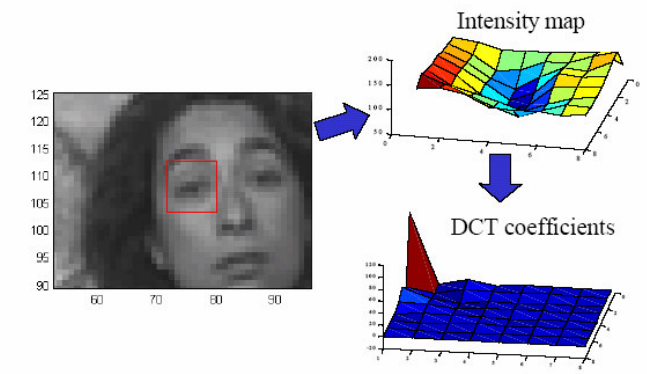


There is a direct correlation between the magnitude of the coefficient for a given waveform and the impact of that particular waveform on the quality of the picture. The coefficient that scales the constant basis function (u = 0) is called the DC (direct current) coefficient, while the other coefficients are called AC (alternating current) coefficients. Note that the DC term gives the average over the set of samples. Furthermore, the DC term is usually a great deal larger in magnitude than the AC terms, and, though it may not be evident in the small sampling depicted by figure (c), as the elements move farther away from the DC term, they tend to become lower and lower in value. Recall that the AC terms farther away from the DC term represent coefficients of waveforms of greater spatial frequencies. The fact that these coefficients tend to be smaller in magnitude suggests that higher-frequency image components play a relatively small role in the determining picture quality, while the majority of image definition comes from lower-frequency image components. This idea becomes extremely important when applied two-dimensionally to an image, for JPEG exploits this exact concept when deciding what information can be eliminated to achieve compression.

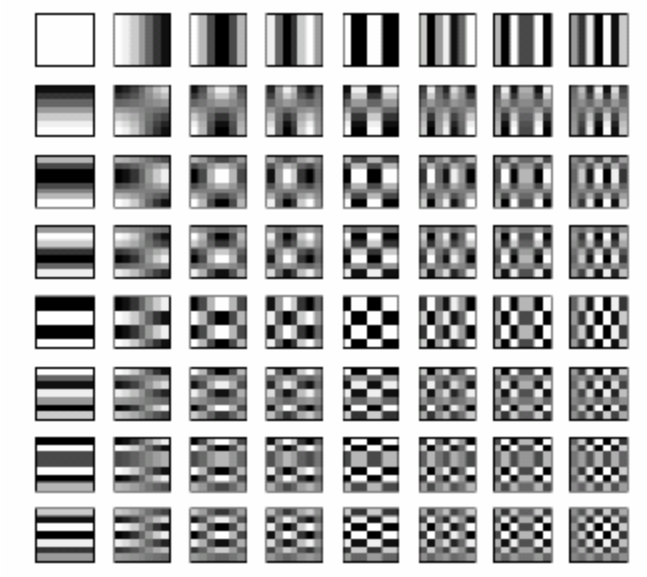
The Two-Dimensional DCT

The one-dimensional DCT described above can be extended to apply to two-dimensional image arrays. The two-dimensional cosine basis functions from which sample waveforms are composed are created by multiplying a horizontally oriented set of one-dimensional 8-point basis functions by a vertically oriented set of the same functions. It logically follows that the horizontally oriented set of basis functions represents horizontal frequencies and the other set of basis functions represents vertical frequencies.

