
©1998 Ben Liblit and Guy Blelloch

- Case Studies
  1. JPEG Encoding
  2. MPEG Encoding
- Other Lossy Transform Codes
  1. Wavelet Compression
  2. Fractal Compression
  3. Model-Based Compression

1 Case Studies

In previous lectures we examined several algorithms for lossless and lossy compression. Now we draw together these isolated parts into complete compression schemes suitable for use on real-world data.

We will examine, as case studies, two popular lossy compression schemes: JPEG for still images, and MPEG for video.

1.1 JPEG

1.1.1 JPEG Introduction

JPEG is a lossy compression scheme for color and grayscale images. It has several important properties:

- designed for natural, real-world scenes

  JPEG sustains excellent compression rations on photographic material and naturalistic artwork. It performs poorly on cartoons, line drawings, and other images with large areas of solid color delimited by sharp boundaries.

- tunable lossy compression

  JPEG discards data at several phases of the compression. In general, data is discarded where it will have the smallest subjective impact on a human viewer. JPEG contains tunable quality "knobs" that allow one to trade off between compression effectiveness and image fidelity.

- 24-bit color

  JPEG maintains 24 bits of color data, in contrast to GIF's limit of 8 bits. In practice, this can mean that GIF discards more data than JPEG, since GIF must first quantize an image down to no more than 256 distinct colors.

