Introduction to Data Compression (Part II)

Guy E. Blelloch*

September 26, 2000

Contents

1 Lossy Compression Techniques

Lossy compression is compression in which some of the information from the original message sequence is lost. This means the original sequences cannot be regenerated from the compressed sequence. Just because information is lost doesn't mean the quality of the output is reduced. For example, random noise has very high information content, but when present in an image or a sound file, we would typically be perfectly happy to drop it. Also certain losses in images or sound might be completely imperceptible to a human viewer (e.g. the loss of very high frequencies). For this reason, lossy compression algorithms on images can often get a factor of 2 better compression than lossless algorithms with an imperceptible loss in quality. However, when quality does start degrading in a noticeable way, it is important to make sure it degrades in a way that is least objectionable to the viewer (*e.g.*, dropping random pixels is probably more objectionable than dropping some color information). For these reasons, the way most lossy compression techniques are used are highly dependent

*Some of the text and many of the figures in here are from scribe notes taken by Ben Liblit. This is a first pass, while PART I was a second pass, so there is likely to be more typos in here (I apologize).

on the media that is being compressed. Lossy compression for sound, for example, is very different than lossy compression for images.

In this section we go over some general techniques that can be applied in various contexts, and in the next two sections we go over more specific examples and techniques.

1.1 Scalar Quantization

A simple way to implement lossy compression is to take the set of possible messages S and reduce it to a smaller set S' by mapping each element of S to an element in S'. For example we could take 8-bit integers and divide by 4 (*i.e.*, drop the lower two bits), or take a character set in which upper and lowercase characters are distinguished and replace all the uppercase ones with lowercase ones. This general technique is called *quantization*. Since the mapping used in quantization is many-to-one, it is irreversible and therefore lossy.

In the case that the set S comes from a total order and the total order is broken up into regions that map onto the elements of S', the mapping is called *scalar quantization*. The example of dropping the lower two bits given in the previous paragraph is an example of scalar quantization. Applications of scalar quantization include reducing the number of color bits or gray-scale levels in images (used to save memory on many computer monitors), and classifying the intensity of frequency components in images or sound into groups (used in JPEG compression). In fact we mentioned an example of quantization when talking about JPEG-LS. There quantization is used to reduce the number of contexts instead of the number of message values. In particular we categorized each of 3 gradients into one of 9 levels so that the context table needs only 9^3 entries (actually only $(9^3 + 1)/2$ due to symmetry).

The term *uniform scalar quantization* is typically used when the mapping is linear. Again, the example of dividing 8-bit integers by 4 is a linear mapping. In practice it is often better to use a *nonuniform scalar quantization*. For example, it turns out that the eye is more sensitive to low values of red than to high values. Therefore we can get better quality compressed images by making the regions in the low values smaller than the regions in the high values. Another choice is to base the nonlinear mapping on the probability of different input values. In fact, this idea can be formalized—for a given error metric and a given probability distribution over the input values, we want a mapping that will minimize the expected error. For certain error-metrics, finding this mapping might be hard. For the root-mean-squared error metric there is an iterative algorithm known as the Lloyd-Max algorithm that will find the optimal mapping. An interesting point is that finding this optimal mapping will have the effect of decreasing the effectiveness of any probability coder that is used on the output. This is because the mapping will tend to more evenly spread the probabilities in S'.

1.2 Vector Quantization

Scalar quantization allows one to separately map each color of a color image into a smaller set of output values. In practice, however, it can be much more effective to map regions of 3-d color space into output values. By more effective we mean that a better compression ratio can be achieved based on an equivalent loss of quality.

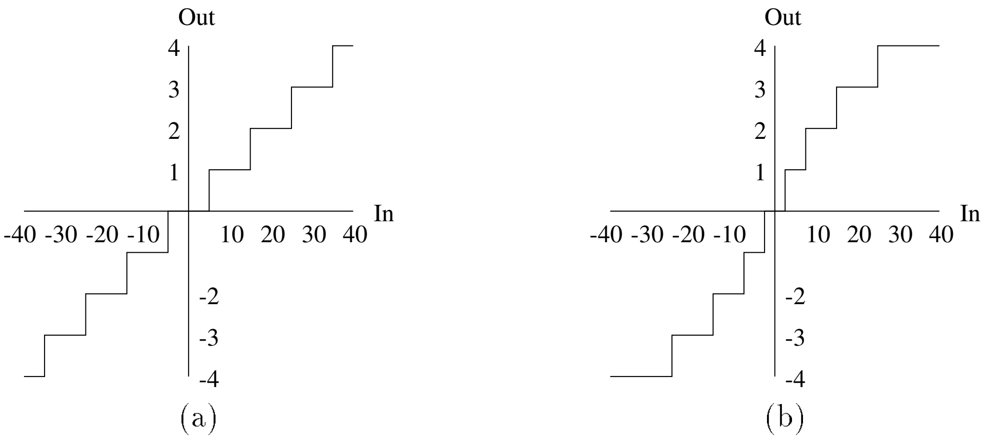

Figure 1: Examples of (a) uniform and (b) non-uniform scalar quantization.