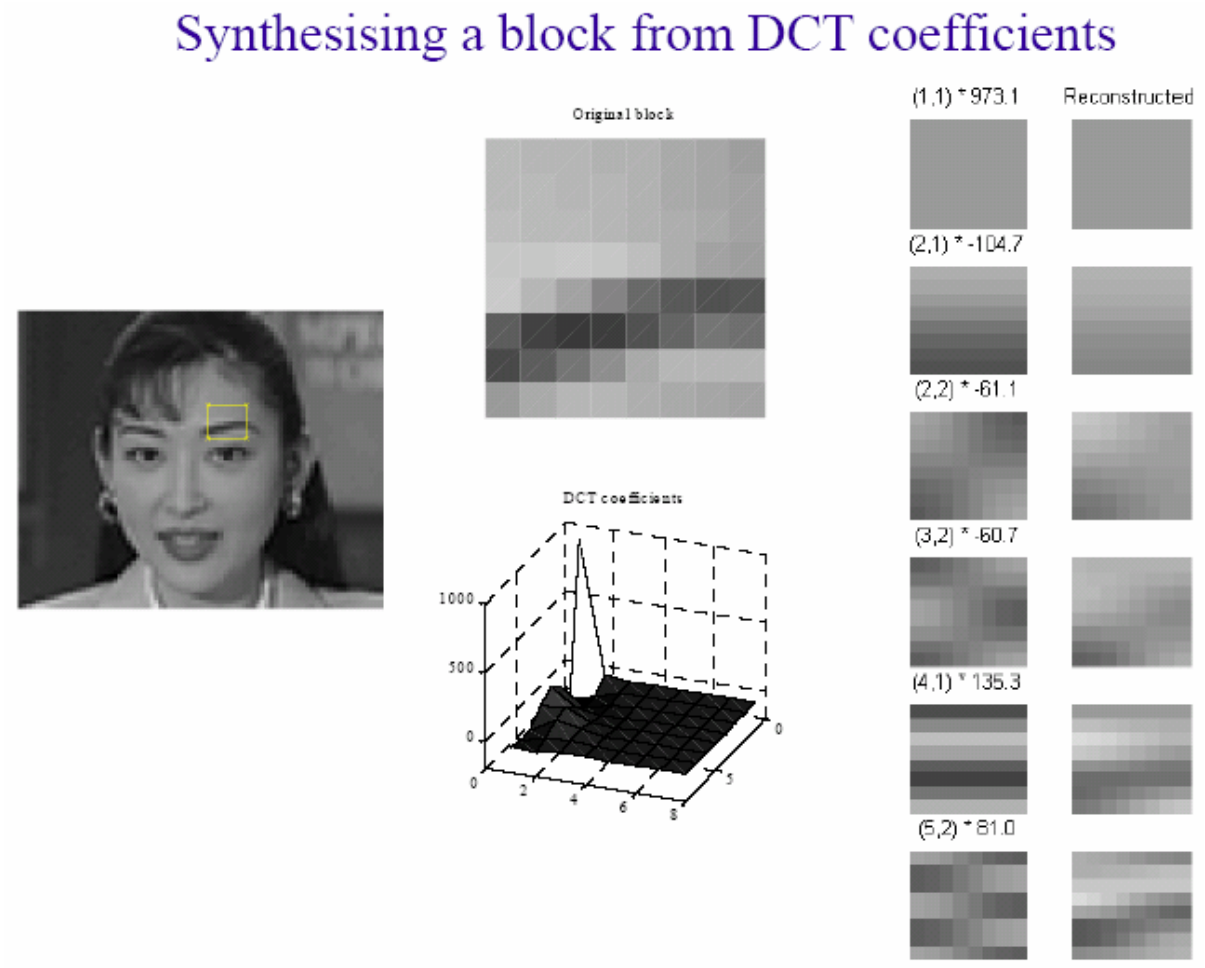
The overall process is depicted below. For each 8x8 pixel block, we map the intensity to DCT coefficients:



