The mechanics of the JPEG image encoding system

Douglas A. Kerr

Issue 2
June 7, 2014

ABSTRACT

The JPEG image encoding system provides for a representation of a digital image in far fewer bits that would be required by a "straightforward" representation. The system comprises many ingenious stages. In this article, I describe the working of these stages in considerable detail.

1. INTRODUCTION

1.1 Preamble

The JPEG image encoding system was originally developed under the auspices of a group called the Joint Photographic Experts Group, hence its initialistic name.

The system provides for a representation of a digital image in far fewer bits than would be required by a "straightforward" representation. However, in its most common mode of operation, the encoding is not "reversible"[1]; that is, from the encoded representation (such as we might find in a "JPG" file out of our camera), we cannot precisely reconstruct (for display or printing) the original image. This is the price we pay for a far more economical (in terms of bit storage) representation of the image.

The JPEG encoding system comprises several stages, each of which has a fascinating story and is the result of extremely clever technical work.

But their very cleverness makes them hard to describe concisely but accurately. And they are often given names that are not too helpful (sometimes actually ill-advised).

And we often seek to find "just where the bits are saved", but that quest can be frustrating. We think that a certain stage must do it, but in fact we may find that after that stage the bit load is even increased.

[1] Often described as "not lossless" or "lossy", this based on the notion that an inaccurate reconstruction results from the "loss" of data. In fact, the data is still there; it has just been "inaccurately reproduced".

But it turns out that, like a good magic trick, this is all preparation, and just when we least expect it, the white doves fly out of the hat, and there are far less of them than we had backstage.

In this article, I will try and describe the various stages, just what really happens in each, and where it fits in getting the white doves out of the hat.

1.2 Architecture

The principal stages of the actual JPEG encoding process are these:

- Partitioning of the image data into blocks for encoding.
- Transformation of the color space
- Chroma Subsampling
- Discrete cosine transform (DCT)
- (Re)quantization
- Compaction of the data by zero-run-length encoding and Huffman encoding.
- *Consolidation of the encoded data for all blocks into a single data set.*
- *Composing the output file, whose "payload" is that data set.*

I will discuss in considerable detail each of these except for the ones shown in italics.

2. THE STAGES OF JPEG ENCODING

2.1 Partitioning of the image data into blocks for encoding.

Simplistically speaking, the initial image data is treated in blocks of 8 × 8 pixels. But there are two complications:

- If the dimensions of the image are not multiples of 8, extra "padding" pixels must be added. (These will be shorn off after decoding.)
- As we will see shortly, the chroma aspect of pixel color may only be included for every 2, 4, or even 8 pixels. That aspect is treated in blocks of 8 × 8 values. Such a block may correspond to an "encoding unit" of perhaps 16 × 8, 16 × 16, or even 32 × 16 pixels. To make that work out, we may need to "pad" the image until its dimensions are multiples of the "encoding unit " size.

2.2 Transformation of the Color Space

2.2.1 *The sRGB color space*

The image as presented to the JPEG encoder will be in some recognized color space. For our purposes here I will assume that the color space is the one identified as **sRGB**. In this scheme the color of each image pixel is described by a set of three coordinates, known as R, G, and B. They each tell the amount of three precisely-defined kinds of light (the three primaries), also called R, G, and B (so you can see that there will be ample opportunity for confusion here).

The coordinates R, G, and B all have a range of 0-255 (integer values only). They bear a nonlinear relationship to the "amount" of the primary they indicate (a matter known, for historical reasons, as *gamma precompensation*).

2.2.2 *The sYCC color space*

As the first stage of the JPEG encoding process, we convert the color of each pixel from the sRGB description into a form known as sYCC (which is short for "standard Y'CbCr"). It is something like a *luminance-chrominance* representation, but not quite.

The description of a color in this scheme has two components:

- A value, Y', that is obtained by multiplying the R, G, and B values for the pixel color in the sRGB color space by three different constants and adding the results. It at first seems that if we do this right, this will be the luminance, Y, of the color. But it isn't, because the R, G, and B are nonlinear transforms of the variables telling the amounts of the three primaries. (True luminance, Y, must be determined from the linear values, sometimes designated **r**, **g**, and **b**). This value, Y' is not even a non-linear transform of Y. But it is a nonlinear sort-of luminance.