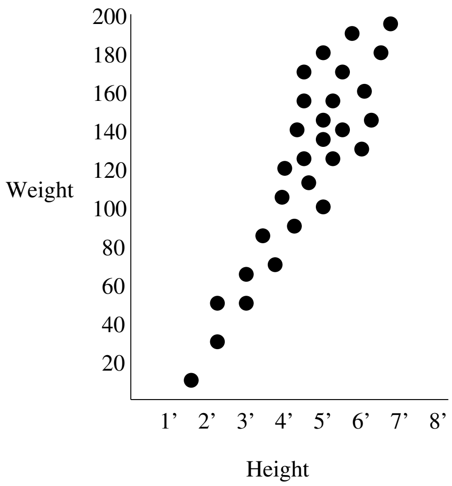

Figure 2: Example of vector-quantization for a height-weight chart.

The general idea of mapping a multidimensional space into a smaller set of messages S' is called *vector quantization*. Vector quantization is typically implemented by selecting a set of representatives from the input space, and then mapping all other points in the space to the closest representative. The representatives could be fixed for all time and part of the compression protocol, or they could be determined for each file (message sequence) and sent as part of the sequence. The most interesting aspect of vector quantization is how one selects the representatives. Typically it is implemented using a clustering algorithm that finds some number of clusters of points in the data. A representative is then chosen for each cluster by either selecting one of the points in the cluster or using some form of centroid for the cluster. Finding good clusters is a whole interesting topic on its own.

Vector quantization is most effective when the variables along the dimensions of the space are correlated. Figure 2 gives an example of possible representatives for a height-weight chart. There is clearly a strong correlation between people's height and weight and therefore the representatives can be concentrated in areas of the space that make physical sense, with higher densities in more common regions. Using such representatives is very much more effective than separately using scalar quantization on the height and weight.

We should note that vector quantization, as well as scalar quantization, can be used as part of a lossless compression technique. In particular if in addition to sending the closest representative, the coder sends the distance from the point to the representative, then the original point can be reconstructed. The distance is often referred to as the residual. In general this would not lead to any compression, but if the points are tightly clustered around the representatives, then the technique can be very effective for lossless compression since the residuals will be small and probability coding will work well in reducing the number of bits.

1.3 Transform Coding

The idea of transform coding is to transform the input into a different form which can then either be compressed better, or for which we can more easily drop certain terms without as much qualitative loss in the output. One form of transform is to select a linear set of basis functions (ϕ_i) that span the space to be transformed. Some common sets include sin, cos, polynomials, spherical harmonics, Bessel functions, and wavelets. Figure 3 shows some examples of the first three basis functions for discrete cosine, polynomial, and wavelet transformations. For a set of n values, transforms can be expressed as an $n \times n$ matrix T. Multiplying the input by this matrix T gives, the transformed coefficients. Multiplying the coefficients by T^{-1} will convert the data back to the original form. For example, the coefficients for the discrete cosine transform (DCT) are

$$T_{ij} = \begin{cases} \sqrt{1/n}\cos\frac{(2j+1)i\pi}{2n} & i = 0, 0 \leq j < n \\ \sqrt{2/n}\cos\frac{(2j+1)i\pi}{2n} & 0 < i < n, 0 \leq j < n \end{cases}$$

The DCT is one of the most commonly used transforms in practice for image compression, more so than the discrete Fourier transform (DFT). This is because the DFT assumes periodicity, which is not necessarily true in images. In particular to represent a linear function over a region requires many large amplitude high-frequency components in a DFT. This is because the periodicity assumption will view the function as a sawtooth, which is highly discontinuous at the teeth requiring the high-frequency components. The DCT does not assume periodicity and will only require much lower amplitude high-frequency components. The DCT also does not require a phase, which is typically represented using complex numbers in the DFT.

For the purpose of compression, the properties we would like of a transform are (1) to decorrelate the data, (2) have many of the transformed coefficients be small, and (3) have it so that from the point of view of perception, some of the terms or more important than others.

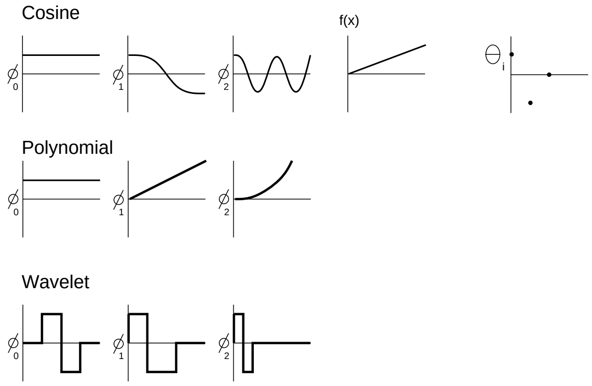

Figure 3: Transforms

2 A Case Study: JPEG and MPEG

The JPEG and the related MPEG format make good real-world examples of compression because (a) they are used very widely in practice, and (b) they use many of the compression techniques we have been talking about, including Huffman codes, arithmetic codes, residual coding, run-length coding, scalar quantization, and transform coding. JPEG is used for still images and is the standard used on the web for photographic images (the GIF format is often used for textual images). MPEG is used for video and after many years of debated MPEG-2 has become the standard for the transmission of high-definition television (HDTV). This means in a few years we will all be receiving MPEG at home. As we will see, MPEG is based on a variant of JPEG (i.e. each frame is coded using a JPEG variant). Both JPEG and MPEG are lossy formats.

