PVRG-MPEG CODEC 1.1

Andy C. Hung

March 1, 1993

This package is based on the UNIX operating system. We appreciate any comments or corrections - they can be sent to Andy C. Hung at achung@cs.stanford.edu. We cannot guarantee that any bugs will be corrected however.

Funded by the Defense Advanced Research Projects Agency.

I am especially grateful to Hewlett Packard and Storm Technology for their financial support during the earlier stages of codec development. Any errors in the code and documentation are my own. The following people are acknowledged for their advice and assistance. Thanks, one and all.

- The Portable Video Research Group at Stanford: Teresa Meng, Peter Black, Ben Gordon, Sheila Hemami, Wee-Chiew Tan, Eli Tsern.
- Adriaan Ligtenberg of Storm Technology. Jeanne Wiseman, Andrew Fitzhugh, Gregory Yovanof of Hewlett Packard. Eric Hamilton and Jean-Georges Fritsch of C-Cube Microsystems.
- Lawrence Rowe of the Berkeley Plateau Research Group.
- Karl Lillevold.

Dedicated to the standardization committees and their efforts in bringing the advancement of technology to everyone.

Contents

Chapter 1

Getting Started

This document describes the Portable Research Video Group's (PVRG) PVRG-MPEG software codec. The initial version of this codec was developed in 1990, based on the very early MPEG simulation model specifications. The PVRG-MPEG software codec that is described here is based on the Santa Clara 1991 draft. Since 1991, very little development on the PVRG-MPEG software codec has occurred, until recent verification became possible. We should caution the user that bugs are probably still lurking.

1.1 Common Questions and Answers

Let's answer some common questions about the MPEG-I proposed standard. In the following discussion we will use MPEG to refer to MPEG-I. Another emerging standard for higher bit-rates is MPEG-II (or MPEG phase II).

- **What is MPEG** MPEG stands for Moving Picture Experts Group which is responsible for audio, video, and combined synchronization of information in digital image sequence coding. MPEG is a joint international committee (ISO-IEC/JTC1/SC2/WG8).
- **How does MPEG-I differ from H.261?** The MPEG-I algorithm is defined for higher bit rates–0.9 Mbps to 1.5 Mbps and consequently for higher quality the H.261. The H.261 specification targets a somewhat lower telecommunication rate, as low as 64 kbps, and a short interactive delay. The combined encoder-decoder delay in MPEG may be higher than is acceptable for true telecommunication; this delay may not be objectionable in multimedia.
- **How does MPEG differ from JPEG?** MPEG is targeted toward image sequence compression. JPEG is targeted toward still (single) image compression.
- **What does this particular package do?** This package implements the MPEG Committee Draft (Santa Clara '91) compression specifications with some encoding algorithms based on MPEG simulation models 1-3, though not identical. It should be able to encode and decode streams for most MPEG players.
- **Where can I obtain the code?** The PVRG-MPEG codec source code can be obtained by anonymous ftp from havefun.stanford.edu:pub/mpeg/MPEGv1.1.tar.Z. Some MPEG coded sequences are also available there.
- **What distinguishes this package from other available systems?** This package is implemented in software for simulation purposes. It has not been optimized for speed - parts have been kludged from the PVRG-JPEG codec. We have a small internal program interpreter for experimenting with different rate control techniques. This implementation is also free; commercial MPEG codec implementations will likely have a much more sophisticated rate-control and a faster decoder. A public domain MPEG software decoder produced by the Berkeley Plateau Research Group is available at toe.cs.berkeley.edu in /pub/multimedia/mpeg/mpeg-2.0.tar.Z. The Berkeley Plateau Research Group's decoder is faster than the PVRG-MPEG decoder and goes directly to the X display.

- **What kind of system do I need?** Pure storage memory requirement (excluding the size of code) for the MPEG codec is approximately 2.5 times the luminance image size multiplied by the frame interval.

1.2 Further Information

Some references on MPEG are listed in the bibliography.

Chapter 2

Understanding Image Compression

2.1 Introduction

The complete picture of the image starts from the color representation. Thus, we begin with analysis of color spaces and then start the examination of motion compensation and estimation, transforms, quantization, and entropy coding.

2.2 Color Space

Since Newton's time, it has been known that a wide spectrum of colors can be generated from a set of three primaries. An artist, for example, can paint most colors from layering pigments of the subtractive primaries: red, yellow, and blue. Television displays generate colors by mixing lights of the additive primaries: red, green, and blue. Although two primary systems can be used– most generally a red-orange and a green-blue– the image rendered is not lifelike, and two color systems never became successful in either motion picture or pre-standard television[1].

The color space obtained through combining the three colors can be determined by drawing a triangle on a special color chart with each of the base colors as an endpoint. The classic color chart used in early television specifications was established in 1931 by the Commission Internationale de L'Eclairage (CIE).