  It is formally designated Y' to remind us of this important difference from Y (but often the prime is omitted, so be careful). It is often called *luma*, a term borrowed from a somewhat similar situation in the NTSC analog color television system.[2] Sadly, it is often (but incorrectly) called "luminance"

- A two-dimensional (in the mathematical sense) value represented by the values of two variables, Cb and Cr. These are defined thus:

[2] There, *luma* is conveyed by the video carrier proper; it is the "monochrome" signal.

$$Cb = B - Y'$$

$$Cr = R - Y'$$

These together sort-of describe the *chrominance* of the color, but not quite (because they are derived from the non-linearized values R, G, and B). Their symbols do not have a prime to warn us of that, since there is no use of Cb and Cr to mean "aspects of an actual chrominance".

The property defined by these two values is often called *chroma*, again borrowed from analog television practice.[3] Sadly, it is often (but incorrectly) called "chrominance"

Now, what is the point of this? Is less data required to represent the image in this form than in sRGB form? No. In fact, at this stage of the process (inside the JPEG encoder), Y', Cb, and Cr may be represented in more than 8 bits each, so the total number of bits may have escalated!

But having the image in this form allows us to (later in the chain) take advantage, in several ways, of properties of the human visual system, which will help us on the way to reducing the number of bits needed to represent the image.

2.3 Chroma Subsampling

When we left our image at the end of the prior section, each pixel was described in terms of three values, Y' (a sort-of-luminance) and Cb and Cr (together a sort-of-chrominance).

During the work on the NTSC color television system, note was taken of an earlier discovery: that the human eye could note finer changes in luminance (or luma) than in chromaticity (or chrominance, or chroma). Advantage was taken of this in the partitioning of the overall TV channel bandwidth into portions for *luma* and *chroma*. Chroma was given a substantially smaller ration than luma.

In JPEG encoding, we do a wholly-comparable thing. We do not include in the final data package a representation of Cb and Cr for every pixel. Rather, we include a representation of Cb and Cr for only 1 out of 2 pixels, or maybe only 1 out of 4 pixels, or maybe even only

[3] There, *chroma* is conveyed by the *color subcarrier*, which is phase and amplitude modulated to convey the two aspects of the property. It is zero where the image color is "gray" (that is to say, white of some luminance).

1 out of 8 pixels.[4] This then results in a substantial decrease in the overall bit content of the final encoded image.

In fact, this may not be done by just dropping the other Cb and Cr values. Rather, it may be that the Cb and Cr values that are included are each the average of 2, or 4, or 8 original values that are sort-of centered on the pixel that carries a Cb and Cr.

In either case, this process in effect just reduces the pixel resolution of the system for chromaticity (chroma is its proxy) compared to luminance.[5] [6]

The amount of "decimation" can be varied to trade off the reduction in bit load against the reduction of chromaticity resolution and the attendant visual degradation of the image.

The notation for describing the chroma subsampling pattern (and resulting decimation fraction) is a very curious one, with such "values" as 4:4:4, 4:2:2, 4:2:0, and so forth. That is beyond the scope of this article.

2.4 THE DISCRETE COSINE TRANSFORM (DCT)

2.4.1 *Concept of the discrete cosine transform*

If we have a spatial "waveform" (such as the luma of an image across a horizontal track) described by samples spaced at spatial interval *p* (perhaps the pixel pitch), and if the waveform contains no frequencies at or above a spatial frequency of 1/2*p* (that limit is called the Nyquist frequency, f_N), then that set of samples precisely describes the underlying waveform.

So I can use numerical examples, suppose we deal with a fixed-length segment of the waveform, in particular the length such that the number of samples over the length of the waveform is 8. (In fact this is what we usually encounter in JPEG encoding).

We can also describe the waveform by conceptually decomposing it into:

- A constant value ("flat line") of a certain amplitude.

[4] This is sometimes described as a *decimation ratio* of 1:2, 1:4, 1:8, and so forth.

[5] Much the same thing was done earlier in the camera with the use of a CFA sensor. This is however a process wholly separate from that.

[6] The "decimation ratio" to be used is chosen to get the desired balance between (a) reduced bit load and (b) degradation of the representation of chromaticity.

- A cosine function of a certain amplitude with frequency $f_N/8$.
- Cosine functions with certain amplitudes of frequencies $2 \cdot f_N/8$, $3 \cdot f_N/8$, and so forth through $7 \cdot f_N/8$ (that being just beneath the Nyquist limit in this case).

and stating these 8 amplitudes (then known as DCT *coefficients*). This is called the *discrete cosine transform* (DCT) of the original set of 8 samples (and thus of the original waveform). From it, the original set of samples (and the waveform they describe) can be reconstructed.