2.1 JPEG

JPEG is a lossy compression scheme for color and gray-scale images. It works on full 24-bit color, and was designed to be used with photographic material and naturalistic artwork. It is not the ideal format for line-drawings, textual images, or other images with large areas of solid color or a very limited number of distinct colors. The lossless techniques, such as JBIG, work better for such images.

JPEG is designed so that the loss factor can be tuned by the user to tradeoff image size and image quality, and is designed so that the loss has the least effect on human perception. It however does have some anomalies when the compression ratio gets high, such as odd effects across the boundaries of 8x8 blocks. For high compression ratios, other techniques such as wavelet compression appear to give more satisfactory results.

An overview of the JPEG compression process is given in Figure 4. We will cover each of the steps in this process.

The input to JPEG are three color planes of 8-bits per-pixel each representing Red, Blue and Green (RGB). These are the colors used by hardware to generate images. The first step of JPEG compression, which is optional, is to convert these into YIQ color planes. The YIQ color planes are designed to better represent human perception and are what are used

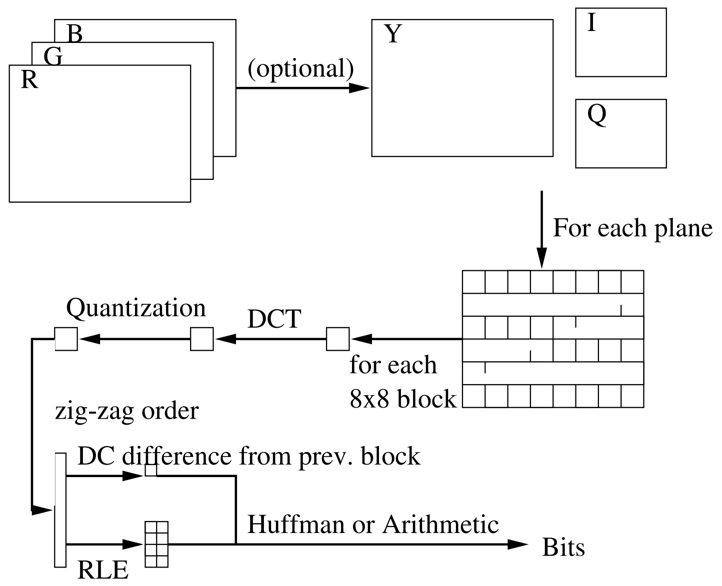

Figure 4: Steps in JPEG compression.

on analog TVs in the US (the NTSC standard). The Y plane is designed to represent the brightness (luminance) of the image. It is a weighted average of red, blue and green (0.59 Green + 0.30 Red + 0.11 Blue). The weights are not balanced since the human eye is more responsive to green than to red, and more to red than to blue. The I (interphase) and Q (quadrature) components represent the color hue (chrominance). If you have an old black-and-white television, it uses only the Y signal and drops the I and Q components, which are carried on a sub-carrier signal. The reason for converting to YIQ is that it is more important in terms of perception to get the intensity right than the hue. Therefore JPEG keeps all pixels for the intensity, but typically down samples the two color planes by a factor of 2 in each dimension (a total factor of 4). This is the first lossy component of JPEG and gives a factor of 2 compression: $(1 + 2 * .25)/3 = .5$.

The next step of the JPEG algorithm is to partition each of the color planes into 8x8 blocks. Each of these blocks is then coded separately. The first step in coding a block is to apply a cosine transform across both dimensions. This returns an 8x8 block of 8-bit frequency terms. So far this does not introduce any loss, or compression. The block-size is motivated by wanting it to be large enough to capture some frequency components but not so large that it causes "frequency spilling". In particular if we cosine-transformed the whole image, a sharp boundary anywhere in a line would cause high values across all frequency components in that line.

After the cosine transform, the next step applied to the blocks is to use uniform scalar quantization on each of the frequency terms. This quantization is controllable based on user parameters and is the main source of information loss in JPEG compression. Since the human eye is more perceptive to certain frequency components than to others, JPEG allows the quantization scaling factor to be different for each frequency component. The scaling factors are specified using an 8x8 table that simply is used to element-wise divide the 8x8 table of frequency components. JPEG defines standard quantization tables for both

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

Table 1: JPEG default quantization table, luminance plane.