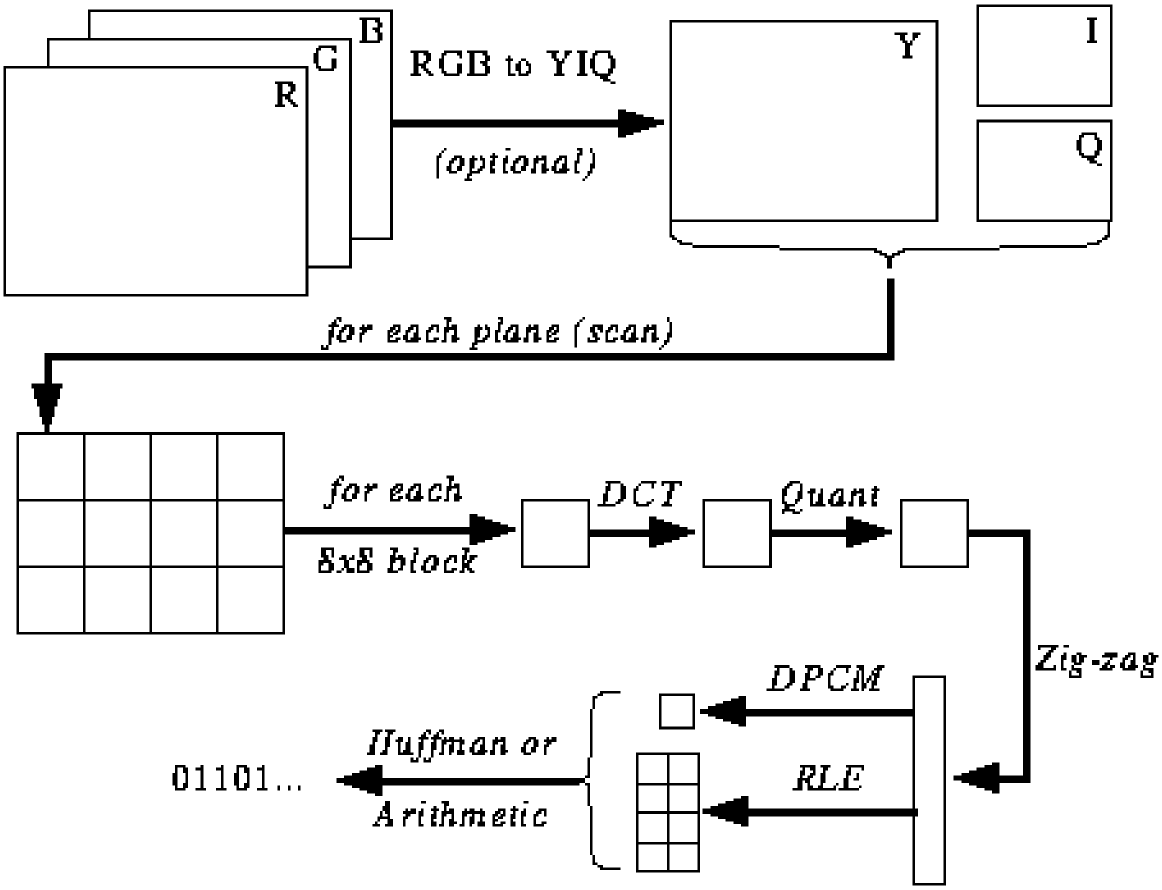

Figure 8: Steps in JPEG compression.

JPEG forms an excellent case study in compression because it synthesizes a number of distinct algorithms:

- Transform Coding
- Scalar Quantization
- Difference Coding
- Run Length Coding
- Huffman or Arithmetic Coding

We will highlight each of these techniques as they appear in the algorithm, below.

1.1.2 JPEG Compression Steps

1. (optional) Convert image from RGB to YIQ.

   RGB is a three-dimensional vector space that describes colors in a manner convenient for computer hardware. YIQ is a different color space that more closely matches subjective human color perception. The "I" represents the relative balance of orange versus cyan. The "Q" component represents the relative balance of green versus magenta. The "I" and "Q" components collectively represent the hue, or "chrominance" of any pixel. The "Y" component represents the overall brightness, or luminosity, of a color. This brightness is weighted to reflect the sensitivities of human vision: green contributes most to perceived luminance, red contributes about half as strongly, and blue has the least impact.

   When converting the the image to YIQ, we must choose how many bits will be used to represent each component. We generally devote four times as many bits to luminance (Y) as we do to chrominance (IQ). The eye is much more sensitive to small changes in brightness than it is to small changes in hue.

   Incidentally, NTSC television transmissions encode colors using the YIQ color space, with the largest portion of bandwidth devoted to luminance. Also, since brightness is represented using a single luminosity component, black and white televisions can display this value directly and simply ignore the two chrominance components.

2. Subdivide the image into blocks of 8×8 pixels.

   Small, fixed-size blocks will greatly simplify subsequent computation. Small blocks also help to limit cosine transformation "leakage" should the image contain sharp (high-frequency) discontinuities. Without these blocks, a single sharp feature could bleed into the entire rest of the image.

3. For each 8×8 block, apply the discrete cosine transformation to produce a transformed 8×8 block with the same bit depth.

   This is a deployment of transform coding, independently applied to the horizontal and vertical axes of each block.

4. Reduce the overall depth of these transformed blocks by a scaling factor.

   This is a deployment of scalar quantization. The quantization error that this necessarily introduces is the most significant source of information loss in JPEG compression.

   One could simply divide all entries by some uniform value, but this will not actually give the best results. When images are intended for use by human viewers, some bits are more important than others. Instead, variable scaling factors are derived from an 8×8 quantization table. Each element in the transformed block is divided by the corresponding element in the quantization table. This allows us to be selective in where we discard information.

   The JPEG standard defines two such tables. One is used for chrominance (I, Q), and is not presented here. The other is used for luminance (Y), and is shown in Table 6. Notice that the largest coefficients are in the lower right corner, while the smallest

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 6: JPEG default quantization table, luminance plane.

are toward the upper left. This will discard the most bits from high frequencies, and the fewest bits from low frequencies. Psychometric studies show that the human eye is more sensitive to lower frequencies, so this is where we can least afford to discard information.

Abstractly, the default tables represent the relative importance of features in any image. If we were to uniformly double all of the quantization coefficients, we would still be using the same relative importance, but we would be discarding more bits. Thus, the entire quantization table may be scaled by any single factor to improve compression while reducing image quality.

In most JPEG compressors, this single factor is the "quality knob" made available to the user. A scaling factor of 8 is a rough upper limit for high-fidelity reproduction; factors up to 16 will be lossy but presentable; factors as high as 32 may still be recognizable but are unusable for most practical tasks.