Those familiar with the similar concept of the discrete Fourier transform (DFT), from which the DCT is descended, might think that since in the DCT we decompose the original "waveform" into only cosine components, not cosine and sine components as we do in a DFT, we have somehow lost the "component phase" information, resulting in an imperfect representation of the original "waveform". However, in the DCT we take cosine components at half the frequency interval of the components in the DFT (leading to the same number of coefficients overall), and the result is that the set of cosine coefficients "perfectly" describes the original "waveform".[7]

By the way, each of those cosine functions is held and handled in sampled form, there being 8 samples over the duration of the original waveform (just as there was for the waveform itself). Thus the moniker "discrete"—it pertains both to the waveform (held as samples) and the cosine functions that form the result of the transform (held as samples).

2.4.2 *Reconstruction of the original "waveform"*

To reconstruct the original set of samples from that description, the "decoder":

- takes each of the coefficients and multiplies it by a cosine function at the corresponding frequency (those each being known in terms of the values of eight samples of the function at intervals *p*), and
- multiplies the values of these eight components for each of the eight points along the length of our "battle zone", thus giving the 8 original sample values (and thus implying the original waveform).

[7] For a source block of 8 values, the DFT will give 4 pairs of coefficients (each with a cosine and sine coefficient), at frequencies of $n \cdot F_N/4$, while the DCT will give 8 cosine coefficients, at frequencies of $n \cdot F_N/8$. Thus in either case, 8 source values result in 8 transform coefficients, a situation of "preservation of degrees of freedom", suggesting that a perfect representation is possible (and it in fact does occur).

2.4.3 *The two-dimensional sample block*

I used a waveform of length 8 samples because, in the most common use of the JPEG encoding algorithm, we work with blocks that comprise 8 × 8 pixels. Again we will work with the luma value (Y') of each pixel.

We could take the first row of 8 pixel values and treat it in just the way I described above for an "8-sample long" waveform, yielding a representation in terms of 8 DCT coefficients. We could then do that (separately) for the 2nd, 3rd, and so forth rows, ending up with 8 sets of 8 coefficients (64 in all) which collectively describe the original set of 64 pixel values. We could show those in an 8 × 8 table, or matrix. Each row would have the 8 coefficients that describe the "waveform" of one row of pixel values.

Recall that in each set of coefficients, the first one (we say "coefficient 0") gives the amplitude of the "zero-frequency" cosine function (the one that is really a constant term), the second one (coefficient 1) gives the amplitude of the cosine function of frequency $1•f_N/8$, the third one (coefficient 2) gives the amplitude of the cosine function of frequency $2•f_N/8$, the fourth one (coefficient 3) gives the amplitude of the cosine function of frequency $3•f_N/8$, and so forth. [8]

2.4.4 *The two-dimensional discrete cosine transform*

But it turns out to be advantageous (for reasons that are beyond the scope of this article) to treat the two-dimensional nature of our little array of 64 pixel values in a consolidated way (using what is called the "two-dimensional DCT"—who would have guessed!).

Again, the result can be presented as an 8 × 8 matrix of coefficients. All of it pertains to the entire 8 × 8 block of pixels (that is, it is not "row per row", as in the example above).

The coefficient in position 2, 1 (third column, second row) prescribes the amplitude of a "two-dimensional" pattern that covers the entire pixel block. It consists of the scaled sum of two cosine functions (each with the same amplitude: that prescribed by the coefficient) that extend over the entire field of the 8 × 8 block of pixels. One, which "runs horizontally" across the field of 8 × 8 pixels (and covers every row), has frequency of $2•f_N/8$. The other, having the same amplitude, "runs vertically" across the field of 8 × 8 pixels (and covers every column), and has frequency of $1•f_N/8$.

[8] One cycle of F_N , by definition, covers a span of two sample pitches.

There are 64 two-dimensional patterns altogether, based on 8 possibilities each for the frequency of their horizontal variation and the frequency of their vertical variation. For each, a coefficient tells us "how strongly" that entire pattern should be applied (not at all, if the coefficient is 0).

In the left-hand part of figure 1 we see a graphic presentation of these 64 patterns (with amplitude 1). The convention is that white means a value of the pattern function of +1; black means a value of -1.

