UNIT IV COMPRESSION TECHNIQUES **9**

Principles – Text compression – Static Huffman coding – Dynamic Huffman coding – Arithmetic coding – Image compression – Graphics interchange format – Tagged image file format – Digitized documents – Introduction to JPEG standards.

1 Introduction

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand. The neat thing about compression, as with the other topics we will cover in this course, is that the algorithms used in the real world make heavy use of a wide set of algorithmic tools, including sorting, hash tables, tries, and FFTs. Furthermore, algorithms with strong theoretical foundations play a critical role in real-world applications.

In this chapter we will use the generic term *message* for the objects we want to compress, which could be either files or messages. The task of compression consists of two components, an *encoding* algorithm that takes a message and generates a "compressed" representation (hopefully with fewer bits), and a *decoding* algorithm that reconstructs the original message or some approximation of it from the compressed representation. These two components are typically intricately tied together since they both have to understand the shared compressed representation.

We distinguish between *lossless algorithms*, which can reconstruct the original message exactly from the compressed message, and *lossy algorithms*, which can only reconstruct an approximation of the original message. Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Lossy is used in an abstract sense, however, and does not mean random lost pixels, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise. For example, one might think that lossy text compression would be unacceptable because they are imagining missing or switched characters. Consider instead a system that reworded sentences into a more standard form, or replaced words with synonyms so that the file can be better compressed. Technically the compression would be lossy since the text has changed, but the "meaning" and clarity of the message might be fully maintained, or even improved. In fact Strunk and White might argue that good writing is the art of lossy text compression.

Is there a lossless algorithm that can compress all messages? There has been at least one patent application that claimed to be able to compress all files (messages)—Patent 5,533,051 titled "Methods for Data Compression". The patent application claimed that if it was applied recursively, a file could be reduced to almost nothing. With a little thought you should convince yourself that this is not possible, at least if the source messages can contain any bit-sequence. We can see this by a simple counting argument. Lets consider all 1000 bit messages, as an example. There are 2^{1000} different messages we can send, each which needs to be distinctly identified by the decoder. It should be clear we can't represent that many different messages by sending 999 or fewer bits for all the messages — 999 bits would only allow us to send 2^{999} distinct messages. The truth is that if any one message is shortened by an algorithm, then some other message needs to be lengthened. You can verify this in practice by running GZIP on a GIF file. It is, in fact, possible to go further and show that for a set of input messages of fixed length, if one message is compressed, then the average length of the compressed messages over all possible inputs is always going to be longer than the original input messages. Consider, for example, the 8 possible 3 bit messages. If one is compressed to two bits, it is not hard to convince yourself that two messages will have to expand to 4 bits, giving an average of 3 1/8 bits. Unfortunately, the patent was granted.

Because one can't hope to compress everything, all compression algorithms must assume that there is some bias on the input messages so that some inputs are more likely than others, *i.e.* that there is some unbalanced probability distribution over the possible messages. Most compression algorithms base this "bias" on the structure of the messages – *i.e.*, an assumption that repeated characters are more likely than random characters, or that large white patches occur in "typical" images. Compression is therefore all about probability.

When discussing compression algorithms it is important to make a distinction between two components: the model and the coder. The *model* component somehow captures the probability distribution of the messages by knowing or discovering something about the structure of the input. The *coder* component then takes advantage of the probability biases generated in the model to generate codes. It does this by effectively lengthening low probability messages and shortening high-probability messages. A model, for example, might have a generic "understanding" of human faces knowing that some "faces" are more likely than others (*e.g.*, a teapot would not be a very likely face). The coder would then be able to send shorter messages for objects that look like faces. This could work well for compressing teleconference calls. The models in most current real-world compression algorithms, however, are not so sophisticated, and use more mundane measures such as repeated patterns in text. Although there are many different ways to design the model component of compression algorithms and a huge range of levels of sophistication, the coder components tend to be quite generic—in current algorithms are almost exclusively based on either Huffman or arithmetic codes. Lest we try to make to fine of a distinction here, it should be pointed out that the line between model and coder components of algorithms is not always well defined.

It turns out that information theory is the glue that ties the model and coder components together. In particular it gives a very nice theory about how probabilities are related to information content and code length. As we will see, this theory matches practice almost perfectly, and we can achieve code lengths almost identical to what the theory predicts.

Another question about compression algorithms is how does one judge the quality of one versus another. In the case of lossless compression there are several criteria I can think of, the time to compress, the time to reconstruct, the size of the compressed messages, and the generality—*i.e.*, does it only work on Shakespeare or does it do Byron too. In the case of lossy compression the judgement is further complicated since we also have to worry about how good the lossy approximation is. There are typically tradeoffs between the amount of compression, the runtime, and the quality of the reconstruction. Depending on your application one might be more important than another and one would want to pick your algorithm appropriately. Perhaps the best attempt to systematically compare lossless compression algorithms is the Archive Comparison Test (ACT) by Jeff Gilchrist. It reports times and compression ratios for 100s of compression algorithms over many databases. It also gives a score based on a weighted average of runtime and the compression ratio.