After quantization, the DCT coefficients will lie in a much narrower range. This reduces their overall entropy and will make subsequent compression more effective. In fact, much of the high-frequency region toward the lower-right may have been reduced to zeros.

5. Encode the DC components.

   This is a deployment of difference coding followed by Huffman or arithmetic coding.

   The DC component contains most of the information in the image. Across an entire image, the DC component is large and varied. Within any local region, though, the DC component is often close to the previous value. Thus, instead of encoding the (high-entropy) DC values directly, we encode the (low-entropy) differences between DC values. Specifically, for each DC component in each 8×8 block, we encode the difference from the corresponding DC component in the previous 8×8 block.

   These differences will tend to be small, and thus amenable to entropy coding. Either Huffman or arithmetic coding is used to further compress the DC difference stream.

6. Linearize the AC components.

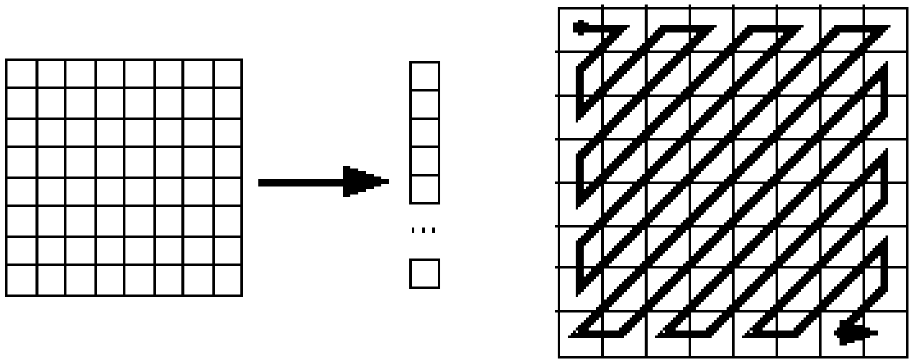

Figure 9: Zig-zag scanning of JPEG blocks.

In order to sequentially encode the AC components of an 8×8 block, we must first represent it as a simple 1×64 vector. The obvious approach would be to iterate across the 64 cells in row-major or column-major order. A much better approach is to zig-zag across the block, as shown in Figure 9.

Why is this desirable? Recall that the upper-left region represents high-frequency features, while the lower-right represents low-frequency features. A zig-zag path groups low- and high-frequency coefficients close together in the 1×64 vector.

This means that successive coefficients will tend to have similar values. Furthermore, many coefficients will be zero, particularly toward the low-frequency lower-right corner. These correlations will make subsequent compression much more effective.

7. Encode the AC components.

   This is a deployment of run length encoding followed by Huffman or arithmetic coding.

   AC components are encoded as (skip, value) pairs, where skip is the number of zeros and value is the next non-zero component. The special pair (0, 0) designates the end of a block.

   The code pairs are then further compressed using Huffman or arithmetic coding.

1.2 MPEG

1.2.1 MPEG Introduction

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

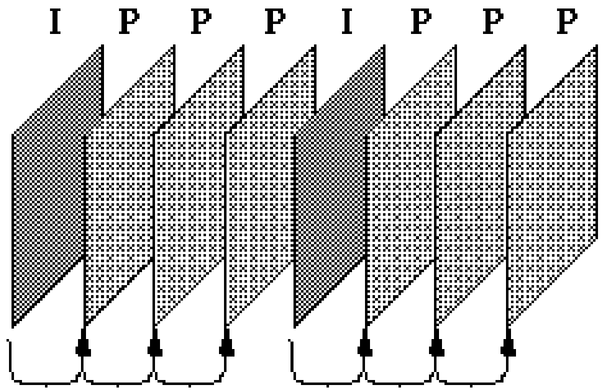

Figure 10: MPEG I-frames interspersed with P-frames.

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 7: MPEG B-frames postponed in data stream.

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. I-frames are encoded as isolated images. They do not take advantage of redundant information in any other frames.

P-frame , or predictive coded frame. A P-frame can take advantage of redundant information in the previous I- or P-frame.