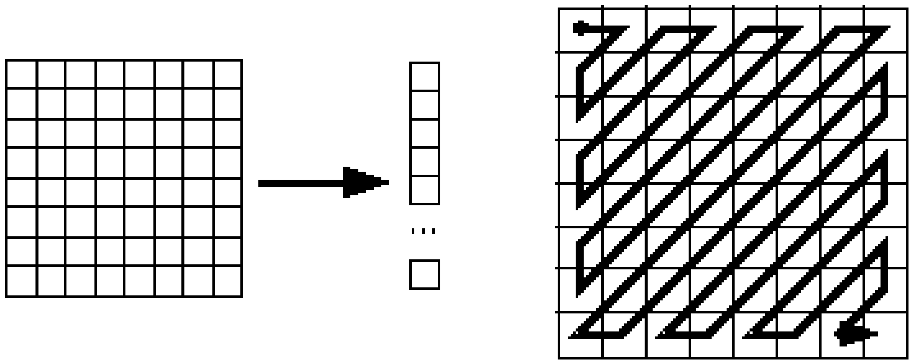

Figure 5: Zig-zag scanning of JPEG blocks.

the Y and I-Q components. The table for Y is shown in Table 1. In this table the largest components are in the lower-right corner. This is because these are the highest frequency components which humans are less sensitive to than the lower-frequency components in the upper-left corner. The selection of the particular numbers in the table seems magic, for example the table is not even symmetric, but it is based on studies of human perception. If desired, the coder can use a different quantization table and send the table in the head of the message. To further compress the image, the whole resulting table can be divided by a constant, which is a scalar "quality control" given to the user. The result of the quantization will often drop most of the terms in the lower left to zero.

JPEG compression then compresses the DC component (upper-leftmost) separately from the other components. In particular it uses a difference coding by subtracting the value given by the DC component of the previous block from the DC component of this block. It then Huffman or arithmetic codes this difference. The motivation for this method is that the DC component is often similar from block-to-block so that difference coding it will give better compression.

The other components (the AC components) are now compressed. They are first converted into a linear order by traversing the frequency table in a zig-zag order (see Figure 5).

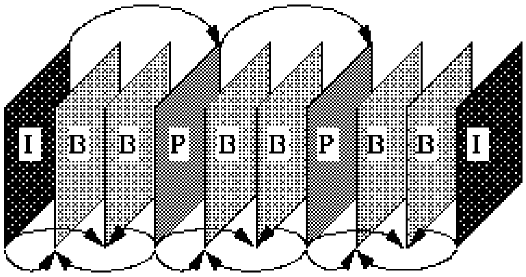

Figure 6: MPEG ordering of I-, P-, and B-frames.

The motivation for this order is that it keeps frequencies of approximately equal length close to each other in the linear-order. In particular most of the zeros will appear as one large contiguous block at the end of the order. A form of run-length coding is used to compress the linear-order. It is coded as a sequence of (skip,value) pairs, where skip is the number of zeros before a value, and value is the value. The special pair (0,0) specifies the end of block. For example, the sequence [4,3,0,0,1,0,0,0,1,0,0,0,...] is represented as [(0,4),(0,3),(2,1),(3,1),(0,0)]. This sequence is then compressed using either arithmetic or Huffman coding. Which of the two coding schemes used is specified on a per-image basis in the header.

2.2 MPEG

Correlation improves compression. This is a recurring theme in all of the approaches we have seen; the more effectively a technique is able to exploit correlations in the data, the more effectively it will be able to compress that data.

This principle is most evident in MPEG encoding. MPEG compresses video streams. In theory, a video stream is a sequence of discrete images. In practice, successive images are highly interrelated. Barring cut shots or scene changes, any given video frame is likely to bear a close resemblance to neighboring frames. MPEG will exploit this strong correlation to achieve far better compression rates than would be possible with isolated images.

Each frame in an MPEG image stream is encoded using one of three schemes:

I-frame , or intra-frame, are coded as isolated images.

P-frame , or predictive coded frame, are based on the previous I- or P-frame.

B-frame , or bidirectionally predictive coded frame, are based on either or both the previous and next I- or P-frame.

Figure 6 shows an MPEG stream containing all three types of frames. I-frames and P-frames appear in an MPEG stream in simple, chronological order. However, B-frames are moved so that they appear *after* their neighboring I- and P-frames. This guarantees that each frame appears after any frame upon which it may depend. An MPEG encoder can decode any frame by buffering the two most recent I- or P-frames encountered in the data

Playback order:	0	1	2	3	4	5	6	7	8	9
Frame type:	I	B	B	P	B	B	P	B	B	I
Data stream order:	0	2	3	1	5	6	4	8	9	7

Table 2: MPEG B-frames postponed in data stream.