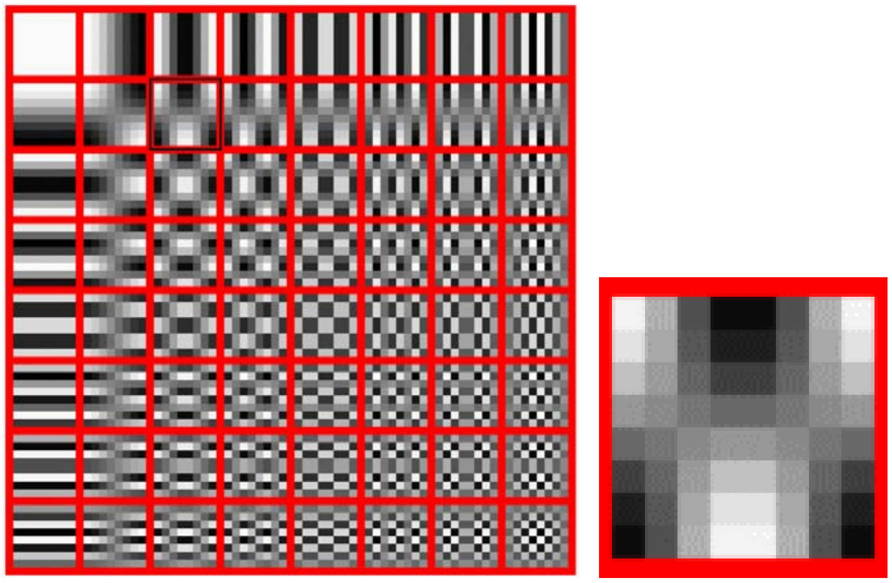

Figure 1. DCT patterns; pattern 2,1.

The scaled summation of all these 64 patterns, each participating with the "potency" dictated by its coefficient (those with zero coefficients not participating at all), form an overall two-dimensional pattern that tells us the pixel luma value at each of the 64 pixel locations in our little "block".

In the right hand part of figure 1, we see an enlargement of the pattern in cell 2,1 (the one discussed earlier). We see that in the horizontal direction, its variation makes 1 cycle across the block (its frequency is $f_N/4$); in the vertical direction, its variation makes 1/2 cycle across the block (its frequency is $f_N/8$).

Note that the patterns always have only 8 different values across the width or height of the block; the cosines, regardless of frequency, are held in terms of 8 values over that span. For example, note that the pattern in cell 0,7 (the lower-left corner) makes, in the vertical direction, 3-1/2 cycles across the height of the block.

But that is hard for us to see, since we see its value at 8 places across the height of the block. This is typical for the sampled representation of a function whose frequency is near the Nyquist frequency (in this case, 7/8 F_N). The samples, nevertheless, represent a pattern that makes 3-1/2 cycles across the height of the block.

Do we fall short of an accurate representation by only recognizing spatial frequency components in the waveform implied by the luma samples up to a frequency of 7•f_N/8? Do we arbitrarily stop there so as not to generate "too many bits"?

No, The waveform implied by the samples cannot contain any component with frequency at or above F_N. (Ah, yes, the Nyquist limit!)

But what if we sampled a waveform that contained a component at, say, 10•f_N/8? Well the resulting set of samples would imply a waveform that had a component at 6•f_N/8. (Ah, yes, that old foldover aliasing!) For better or worse, our DCT scheme would capture that result.

Well, then, suppose we had a set of samples that implied a waveform with a component at, say, 5.5•f_N/8? That can't happen. Our "waveform" has a finite length (that of 8 sample periods), and it can only have components with frequencies of integer multiples of f_N/8 (and up through only 7•f_N/8).

So yes, we have captured the whole enchilada.

We have turned a set of 64 luma values into 64 two-dimensional DCT coefficients, which together represent the very same thing. Has this cut down the amount of data? No. Well, then what was the point of the exercise?

It allows us to do two things (in two later stages of the process) that **will** cut down the amount of data.

2.4.5 *Precision*

At this point the 64 coefficients are probably each represented in 12 bits.

2.5 (Re) quantization

Formally, *quantization* refers to taking information of a "continuous" nature in the value direction (that is, it can actually take on any value, albeit perhaps over a limited range) and expressing it in *discrete form* (where it can only take on certain values, often multiples of some unit). There, it will have an explicit precision ("two decimal places","8 bits"). When we read an analog fever thermometer, and report the result as 98.6° F, we have "on the fly" quantized its reading (to a precision of one decimal place). (The actual temperature might be 98.5853218704... ; we might be able to read it as about 98.55.)

Often we will have information already expressed in discrete form (all the data we work with in JPEG is such) but wish to restate it to a lower precision (generally to reduce the number of digits needed to convey a body of data "which we do not need to know so precisely"). This process is properly called *requantization* (a term we rarely see), but in the JPEG literature it is just spoken of as "quantization". I will nevertheless generally use the rigorous term here.