B-frame , or bidirectionally predictive coded frame. A B-frame can take advantage of similar information in both the preceding and the succeeding I- or P-frame.

Figure 10 shows a simplified MPEG stream containing just I- and P-frames. Figure 11 shows a more complex stream that adds B-frames.

I-frames and P-frames appear in an MPEG stream in simple, chronological order. However, B-frames are moved forward 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 stream. Table 7 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

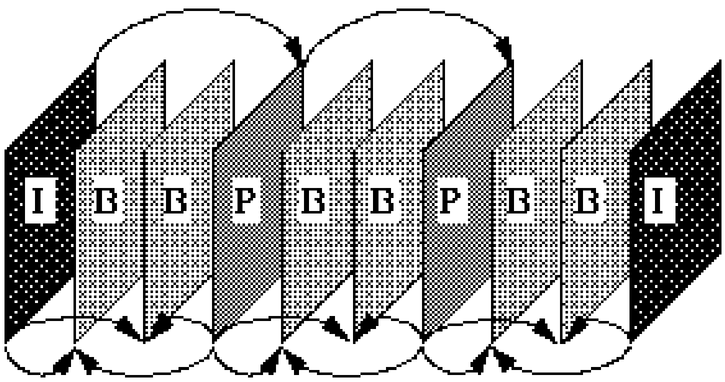

Figure 11: MPEG I- and P-frames interspersed with B-frames.

I-frames. In practice, most encoders use some fixed pattern like the ones shown here: two or three P-frames between each pair of I-frames, and two or three B-frames between those.

1.2.2 MPEG Encoding: I-Frames

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. First, each frame is converted into the YCrCb color space, a variant of JPEG's YIQ. The frame is then subdivided into 16×16 pixel blocks, called "macroblocks". The luminance data (Y) is represented at full resolution: each 16×16 macroblock is simply split into four 8×8 subblocks. Chrominance data (Cr and Cb) is represented at one quarter resolution: each 16×16 macroblock is reduced to a single 8×8 subblock for each hue axis.

We now have six 8×8 blocks (four Y, one Cr, one Cb). Encoding of these 8×8 blocks proceeds using same JPEG scheme presented above. Each block is cosine-transformed, quantized, zig-zag sequenced, run-length encoded, and Huffman encoded to a final bit stream.

One important difference from MPEG concerns the quantization phase. In its original form, MPEG-I did not support JPEG's variable quantization table. Rather, all coefficients were scaled by a single constant value. MPEG-II restored variable, nonuniform quantization.

1.2.3 MPEG Encoding: P-Frames

P-frames are subdivided into macroblocks using the same technique described above. However, we will not encode the macroblocks directly. Rather, we will attempt to encode the *difference* between each macroblock and a similar macroblock from an earlier frame.

For each macroblock in the current ("target") frame, we search for a matching group of pixels in the preceding ("reference") I-frame or P-frame. In many cases, we may find an