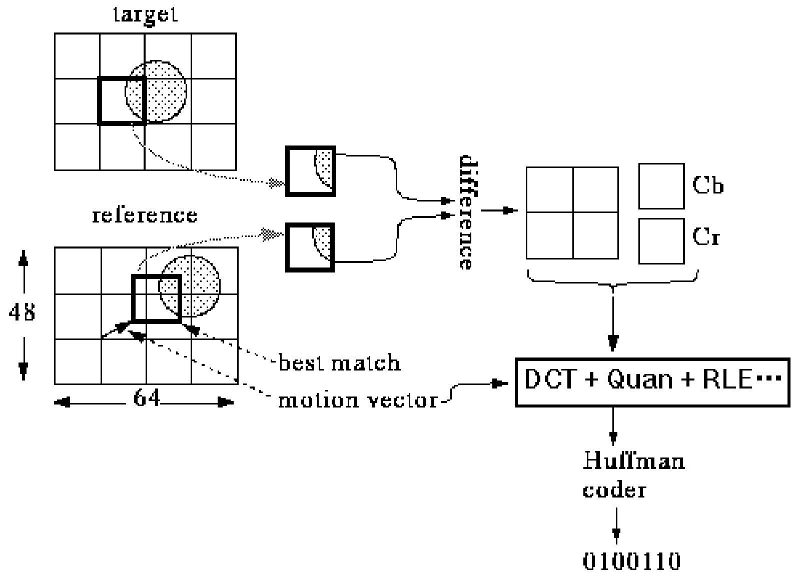

Figure 7: P-frame encoding.

stream. Table 2 shows how B-frames are postponed in the data stream so as to simplify decoder buffering. MPEG encoders are free to mix the frame types in any order. When the scene is relatively static, P- and B-frames could be used, while major scene changes could be encoded using I-frames. In practice, most encoders use some fixed pattern like the ones shown in Figure 6.

Because I-frames are independent, an I-frame can be decoded quickly and simply, without requiring additional information from elsewhere in the MPEG stream. Thus, I-frames form anchor points for fast random access, reverse playback, error recovery, and the like. I-frames are encoded using a variant of JPEG (the color transformation and quantization steps are slightly different).

P-frames are based on the previous P- or I-frame by trying to find a match in the previous image and then coding the difference between this image and the previous match. The matches are based on 16x16 *macroblocks*. The job of the encoder is to find for each *target* block in the current image a *reference* block in the previous image that most closely matches it. This could be implemented by trying all blocks and finding the best match, or more likely in practice, to search some neighborhood for the best match. The reference block need not be aligned on a 16-pixel boundary. An example is shown in Figure 7. The motivation for searching for nearby blocks for a match is that if there is a moving object in the sequence of images (e.g. a car or a ball), or if the whole video is panning, then the match will not be in the same location in the image. Once a reference block is found, the coder needs to send both the motion vector (the x-y displacement from the target to the reference), and the difference between the target and reference blocks. This difference is coded similarly to a JPEG image. The motion vector is Huffman coded. If no good match is found, then the block is coded as if it were from an I-frame.

In practice, the search for a good match for each target macroblock is the most computationally difficult part of MPEG encoding. With current technology, real-time MPEG encoding is only possible with the help of custom hardware. Note, however, that while the *search* for a match is expensive, regenerating the image as part of the decoder is cheap since the decoder is given the motion vector and only needs to look up the block from the previous image.

B-frames were not present in MPEG's predecessor, H.261. They were added in an effort to address the following situation: portions of an intermediate P-frame may be completely absent from all previous frames, but may be present in future frames. For example, consider a car entering a shot from the side. Suppose an I-frame encodes the shot before the car has started to appear, and another I-frame appears when the car is completely visible. We would like to use P-frames for the intermediate scenes. However, since no portion of the car is visible in the first I-frame, the P-frames will not be able to "reuse" that information. The fact that the car is visible in a later I-frame does not help us, as P-frames can only look *back* in time, not forward.

B-frames look for reusable data in both directions. The overall technique is very similar to that used in P-frames, but instead of just searching in the previous I- or P-frame for a match, it also searches in the next I- or P-frame. Assuming a good match is found in each, the two reference frames are averaged and subtracted from the target frame. If only one good match is found, then it is used as the reference. The coder needs to send some information on which reference(s) is (are) used, and potentially needs to send two motion vectors.

How effective is MPEG compression? We can examine typical compression ratios for each frame type, and form an average weighted by the ratios in which the frames are typically interleaved.

Starting with a 356×260 pixel, 24-bit color image, typical compression ratios for MPEG-I are:

Type	Size	Ratio
I	18 Kb	7:1
P	6 Kb	20:1
B	2.5 Kb	50:1
Avg	4.8 Kb	27:1

If one 356×260 frame requires 4.8 Kb, how much bandwidth does MPEG require in order to provide a reasonable video feed at thirty frames per second?

$$30 frames/sec \cdot 4.8 Kb/frame \cdot 8b/bit = 1.2 Mbits/sec$$

Thus far, we have been concentrating on the visual component of MPEG. Adding a stereo audio stream will require roughly another 0.25 Mbits/sec, for a grand total bandwidth of 1.45 Mbits/sec.

This fits nicely within the 1.5 Mbit/sec capacity of a T1 line. In fact, this specific limit was a design goal in the formation of MPEG. Real-life MPEG encoders track bit rate as they encode, and will dynamically adjust compression qualities to keep the bit rate within some user-selected bound. This bit-rate control can also be important in other contexts. For example, video on a multimedia CD-ROM must fit within the relatively poor bandwidth of a typical CD-ROM drive.