The 64 coefficients in the output of the DCT (say, for luma) have a certain precision (likely 12 bits). If we reduce the precision, fewer bits will be needed in our final image file, but of course the image will look more coarse. Such artifacts as "posterization" and "banding" are in fact manifestations of "excessively coarse" quantization of luma.

But the human eye is not equally sensitive to the "coarseness" of reproducing luminance for all spatial frequencies. Generally speaking, it is more sensitive for the lower spatial frequencies, and less sensitive for the higher spatial frequencies.

Recall now that we have the "cosine" description of the luma of our little 8 × 8 pixel block organized by spatial frequency (in a vertical/horizontal combination pattern way) in the 64 cells of the DCT output table.

So we could set up a plan in which the values near the upper-left corner (lower frequencies in both directions) are held to a high precision, while the values near the lower-right corner (lower frequencies in both directions) are reduced to a much lower precision, and those near the lower-left and upper-right corners (high frequency in one direction, low frequency in the other). Those near the center (moderate frequency in each direction) are reduced to an intermediate precision, and so forth.

And so we do just that.

The re-quantization is controlled by a *quantizing table*. Figure 2 shows a commonly-used one for luminance:

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

Figure 2. Base luminance quantizing table

(This one is used for re-quantization of the luma values; a different table is used for re-quantization of the chroma values.)

The value in each cell (the *quantization control value*[9], *q*) controls the new quantization of the DCT coefficient found in the corresponding cell of the DCT output matrix. The larger the value, the more is the precision reduced. A value of 1 keeps the precision of the value as is. The higher the value, the more "coarse" is the new quantization (remember, the value was already quantized!). A value of 2 makes the precision "twice as coarse" as it was.

To understand how this works, we will look at an example with decimal numbers. Suppose that as part of a set of data we have the number 4325 (expressed to a precision of "units"). We wish to round (re-quantize) that value to "hundreds". Thus our quantizing control value, *q*, would be 100.

First we divide the value by *q* (100): 4325/100 = 43.25.

Then we round that to the nearest integer and get 43.[10]

Note that the value is now represented to lesser precision than before and **scaled down in size**.

Note that at this point, the value can be represented by fewer decimal digits (fewer bits) than before. We tend to say that this is because of the "more coarse precision", and philosophically it is, but as to the actual mechanics, it is a result of the scaling down—the number is just plain smaller!

Then we multiply by *q* (100): 43•100 = 4300.

So 4300 is our value (now "rounded" to hundreds).

> Note that at this point, despite its "lesser precision", the value in this condition takes as many decimal digits (or bits) to represent it as before (at least in the obvious way). Hold that thought.

Now suppose our reason for making the value "more coarse" is to save data in transmitting our data collection to a distant point. In that case, we can omit the last step (multiplication by *q*) and just send

[9] My term.

[10] There are of course several ways to do that, and we need not be too concerned with that here. For example, one way is to add 50 before dividing by 100, and then just drop the fractional part. I didn't describe that because it obscures the real thing we want to do.

"43" to the distant end (saving the transmission of two digits, "00", worth about 6.6 bits). The distant end knows that this kind of number is "in hundreds", so it interprets the received number as 4300.

> Note that the reason we can uses fewer digits is not that the representation is "more coarse". It is because the value is smaller!

Now, looking at the table, and considering the lower right-hand cell, we see a *q* value of 99. That means that after re-quantization, the value of that coefficient is:

- 1/99 its original size
- 99 times as "coarse" (**relative to its scale**), as if its "unit" were now 99 times as big as before. Actually, its unit is still 1 (always is), but because the value is 1/99 its original size, in effect the unit is 99 times as big as before.

But if we have done this so we can represent the value in fewer bits than before, it is the change in its size—not the greater "coarseness of its representation, *per se*— that brings that about!

By the way, the quantizing table is not fixed by the JPEG specification (although the one shown above is "suggested" there). The manufacturer of the JPEG encoder may choose a different table, based on its own experience with file size-performance tradeoffs.

And in any case, the user of the application or camera may be given the opportunity to change the quantizing plan. A more coarse quantizing plan will result in a smaller overall file size for the image, but will typically increase the degree of imperfection in the reconstructed image. This choice is typically made by the user by way of a "JPEG quality" control in the application. (In the case of JPEG files from a camera, this may be a camera menu setting.)