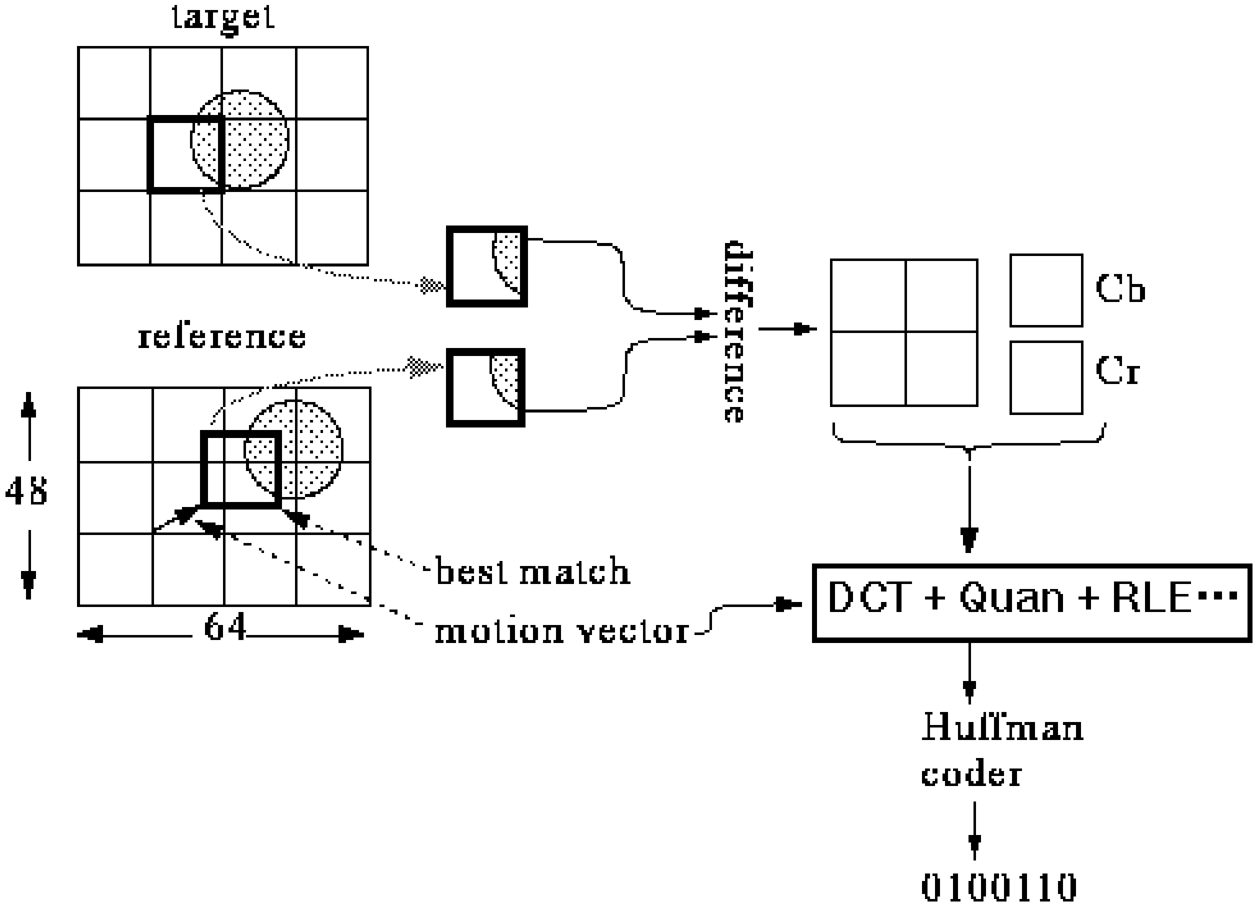

Figure 12: P-frame encoding.

excellent match at the same position in the reference frame. If that region of the frame represents static background scenery, it will likely be unchanged from target to reference.

In general, though, we wish to accommodate motion. Thus, we search for a good target match throughout some larger region of the reference image. Ultimately, we will find our best match at some (x, y) offset from the position of our target macroblock. This offset, or "motion vector", is Huffman encoded for each target macroblock.

The motion vector may direct us to the best match, but it is unlikely to direct us to a set of pixels that are truly identical. Here we use difference coding. We form a macroblock representing the difference between the target macroblock and best-match reference macroblock. This difference macroblock is encoded using the same JPEG technique used for I-frame macroblocks: DCT, quantize, RLE, etc.

In practice, the search for a good match for each target macroblock is the most computationally difficult part of MPEG encoding. With current technology, realtime MPEG encoding is only possible with the help of custom hardware. Note, though, that while the *search* for a match is extremely difficult, *using* the motion vector so derived is as simple as extracting a fixed-size block of pixels given an (x, y) offset into a buffered frame. Encoders are hard; decoders are easy.

One final point: it is entirely possible that for some target macroblock, we will find *no* similar region in the reference frame. Should this happen, we simply give up on difference coding for that block. We omit the motion vector and JPEG-encode the target macroblock directly. In a sense, we have fallen back onto I-frame coding for just that macroblock.