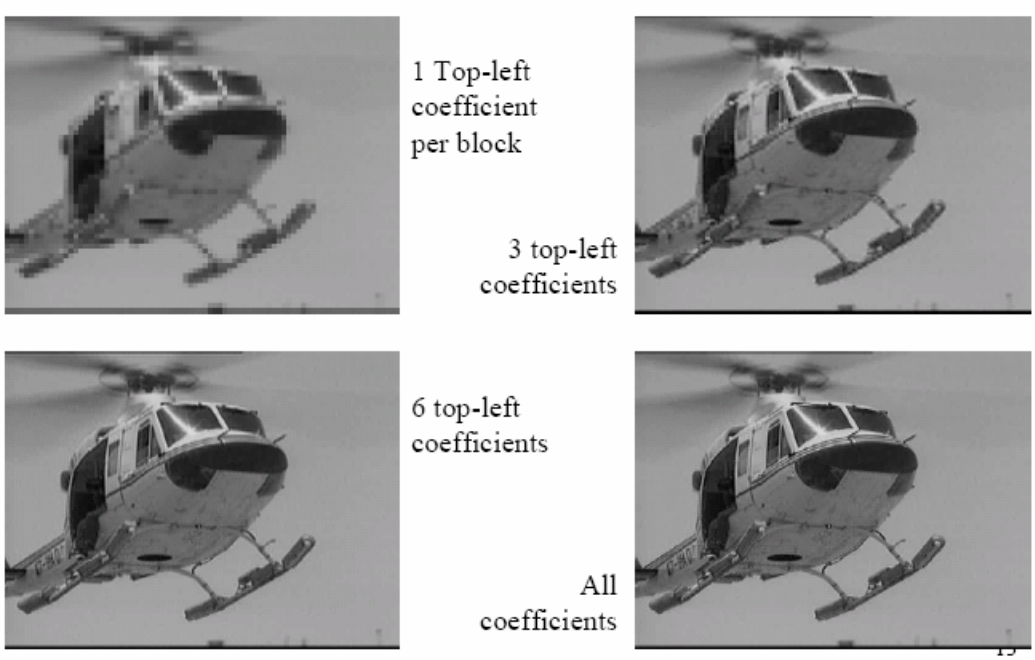
With an 8x8 matrix we have a total of 64 basis patterns. The upper left corner is the "DC" or bias term, and going across the top row we change the horizontal frequency. Going down columns we change the vertical frequency:



Any 8x8 block of pixels can be represented as a sum of the 64 basis patterns. The output of the DCT is the set of weights, or coefficients, of these patterns. Multiply each basis pattern by its weight and add them together to get back the original image. Some examples are shown below:



Energy tends to be concentrated into a few significant coefficients, generally at low frequencies. Other coefficients are closer to zero or insignificant. This is illustrated below:



By convention, the DC term of the horizontal basis functions is to the left, and the DC term for the vertical basis functions is at the top. Consequently, the top, left element of a two-dimensional DCT matrix contains a value that is almost always of a very great magnitude. Furthermore, mirroring the trend found in a one-dimensional DCT matrix, the farther away an AC term is from the DC term, the higher the frequency its corresponding waveform will have and the smaller its magnitude will be. Although the concept of the two-dimensional DCT is not as easily pictured using histograms as is the one-dimensional DCT, the two-dimensional version employs the same underlying principles and thus can be interpreted using identical reasoning.

The actual formula for the two-dimensional DCT is shown below. The DCT is performed on an N x N square matrix of pixel values, and it yields an N x N square matrix of frequency coefficients. (In practice, N most often equals 8 because a larger block, though would probably give better compression, often takes a great deal of time to perform DCT calculations, creating an unreasonable tradeoff. As a result, DCT implementations typically break the image down into more manageable 8 x 8 blocks.) The DCT formula looks somewhat intimidating at first glance but can be implemented with a relatively straightforward piece of code.

$$DCT(i, j) = \frac{1}{\sqrt{2N}} C(i)\, C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x, y) \cos\left[\frac{(2x + 1)i\pi}{2N}\right] \cos\left[\frac{(2y + 1)j\pi}{2N}\right]$$

$$C(x) = \frac{1}{\sqrt{2}} \text{ if } x \text{ is } 0, \text{ else } 1 \text{ if } x > 0$$

When writing code to implement this function, simple table lookups can replace several terms of the equation to simplify the appearance of the algorithm. The two cosine terms only need to be calculated once at the beginning of the program, and they can be stored for later use. Likewise, the C(x) terms can also be replaced with table lookups. Code to compute the DCT matrix for an N-by-N portion of a display looks somewhat like the following (adapted from *The Data Compression Book* by Mark Nelson).

```
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      temp = 0.0;
      for (x = 0; x < N; x++) {
         for (y = 0; y < N; y++) {
            temp += Cosines[x][i] *
               Cosines[y][j] *
               Pixel[x][y];
         }
      }
      temp *= sqrt(2 * N) * Coefficient[i][j];
      DCT[i][j] = INT_ROUND(temp);
   }
}
```

The above code is only one example of an algorithm that implements the DCT transformation on an N x N matrix. This particular algorithm is extremely inefficient due to a double nested for-loop. Although a considerably more efficient form of the DCT can be calculated using matrix operations, the algorithm displayed above is the most straightforward.