This variability as to the quantizing table will not confuse the receiving JPEG decoder, since the quantizing table actually used for the particular image (wherever it came from) is explicitly described in the file "header". The *q* values in it are used by the receiver to multiply the received coefficient values at the receiving end to actually complete the re-quantizing process (as when in the decimal example we finally got to "4300").[11]

[11] In fact, in the ITU JPEG specification, this process is called "dequantification", which is not at all apt (it is actually just "descaling"—the number is still of "reduced precision").

A common scheme of creating different quantizing tables (to suit different user wishes as to the file size vs. image "quality" tradeoff) is to start with a base table (to be used as is for some "midrange" quality) and then scale all the *q* values in it up or down to make other tables (with their own "quality" implications.

Often the base table is in fact the one suggested by the JPEG specification.

How do we tell the application how to scale the quantizing table? There are many ways. One way defined by the Independent JPEG Group (which provides a library of functions for use in constructing JPEG encoders and decoders) uses as the "knob" a *quality level*,[12] *Q*, which can potentially be given any value from 1-100. A fixed (but nonlinear) algorithm turns any value of *Q* into a value of a scaling level, *S*. Then, all the *q* values of the base table are multiplied by a single scaling factor, which is S/100.

The algorithm is such that for various values of *Q* (chosen by the user), the resulting scaling factor is as shown in the table of figure 3:

Quality setting, Q	**Scaling factor, S/100**
1	50
10	1.9
50	1.0
75	0.5
90	0.2
100	0*

* But the actual scaling process when S/100 = 0 leads to all *q* values being 1, making the "do not requantize" table.

Figure 3. Table scaling based on "Q".

We see that a quality (Q) setting of 50 results in a table just like the base table. Thus we can say that the base table produces "quality 50" quantization. (How "good" is that in terms of the reconstructed image? There is no simple way to describe that.)

In figure 4, we see a table made from the base table we saw above with a Q setting of 80 (so S will be 40, and all the *q* values will be 0.40 times the values in the base table).

[12] My term.

6	4	4	6	10	16	20	24
5	5	6	8	10	23	24	22
6	5	6	10	16	23	28	22
6	7	9	12	20	35	32	25
7	9	15	22	27	44	41	31
10	14	22	26	32	42	45	37
20	26	31	35	41	48	48	40
29	37	38	39	45	40	41	40

Figure 4. Scaled luminance quantizing table (Q = 80)

But not all applications work that way. Some may offer "JPEG quality settings" from 1-12, and some may offer JPEG quantity settings on a 1-100 scale (but that does not necessarily correspond to values of the quality control variable Q I spoke of above). Or in cameras we may have two or three choices with names like "normal", "fine", and "superfine".

The question of "how would we describe the image degradation resulting from various values of Q" is a complex one, beyond the scope of this article. But note that in today's digital cameras, for the "best" JPEG quality setting, the value of Q (for both luma and chroma re-quantization) is typically made about 95.

In any event, although we can't see just yet how this happens mechanically, we can readily imagine that making the quantizing "more coarse" should somehow result in fewer bits being needed to convey the 64 DCT coefficients that describe each 8 × 8 pixel block of our image. Where that actually pays off is in the next stage.

2.6 Entropy Encoding

2.6.1 *The strategy*

At this point in the overall process we have for each 8 × 8 block of pixel luma values a set of 64 DCT coefficients, each held in a fixed number of bits. By a very ingenious process, we are able to draw upon certain statistical properties of the set of 64 values to help us precisely represent them in far fewer bits (overall) than in their present form.

2.6.2 *The "DC coefficient"*

The coefficient in location 0,0 of the table gives the amplitude of the two-dimensional pattern which is a constant in both directions because its two cosine functions both have zero frequency. Borrowing a term from electrical engineering for a zero-frequency component ("direct current", or DC), this is called the "DC coefficient"[13]. It is in effect the average of the luma values over the entire block.

Of course, this value does not typically vary rapidly from block to block.

Thus, in the "final result" for each block, we record the difference in the DC coefficient for this block from that of the previous block. The representation is called *differential pulse code modulation* (DPCM), a term mindlessly borrowed from digital audio encoding practice (where it already doesn't makes much sense).

These differences can be, on average, encoded in fewer bits than would be required for the actual values.

2.6.3 *The "AC coefficients"*

Since the 63 remaining coefficients describe cosine patterns, they are, by analogy to electrical engineering concepts, of an "alternating current" (AC) nature, and thus are spoken of collectively as the "AC coefficients."[13]

2.6.4 *Encoding*

A very clever scheme is used to encode this suite of 64 values. It depends in part on the fact that there are usually a lot of zero values in the suite, and they tend to be concentrated toward the coefficients in the lower-right portion of the table (those for patterns with higher vertical and horizontal frequencies).