1.2.4 MPEG Encoding: B-Frames

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 identical to that used in P-frames. For each macroblock in the target image, search for a similar collection of pixels in the reference. However, whereas P-frames searched in only the preceding I- or P-frame, B-frames also search in the succeeding I- or P-frame. We find the best match in each frame, and record both. The best match in the previous frame is encoded as a backward-predicted motion vector; the best match in the next frame is encoded as a forward-predicted motion vector.

Recall that a P-frame also encodes the difference between the target and the reference. B-frames have two motion vectors, and therefore two references. We simply merge the two reference blocks and encode the difference between this average and our target.

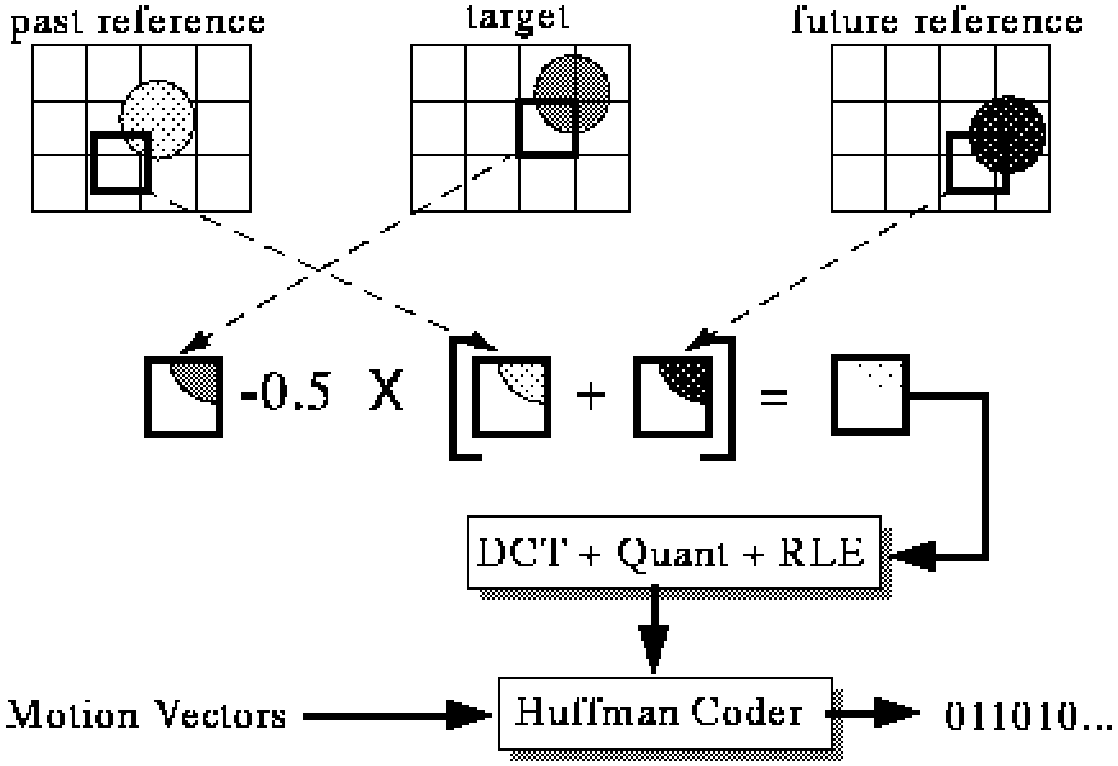

Figure 13: B-frame encoding.

Like P-frames, B-frames may fail to find a good match in either the forward or backward reference. Should this happen, we omit that motion vector and rely exclusively upon the other. In a sense, we will have fallen back on P-frame or backward P-frame coding for just that macroblock. If no good match can be found in either direction, we directly encode the raw target macroblock as though for an I-frame.

1.2.5 MPEG Compression Effectiveness

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?

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

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.

1.2.6 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.