MPEG in the Real World

MPEG has found a number of applications in the real world, including:

1. Direct Broadcast Satellite. MPEG video streams are received by a dish/decoder, which unpacks the data and synthesizes a standard NTSC television signal.

2. Cable Television. Trial systems are sending MPEG-II programming over cable television lines.

3. Media Vaults. Silicon Graphics, Storage Tech, and other vendors are producing on-demand video systems, with twenty file thousand MPEG-encoded films on a single installation.

4. Real-Time Encoding. This is still the exclusive province of professionals. Incorporating special-purpose parallel hardware, real-time encoders can cost twenty to fifty thousand dollars.

3 Other Lossy Transform Codes

3.1 Wavelet Compression

JPEG and MPEG decompose images into sets of cosine waveforms. Unfortunately, cosine is a periodic function; this can create problems when an image contains strong aperiodic features. Such local high-frequency spikes would require an infinite number of cosine waves to encode properly. JPEG and MPEG solve this problem by breaking up images into fixed-size blocks and transforming each block in isolation. This effectively clips the infinitely-repeating cosine function, making it possible to encode local features.

An alternative approach would be to choose a set of basis functions that exhibit good locality without artificial clipping. Such basis functions, called "wavelets", could be applied to the entire image, without requiring blocking and without degenerating when presented with high-frequency local features.

How do we derive a suitable set of basis functions? We start with a single function, called a "mother function". Whereas cosine repeats indefinitely, we want the wavelet mother function, ϕ, to be contained within some local region, and approach zero as we stray further away:

$$\lim_{x \to \pm\infty} \phi(x) = 0$$

The family of basis functions are scaled and translated versions of this mother function. For some scaling factor s and translation factor l,

$$\phi_{sl}(x) = \phi(2^s x - l)$$

A well know family of wavelets are the Haar wavelets, which are derived from the following mother function:

$$\phi(x) = \begin{cases} 1 & : \quad 0 < x \leq 1/2 \\ -1 & : \quad 1/2 < x \leq 1 \\ 0 & : \quad x \leq 0 \text{ or } x > 1 \end{cases}$$

Figure 8 shows a family of seven Haar basis functions. Of the many potential wavelets, Haar wavelets are probably the most described but the least used. Their regular form makes the underlying mathematics simple and easy to illustrate, but tends to create bad blocking artifacts if actually used for compression.

Many other wavelet mother functions have also been proposed. The Morret wavelet convolves a Gaussian with a cosine, resulting in a periodic but smoothly decaying function. This function is equivalent to a wave packet from quantum physics, and the mathematics of Morret functions have been studied extensively. Figure 9 shows a sampling of other popular wavelets. Figure 10 shows that the Daubechies wavelet is actually a self-similar fractal.

Wavelets in the Real World

Summus Ltd. is the premier vendor of wavelet compression technology. Summus claims to achieve better quality than JPEG for the same compression ratios, but has been loathe to divulge details of how their wavelet compression actually works. Summus wavelet technology has been incorporated into such items as:

- Wavelets-on-a-chip for missile guidance and communications systems.
- Image viewing plugins for Netscape Navigator and Microsoft Internet Explorer.
- Desktop image and movie compression in Corel Draw and Corel Video.
- Digital cameras under development by Fuji.

In a sense, wavelet compression works by characterizing a signal in terms of some underlying generator. Thus, wavelet transformation is also of interest outside of the realm of compression. Wavelet transformation can be used to clean up noisy data or to detect self-similarity over widely varying time scales. It has found uses in medical imaging, computer vision, and analysis of cosmic X-ray sources.