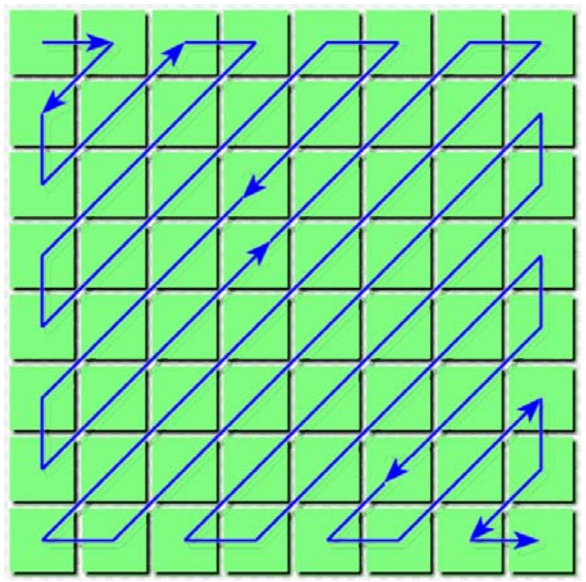

Figure 5. DCT coefficient zig-zag readout path

Adapted from an illustration by Alex Kristov
via Wikimedia Commons

[13] No, as an electrical engineer, this does not delight me.

We start by making a single string of all 64 of those values, taking them in order from the upper left corner and working overall toward the lower right along a zigzag trail. Figure 5 shows the path.

Having done that, we find that, especially in the later part of the string, we may have many groups of consecutive zeros. And we exploit this in our encoding.

Basically, in the output of this stage, which is another string, we only actually include the non-zero values. But we prefix each one (other than the first one—the "DC" coefficient) with a four-bit field that says "by the way, there were *n* zero values before this non-zero value; be sure and put those back in when this is decoded". (Of course for the first coefficient, there could be no preceding zero-value coefficients, thus no need for that field.) This is described in the JPEG documentation as "run-length encoding". (It is in fact a very specialized version of such, only relating to runs of one value: zero.)

This process is an important contributor to the reduction in the number of bits needed to covey all the DCT coefficients.

Now, we squeeze a little more blood out of the onion. Non-zero values of different sizes can be represented in different numbers of bits (just as we can represent the decimal value "23" in two decimal digits but need four digits for the value "4108".

Many of the coefficients have small values, so it is wasteful to allocate some fixed number of bits in the string for each non-zero value. So we only allocate the number of bits needed, but, so the "recipient" can keep track of what is going on in this string of bits, we prefix the actual value bits with a four-bit field that tells "how many bits (right after me) carry the actual value".

Thus, in this new string, for each non-zero value, we have a group of bits we can think of as representing three numbers, thus:

ZLV

Where Z (four bits) tells how many zero values there were preceding this non-zero value (which need to be put back in during decoding), L (four bits) tells how many bits that follow give the actual value, and V (that many bits) is the actual value.[14]

(Can't you imagine how much fun this must have been to invent!)

[14] V (a signed value) is given in a peculiar (and mathematically-handy) variant of "ones-complement" notation. The peculiar version is possible because we know the range in which the value falls (because of the clue given by L).

In the case of the first (DC) coefficient, the bit group is only LV.

There are two special provisions. If there are 16 or more consecutive zero values preceding a non-zero value, that cannot be represented in the usual way (since the maximum possible value of Z is 15). Rather, one or more runs of 16 consecutive zero values are represented by a special ZL sequence, 15,0 (called "zrl"), which means "16 zero values not followed by a non-zero value."

If, at a certain point in the value sequence, there are only zero values for the rest of the block, they are all represented by a special ZL sequence (called "eob"), 0,0. This means "all further values up to the total number expected for this block are zero."

In each of those cases, the "string parser" does not expect a V component; an L value of 0 means "number of bits for V is: zero".

Now, at this point, we already have the beginning of the payoff in terms of a reduction in the number of bits to represent the overall image. It comes about from the re-quantization of the DCT coefficients. That's because the re-quantization does not just make the representation coarser, but the value (at the point where we stop the process) is actually smaller, and thus can be represented in fewer bits.

And as we have just seen, in the "final" data string, in fact only the number of bits absolutely necessary are used for each coefficient value. So we already collect on some of the promise of bit load reduction. In particular, if the number of bits required to represent the value is less than 8, then that number of bits, plus 4 bits for L, will be less than 12 (the number of bits to represent every coefficient if we didn't use this scheme.