Below are two matrices representing the DCT input and DCT output blocks from a gray-scale image. Each element of the 8 pixel-by 8-pixel input matrix contains the value of the pixel at the corresponding (x, y) location. These integer values are fed to the DCT algorithm, creating the output matrix shown below it. Each element of the output matrix is a coefficient by which the waveform of the corresponding spatial frequency is multiplied in the decomposition of the image sample.

Input Pixel Matrix

140	144	147	140	140	155	179	175
144	152	140	147	140	148	167	179
152	155	136	167	163	162	152	172
168	145	156	160	152	155	136	160
162	148	156	148	140	136	147	162
147	167	140	155	155	140	136	162
136	156	123	167	162	144	140	147
148	155	136	155	152	147	147	136

Output DCT Matrix

186	-18	15	-9	23	-9	-14	19
21	-34	26	-9	-11	11	14	7
-10	-24	-2	6	-18	3	-20	-1
-8	-5	14	-15	-8	-3	-3	8
-3	10	8	1	-11	18	18	15
4	-2	-18	8	8	-4	1	-7
9	1	-3	4	-1	-7	-1	-2
0	-8	-2	2	1	4	-6	0

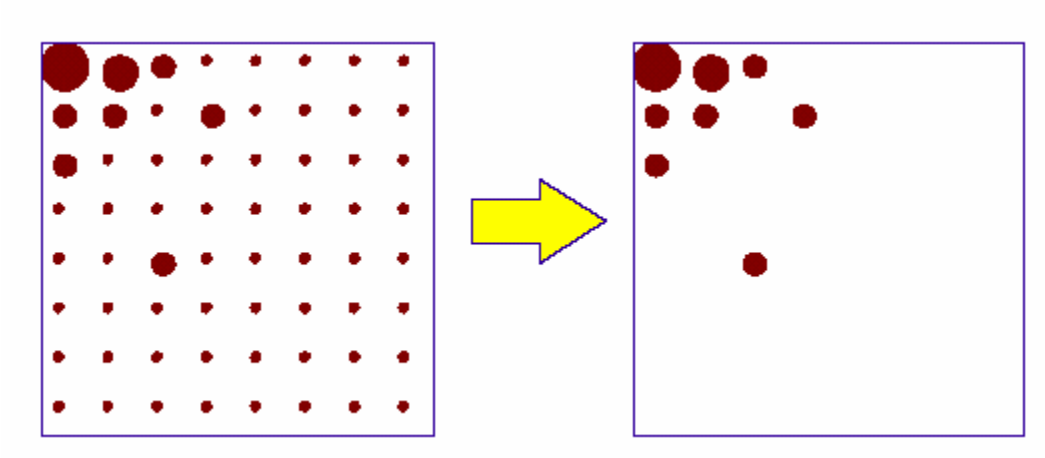
As shown in the output matrix, the DC coefficient, 186, is relatively large in magnitude, and the AC terms become lower in magnitude as they move farther from the DC coefficient. This means that by performing the DCT on the input data, we have concentrated the representation of the image in the upper left coefficients of the output

matrix, with the lower right coefficients of the DCT matrix containing less useful information. The next step, quantization of the coefficients in the output matrix, "discards" the less useful data and in turn compresses the image data.