2 Other Lossy Transform Codes

2.1 Wavelet Compression

2.1.1 Wavelet Introduction

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,

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

2.1.2 Haar Wavelets

Haar wavelets 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 14 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

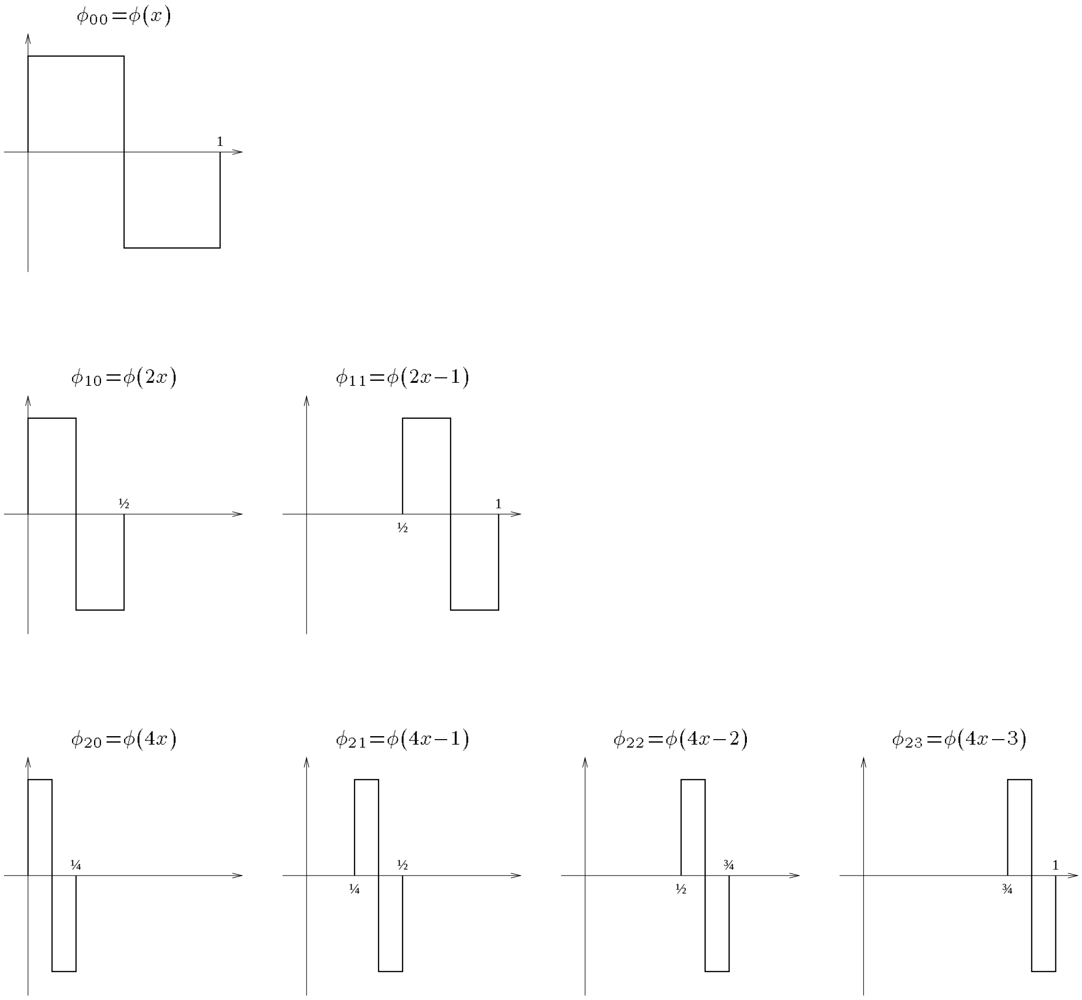

Figure 14: A small Haar wavelet family of size seven.