Of course for values requiring greater than 8 bits to represent, the bit load will be greater than for straightforward representation. But since so many values are fairly small, there is ordinarily a substantial net saving.

But we're not done yet in our quest for a smaller file. We take one more step.

If we interpret the 8 bits in Z+L[15,16]as a single number (although as such it has no actual meaning, like the concatenation of your house number and your age), we find that the different values we encounter

[15] I use "+" here to symbolize concatenation, not addition.

[16] In the case of the first (DC) coefficient, we only work with the four bit number L (there is no Z).

are not present with comparable frequency. Certain 8-bit values are much more common than certain others.

When we have such a situation in a set of data, we have the opportunity to encode these 8-bit values with an average, over the entire image, of less than 8 bits per value. Here this is done by way of a Huffman code[17], which assigns codes of different lengths to different values.

The values known (or assumed) to occur more frequently are given shorter codes; the values known (or assumed) to occur less frequently are given longer codes. If the assumption of relative frequency of occurrence in fact plays out, the total number of bits required for the Z+L "values" in the entire image is less than if every Z+L "value" were left encoded in 8 bits.

The "table" that assigns codes to the various possible values can be based on an assumption as to the relative frequency of occurrence of the different possible values, and used for every image. A suggested table on this basis is provided in the JPEG specification, and any JPEG decoder is able to work with images encoded on that premise.

More compact coding can be usually be achieved with a table that is optimized for the actual frequencies of occurrence of various values as they occur in the particular image (a "preliminary survey pass" being made by the encoder to get the statistics on which to base the table design). That table is then included in the file "header" for the benefit of the decoder.

By the way, Huffman coding is "reversible"; that is, the original values are precisely recoverable from the Huffman-encoded form. (The same is true of the alternative, *arithmetic coding*.)

2.6.5 *The term "entropy encoding"*

Codes of the Huffman type are often said to use "entropy encoding', because they seek to match the number of bits used to represent each data item to the true "amount of information" in it (known in information theory as the *entropy* of the item, a term borrowed from the field of thermodynamics). The same is true of arithmetic coding.

Because one part of the encoding process of this stage is done with a Huffman code (or arithmetic code), the whole process is called in the JPEG literature "entropy encoding".

---

[17] Although the JPEG standard provides for an alternate system, called *arithmetic coding*, which is beyond the scope of this article.

2.7 What about chroma?

So far, after the second stage, we have spoken specifically only of processing the luma aspect of the image information.

In fact, there are parallel processes, almost identical, conducted for Cb and (separately) Cr. There are a few differences, most prominently:

- The block of **values** used is again most commonly 8 × 8, but this will not generally correspond to an 8 × 8 block of **pixels** owing to chroma subsampling.

- A different quantization table is used. Again, there is a "suggested" base table in the JPEG specifications, and again tables with differing quantization may be constructed with the algorithm running on the JPEG *quality control value*, Q. The same value of Q might not be used to scale both tables.

2.8 A curiosity

The luma value, Y' is not a signed quantity (it is always positive), but for various reasons (to some extent consistency with how Cb and Cr, which are signed, are represented) it is arbitrarily represented with an offset. It is basically encoded in 11 bits, with 1024 being subtracted first.

We would think that the highest possible value (which would come from R,G,B = 255,255,255) would be 2047, or 1023 after the offset.

The lowest value, which would come from R,G,B = 0,0,0), would be 0, or −1024 after the offset.

But in practice, the highest encoded Y' value (as reported by utility programs that decode the JPEG file) is 1016 (the lowest is −1024 as we would expect).

I don’t understand all I know about this.

3. WRAPPING UP

3.1 Consolidation

Next, the final bit strings from all the blocks in each encoding unit are consolidated, and then all the bit strings from all encoding units are consolidated into a single byte-organized data stream. This is fairly trivial in principle, but there are some tricky details. I will not here discuss the process.

3.2 The output file

What we have so far is a set of bits that describe the image in JPEG-encoded form. But to be able to use this, we normally put it into a file. There are several standardized formats for such a file, including JFIF and DCT. These file structures have to carry the JPEG-encoded image proper, such vital collateral information as the quantizing tables and Huffman encoding tables used in the process, and various types of metadata related to the image as such. The formats are very complicated. I will not here discuss this stage of the overall process.

4. BENEDICTION

Thanks to Carla Red Fox for her insightful copy editing of this difficult manuscript.

This work is dedicated to her on the occasion of our 15th wedding anniversary, June 12, 2014.

#