This chapter will be organized by first covering some basics of information theory. Section 3 then discusses the coding component of compressing algorithms and shows how coding is related to the information theory. Section 4 discusses various models for generating the probabilities needed by the coding component. Section 5 describes the Lempel-Ziv algorithms, and Section 6 covers other lossless algorithms (currently just Burrows-Wheeler).

Run-length Coding

Probably the simplest coding scheme that takes advantage of the context is run-length coding. Although there are many variants, the basic idea is to identify strings of adjacent messages of equal value and replace them with a single occurrence along with a count. For example, the message sequence `acccbbaaabb` could be transformed to (a,1), (c,3), (b,2), (a,3), (b,2). Once transformed, a probability coder (*e.g.*, Huffman coder) can be used to code both the message values and the counts. It is typically important to probability code the run-lengths since short lengths (*e.g.*, 1 and 2) are likely to be much more common than long lengths (*e.g.*, 1356).

An example of a real-world use of run-length coding is for the ITU-T T4 (Group 3) standard for Facsimile (fax) machines[2]. At the time of writing (1999), this was the standard for all home and business fax machines used over regular phone lines. Fax machines transmit black-and-white images. Each pixel is called a *pel* and the horizontal resolution is fixed at 8.05 pels/mm. The vertical resolution varies depending on the mode. The T4 standard uses run-length encoding to code each sequence of black and white pixels. Since there are only two message values black and white, only the run-lengths need to be transmitted. The T4 standard specifies the start color by placing a dummy white pixel at the front of each row so that the first run is always assumed to be a white run. For example, the sequence `bbbbwwbbbbb` would be transmitted as 1,4,2,5. The

run-length	white codeword	black codeword
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
..		
20	0001000	00001101000
..		
64+	11011	0000001111
128+	10010	000011001000

Table 3: ITU-T T4 Group 3 Run-length Huffman codes.

T4 standard uses static Huffman codes to encode the run-lengths, and uses a separate codes for the black and white pixels. To account for runs of more than 64, it has separate codes to specify multiples of 64. For example, a length of 150, would consist of the code for 128 followed by the code for 22. A small subset of the codes are given in Table 4.1. These Huffman codes are based on the probability of each run-length measured over a large number of documents. The full T4 standard also allows for coding based on the previous line.

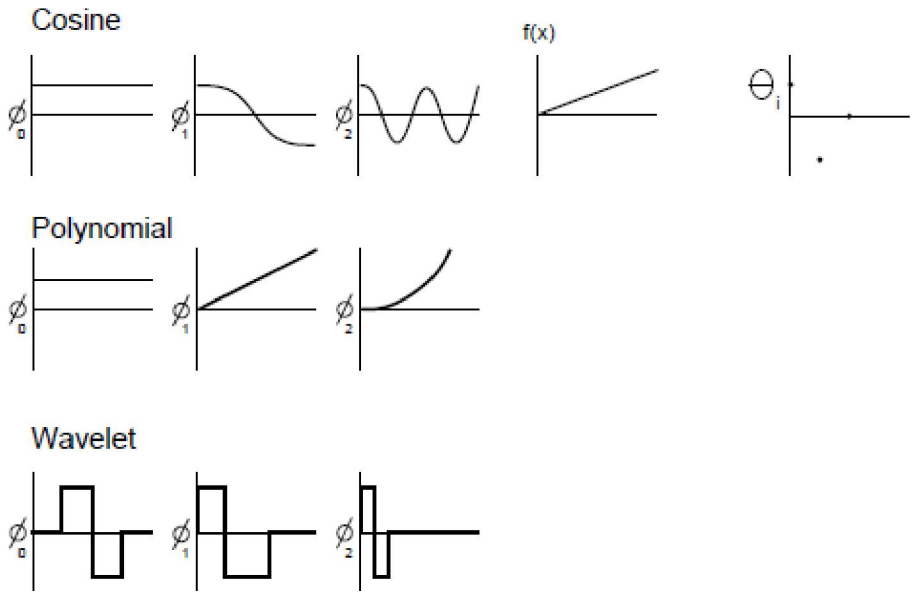
Transform Coding

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 18 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 are more important than others.



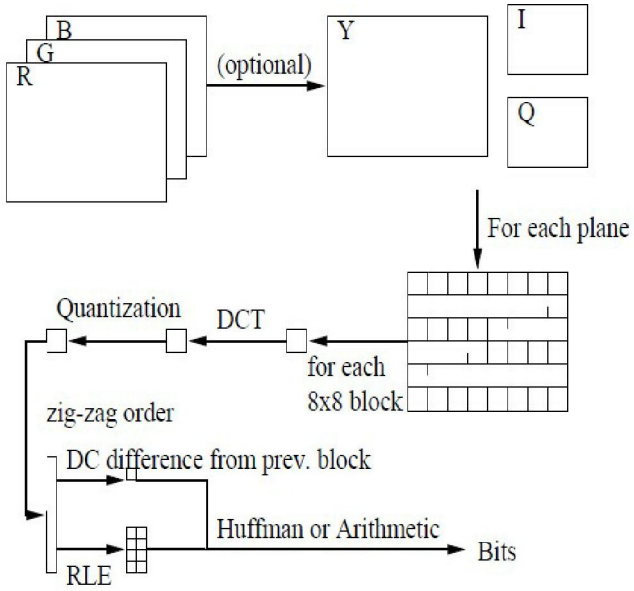


Figure 19: Steps in JPEG compression.

designed to better represent human perception and are what are used 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

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.