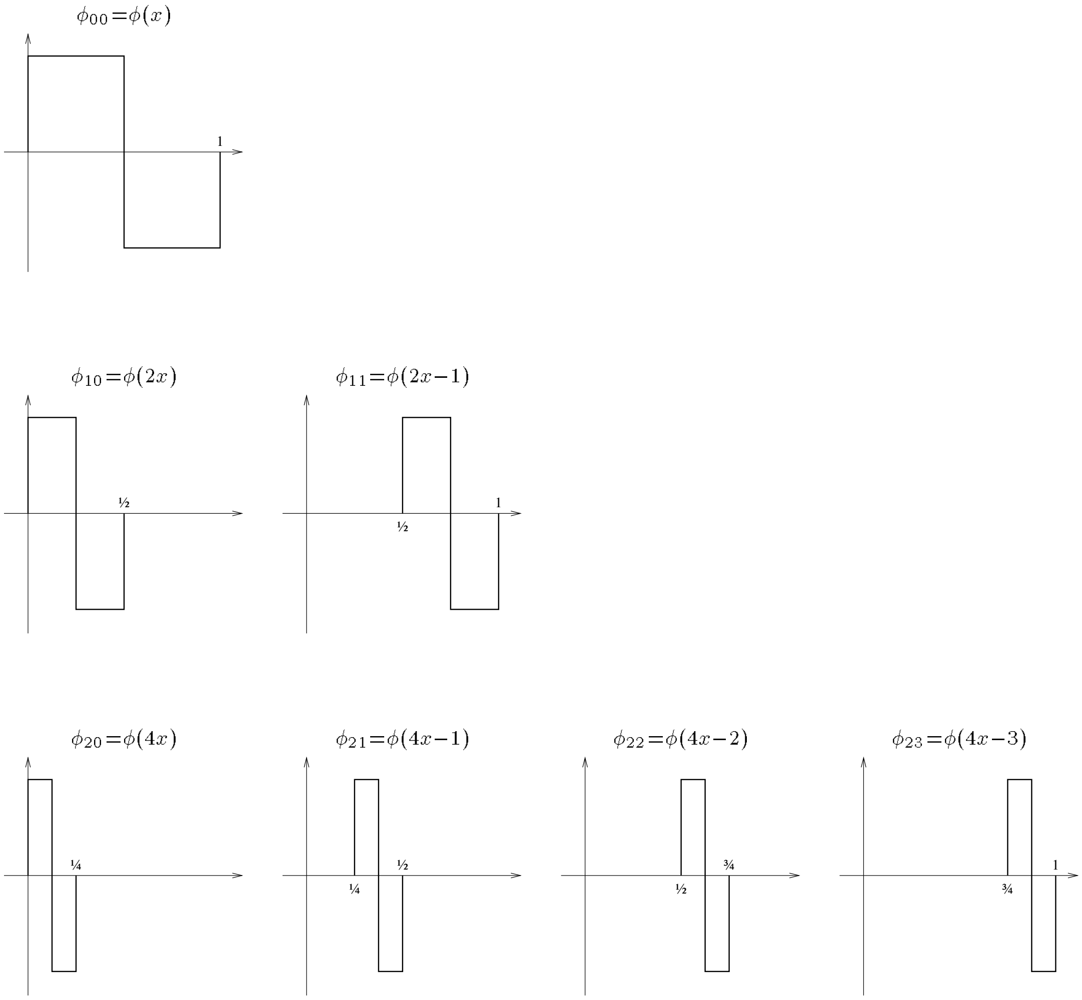

Figure 8: A small Haar wavelet family of size seven.

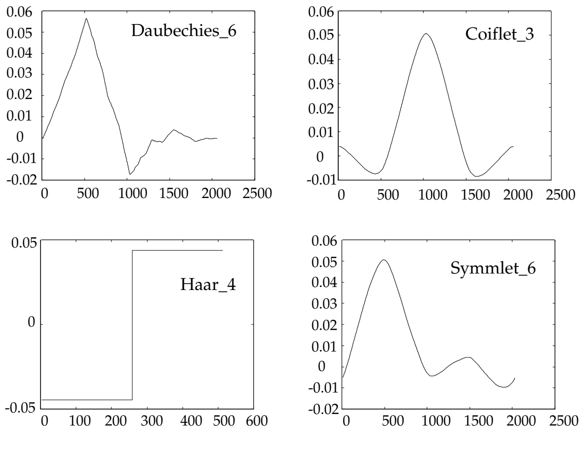

Figure 9: A sampling of popular wavelets.

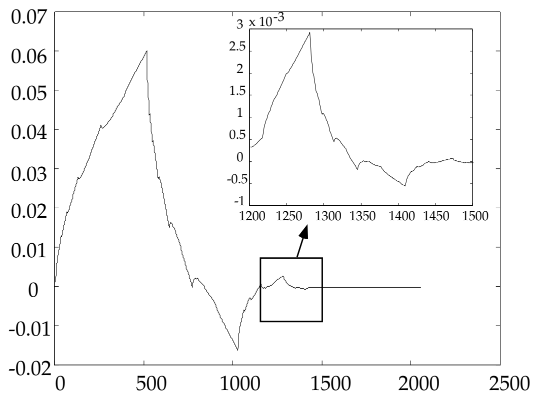

Figure 10: Self-similarity in the Daubechies wavelet.

3.2 Fractal Compression

A function $f(x)$ is said to have a fixed point x_f if $x_f = f(x_f)$. For example:

$$\begin{aligned} f(x) &= ax + b \\ \Rightarrow x_f &= \frac{b}{1-a} \end{aligned}$$

This was a simple case. Many functions may be too complex to solve directly. Or a function may be a black box, whose formal definition is not known. In that case, we might try an iterative approach. Keep feeding numbers back through the function in hopes that we will converge on a solution:

$$\begin{aligned} x_0 &= \text{guess} \\ x_i &= f(x_{i-1}) \end{aligned}$$

For example, suppose that we have $f(x)$ as a black box. We might guess zero as x_0 and iterate from there:

$$\begin{aligned} x_0 &= 0 \\ x_1 &= f(x_0) = 1 \\ x_2 &= f(x_1) = 1.5 \\ x_3 &= f(x_2) = 1.75 \\ x_4 &= f(x_3) = 1.875 \\ x_5 &= f(x_4) = 1.9375 \\ x_6 &= f(x_5) = 1.96875 \\ x_7 &= f(x_6) = 1.984375 \\ x_8 &= f(x_7) = 1.9921875 \end{aligned}$$

In this example, $f(x)$ was actually defined as $1/2x + 1$. The exact fixed point is 2, and the iterative solution was converging upon this value.