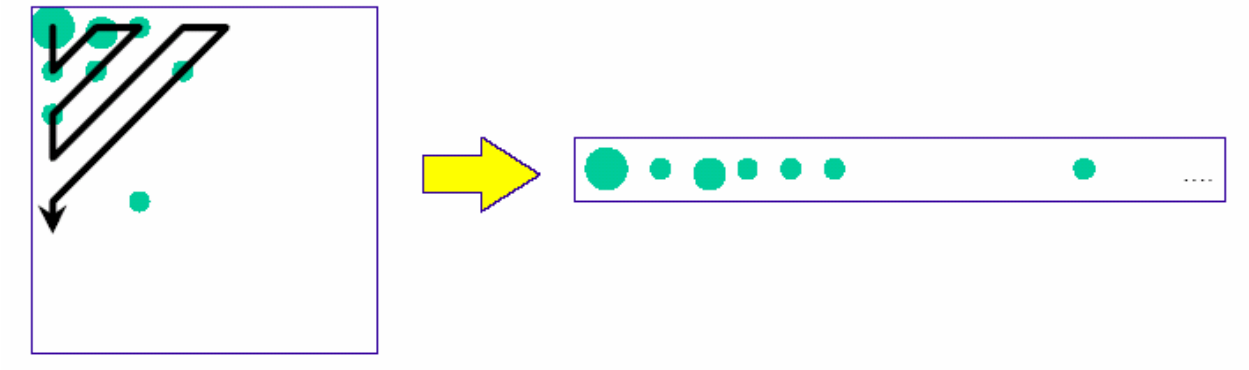
Coefficient Quantization

Quantization is the process of reducing the number of bits needed to store an integer value by reducing the precision of the integer. Given a matrix of DCT coefficients, we can generally reduce the precision of the coefficients more and more as we move away from the DC coefficient. This is because the farther away we are from the DC coefficient, the less the element contributes to the graphical image, and therefore, the less we care about maintaining rigorous precision in its value (Nelson 364).

The algorithm essentially divides each DCT coefficient by an integer and discards the remainder. The result is a loss of precision, but fewer bits that we need to store. Typically, only a few non-zero coefficients are left.



Since the low frequency components are most significant, the coefficients are scanned in a zig-zag pattern:



The resulting data stream is then further compressed using run-length encoding.

In more detail, the JPEG algorithm implements quantization using a separate quantization matrix, an example of which is shown below.

A Sample Quantization Matrix

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

For every element position in the DCT matrix, a corresponding value in the quantization matrix gives a quantum value indicating what the step size is going to be for that element. The coefficients that are most significant to the compressed rendition of the image (those closer to the upper left corner) are encoded with a small step size, while coefficients that are less important (those closer to the lower right corner) are encoded with larger step sizes. The actual formula for quantization is displayed below.

$$\text{Quantized Value}(i, j) = \frac{\text{DCT}(i, j)}{\text{Quantum}(i, j)} \text{ Rounded to the nearest integer}$$

From the formula, one can notice the smaller DCT coefficients of high-frequency elements divided by the larger quantum values will most often result in the high-frequency coefficients being rounded down to zero. The sample matrices below show the effects of quantization on a DCT matrix.

DCT Matrix Before Quantization

92	3	-9	-7	3	-1	0	2
-39	-58	12	17	-2	2	4	2
-84	62	1	-18	3	4	-5	5
-52	-36	-10	14	-10	4	-2	0
-86	-40	49	-7	17	-6	-2	5
-62	65	-12	-2	3	-8	-2	0
-17	14	-36	17	-11	3	3	-1
-54	32	-9	-9	22	0	1	3

DCT Matrix After Dequantization

90	0	-7	0	0	0	0	0
-35	-56	9	11	0	0	0	0
-84	54	0	-13	0	0	0	0
-45	-33	0	0	0	0	0	0
-77	-39	45	0	0	0	0	0
-52	60	0	0	0	0	0	0
-15	0	-19	0	0	0	0	0
-51	19	0	0	0	0	0	0

The quantization/dequantization cycle has readily apparent effects. The low-frequency elements near the DC coefficient have been modified, but only by small amounts.

The high-frequency areas of the matrix have, for the most part, been reduced to zero, eliminating their effect on the decompressed image. In this sense, insignificant data has been discarded and the image information has been compressed.

At this point, one might wonder how the values in the quantization matrix are selected. An enormous number of schemes could be used to define these values, and the two most common experimental approaches for testing the effectiveness of such schemes are as follows:

- Measure the mathematical error found between an input image and its output image after it has been decompressed, trying to determine an acceptable level of error.
- Simply "eyeball it". Although judging the effect of decompression on the human eye is purely subjective, it may, in some cases, be more credible than mathematical differences in error levels.

Although JPEG allows for the use of any quantization matrix, ISO has done extensive testing and developed a standard set of quantization values that cause impressive degrees of compression.

Here are some sample images using varying levels of quantization:

Original image:

QF = 75

QF = 20

QF = 5

QF = 3

References

The material here is compiled from the following sources:

Fernando, Udara. Data Compression,
http://www.stanford.edu/~udara/SOCO/lossy/jpeg/algorithm.htm

Pennebaker, W., et. al, (1993). Jpeg: Still Image Data Compression Standard. Kluwer Academic Publishers.

Richardson, Iain. Introduction to Image and Video Coding,
http://www.vcodex.fsnet.co.uk/videocoding2b.pdf

Wolfgang, R. JPEG Tutorial. http://dynamo.ecn.purdue.edu/~ace/jpeg-tut/jpegtut1.html.