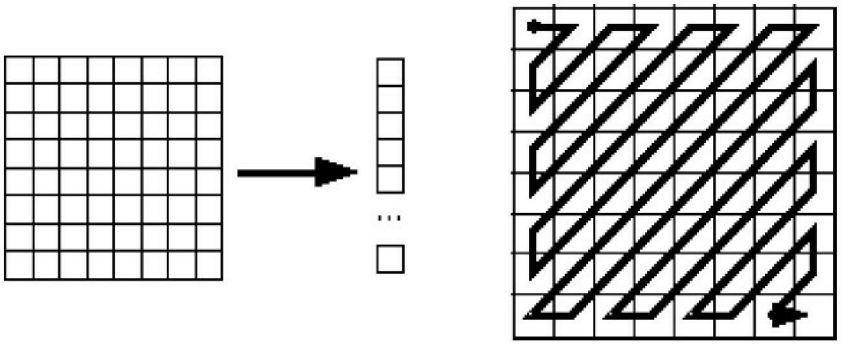


Figure 20: Zig-zag scanning of JPEG blocks.

defines standard quantization tables for both the Y and I-Q components. The table for Y is shown in Table 6. 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 20). The motivation for this order is that it keeps frequencies of approximately equal length close to each other

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

Figure 21: MPEG B-frames postponed in data stream.

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.

MPEG

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 19. 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

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 exploits 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 21 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 stream. Figure 21 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.

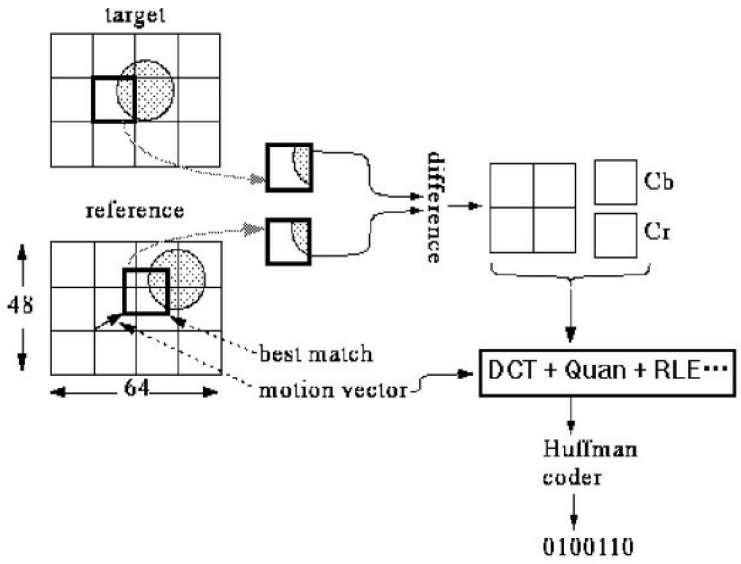


Figure 22: P-frame encoding.

Since I-frames are independent images, they can be encoded as if they were still images. The particular technique used by MPEG is a variant of the JPEG technique (the color transformation and quantization steps are slightly different). I-frames are very important for use as anchor points so that the frames in the video can be accessed randomly without requiring one to decode all previous frames. To decode any frame we need only find its closest previous I-frame and go from there. This is important for allowing reverse playback, skip-ahead, or error-recovery.

The intuition behind encoding P-frames is to find matches, *i.e.*, groups of pixels with similar patterns, in the previous reference frame and then coding the difference between the P-frame and its match. To find these "matches" the MPEG algorithm partitions the P-frame into 16x16 blocks. The process by which each of these blocks is encoded is illustrated in Figure 22. For each *target* block in the P-frame the encoder finds a *reference* block in the previous P- or I-frame that most closely matches it. The reference block need not be aligned on a 16-pixel boundary and can potentially be anywhere in the image. In practice, however, the x-y offset is typically small. The offset is called the *motion vector*. Once the match is found, the pixels of the reference block are subtracted from the corresponding pixels in the target block. This gives a residual which ideally is close to zero everywhere. This residual is coded using a scheme similar to JPEG encoding, but will ideally get a much better compression ratio because of the low intensities. In addition to sending the coded residual, the coder also needs to send the motion vector. This vector is Huffman coded. The motivation for searching other locations in the reference image for a match is to allow for the efficient encoding of motion. In particular 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 best match will not be in the same location in the image. It should be noted that if no good match is found, then the block is coded as if it were from an I-frame.

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.

In practice, the search for good matches for each target block is the most computationally expensive 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.