Iteration is by no means guaranteed to find a fixed point. Not all functions have a single fixed point. Functions may have no fixed point, many fixed points, or an infinite number of fixed points. Even if a function has a fixed point, iteration may not necessarily converge upon it.

In the above example, we were able to associate a fixed point value with a function. If we were given only the function, we would be able to recompute the fixed point value. Put differently, if we wish to transmit a value, we could instead transmit a function that iteratively converges on that value.

This is the idea behind fractal compression. However, we are not interested in transmitting simple numbers, like "2". Rather, we wish to transmit entire images. Our fixed points will be images. Our functions, then, will be mappings from images to images.

Our encoder will operate roughly as follows:

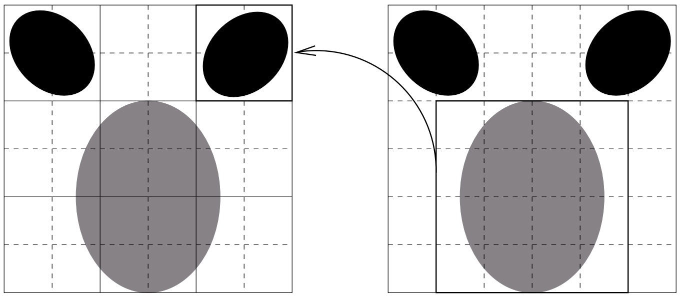

Figure 11: Identifying self-similarity. Range blocks appear on the left; one domain block appears on the left. The arrow identifies one of several collage function that would be composited into a complete image.

1. Given an image, i, from the set of all possible images, $Image$.
2. Compute a function $f : Image \rightarrow Image$ such that $f(i) \approx i$.
3. Transmit the coefficients that uniquely identify f.

Our decoder will use the coefficients to reassemble f and reconstruct its fixed point, the image:

1. Receive coefficients that uniquely identify some function $f : Image \rightarrow Image$.
2. Iterate f repeatedly until its value converges on a fixed image, i.
3. Present the decompressed image, i.

Clearly we will not be using entirely arbitrary functions here. We want to choose functions from some family that the encoder and decoder have agreed upon in advance. The members of this family should be identifiable simply by specifying the values for a small number of coefficients. The functions should have fixed points that may be found via iteration, and must not take unduly long to converge.

The function family we choose is a set of "collage functions", which map regions of an image to similar regions elsewhere in the image, modified by scaling, rotation, translation, and other simple transforms. This is vaguely similar to the search for similar macroblocks in MPEG P- and B-frame encoding, but with a much more flexible definition of similarity. Also, whereas MPEG searches for temporal self-similarity across multiple images, fractal compression searches for spatial self-similarity within a single image.

Figure 11 shows a simplified example of decomposing an image info collages of itself. Note that the encoder starts with the subdivided image on the right. For each "range" block, the encoder searchers for a similar "domain" block elsewhere in the image. We generally want domain blocks to be larger than range blocks to ensure good convergence at decoding time.

Fractal Compression in the Real World

Fractal compression using iterated function systems was first described by Dr. Michael Barnsley and Dr. Alan Sloan. Although they claimed extraordinary compression rates, the computational cost of encoding was prohibitive. The major vendor of fractal compression technology is Iterated Systems, cofounded by Barnsley and Sloan.

Today, fractal compression appears to achieve compression ratios competitive with JPEG at reasonable encoding speeds. Details on Iterated's current techniques are, sadly, few and far between.

Fractal compression describes an image in terms of itself, rather than in terms of a pixel grid. This means that fractal images can be somewhat resolution-independent. Indeed, one can easily render a fractal image into a finer or coarser grid than that of the source image. This resolution independence may have use in presenting quality images across a variety of screen and print media.

3.3 Model-Based Compression

We briefly present one last transform coding scheme, model-based compression. The idea here is to characterize the source data in terms of some strong underlying model. The popular example here is faces. We might devise a general model of human faces, describing them in terms of anatomical parameters like nose shape, eye separation, skin color, cheekbone angle, and so on. Instead of transmitting the image of a face, we could transmit the parameters that define that face within our general model. Assuming that we have a suitable model for the data at hand, we may be able to describe the entire system using only a few bytes of parameter data.

Both sender and receiver share a large body of *a priori* knowledge contained in the model itself (e.g., the fact that faces have two eyes and one nose). The more information is shared in the model, the less need be transmitted with any given data set. Like wavelet compression, model-based compression works by characterizing data in terms of a deeper underlying generator. Model-based encoding has found applicability in such areas as computerized recognition of four-legged animals or facial expressions.