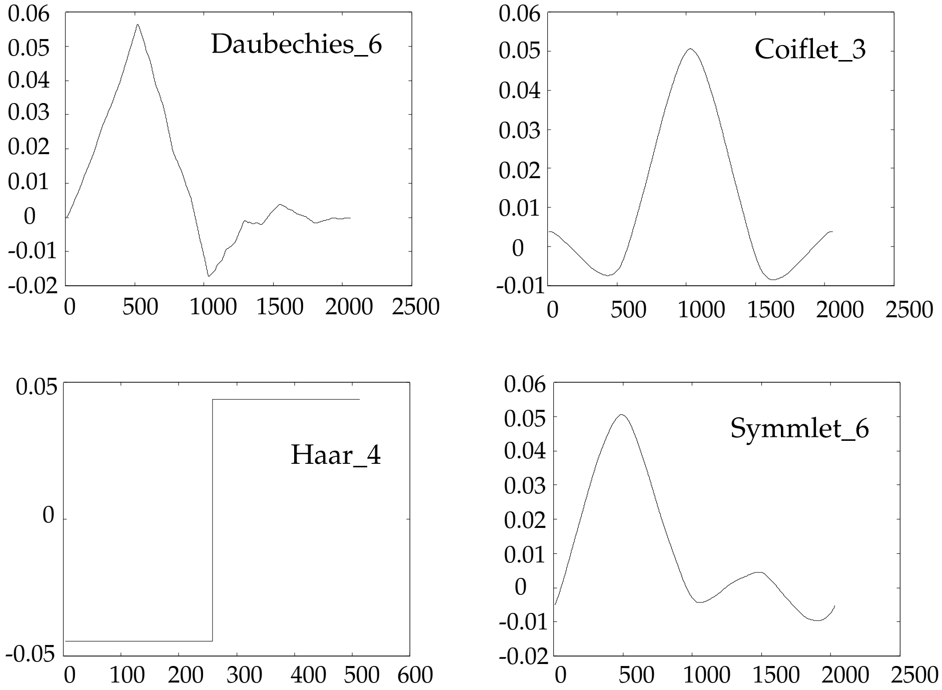

Figure 15: A sampling of popular wavelets.

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 15 shows a sampling of other popular wavelets. Figure 16 shows that the Daubechies wavelet is actually a self-similar fractal.

2.1.3 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.

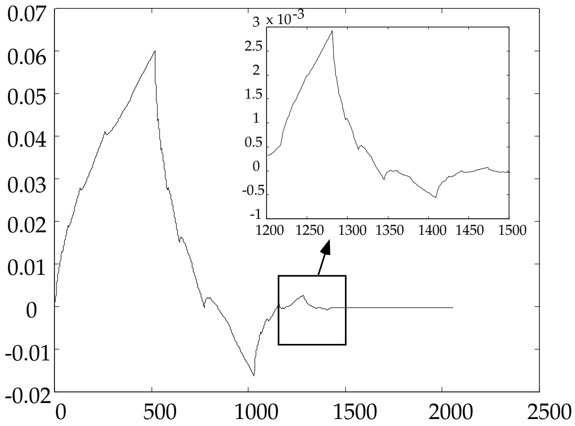

Figure 16: Self-similarity in the Daubechies wavelet.

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.

2.2 Fractal Compression

2.2.1 Function Fixed Points

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.

2.2.2 Fractal Compression and Fixed Points

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:

1. Given an image, i, from the set of all possible images, $Image$.
2. Compute a function $f : Image \to Image$ 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 $f : Image \to Image$.
2. Iterate f repeatedly until its value converges on a fixed image, i.
3. Present the decompressed image, i.

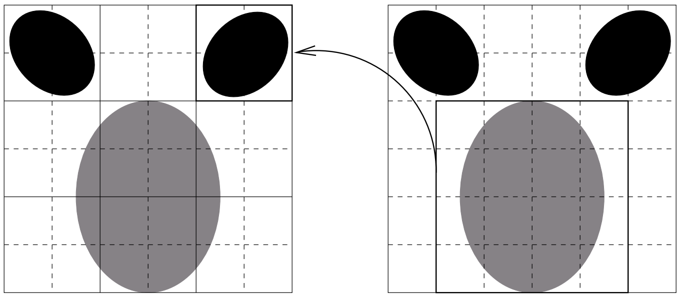

Figure 17: Identifying self-similarity. Range blocks appear on the left; one domain block appears on the left. The arrow identifies one of several collage function that would be composited into a complete image.

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 17 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.

2.2.3 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.

2.3 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.