One of the special concepts introduced by the 1931 CIE chart was the isolation of the luminance, or brightness, from the chrominance, or hue. Using the CIE chart as a guideline, the National Television System Committee (NTSC) defined the transmission of signals in a luminance and chrominance format, rather than a format involving the three color components of the television phosphors. The new color space was labeled YIQ, representing the luminance, in-phase chrominance, and quadrature chrominance coordinates respectively.

In Europe, two television standards were later established, the phase-alternation-line (PAL) format and the Séquentiel couleurà mémoire (SECAM) format, both with an identical color space, YUV. The only change between the PAL/SECAM YUV color space and the NTSC YIQ color space is a 33 degree rotation in UV space[2]. The MPEG algorithm is formulated for a three component color system such as the YUV format.

The conversion between the standard Red Green Blue (RGB) format to YUV format is slightly different for digital signals than for analog signals. For the JPEG JFIF[13] format convention, the full range of 8 bits is used for Y, U, and V. The digital conversion insures that if the RGB inputs are between 0 and 255, (normalized so that equal values represents reference white), then the Y output has values between 0-255 and the U and V have values between 0 and ± 128. The values are stored as 8 bit unsigned characters, thus the U and V values are level shifted by adding 128 so that the values are always non-negative, and if U and V equal 256, they are clamped to 255. The conversion, written in a matrix form, is

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.1}$$

The inverse conversion (after subtracting 128, if the values of U and V are level-shifted) is

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \tag{2.2}$$

While it is easily confirmed that the RGB to YUV conversion yields values that are automatically between 0-255 for the Y and between ± 128 for U and V, conversely not all YUV space maps to legitimate RGB space. Thus the values for RGB should be clamped after colorspace conversion to 0 to 255.

The other common YUV conversion (indeed the specified conversion for H.261 and MPEG) performs slightly different scaling, by CCIR Recommendation 601 specifications. The CCIR YUV nominal range is between 16 to 235 for the luminance and 16 to 240 for the chrominances. Thus this color space is just a rescale and shift on the formulas described above (i.e. Y' = 219/255 * Y + 16, U' = 224/255 * U + 128, V' = 224/255 * V + 128 (assuming both U and V do not have level shifts prior to scaling, but U' and V' do)). Although the CCIR 601 YUV colorspace is specified for H.261, some streams may be generated with JFIF color components.

The YUV format, similar to the YIQ format[3], concentrates most of the image information into the luminance and less in the chrominance. The result is that the YUV elements are less correlated and, therefore, can be coded separately without much loss in efficiency.

Another advantage comes from reducing the transmission rates of the U and V chrominance components. Commonly known from the testing of the NTSC YIQ format and the PAL/SECAM YUV format is that the chrominance need not be specified as frequently as the luminance. Hence, for the MPEG algorithm, only every other U element and every other V element in both the horizontal and vertical directions are sampled. The missing elements can be reconstructed by various means of interpolation, including duplication.

The reduction in data by converting from RGB to YUV is 2 to 1 (denoted 2:1). For example, if the RGB format is specified by eight bits for each color, then each RGB picture element is described by 24 bits; and after conversion and decimation, each YUV pixel is described by an average of 12 bits: the luminance at eight bits, and both the chrominances for every other pixel (horizontally and vertically) at eight bits.

2.3 MPEG Model

The MPEG model, shown in figure 2.1, consists of five stages: a motion compensation stage, a transformation stage, a lossy quantization stage, and two lossless coding stages. The motion compensation stage subtracts the current image from a shifted view of the previous image if they are both alike. The transform concentrates the information energy into the first few transform coefficients, the quantizer causes a controlled loss of information, and the two coding stages further compress the data closer to symbol entropy.

MPEG compression is considered "lossy" because the reconstructed image is not identical to the original. Lossless coders, which create images identical to the original, achieve very poor compression because the least significant bits of each color component become progressively more random, thus harder to code.

2.4 Motion Compensation and Estimation

Since most frames in an image sequence look very similar excepting shifts due to movement such as a pan of the video camera across a scene, we can avoid coding the same block twice by sending over the displacement vector from the previous image caused by the "pan." Continuing this analogy, it is even better to consider the future as well as the past; therefore, MPEG motion compensation/estimation considers both time directions. A typical MPEG sequence consists of three separate parts: a series of intraframes, which are image frames coded individually without any temporal prediction; a series of forward predicted frames, interspersed between these intraframes; and bidirectionally predicted frames interspersed between the forward predicted frames and the intraframes. The bidirectionally predicted frames can be considered motion-compensated interpolation between the predicted and the intra frames. This is shown in figure 2.2. The intraframe is coded first, then the next predicted frame, then the interpolated frames between the two. The process repeats with the next predicted frame and interpolated frames, the next predicted frame and interpolated frames, etc.

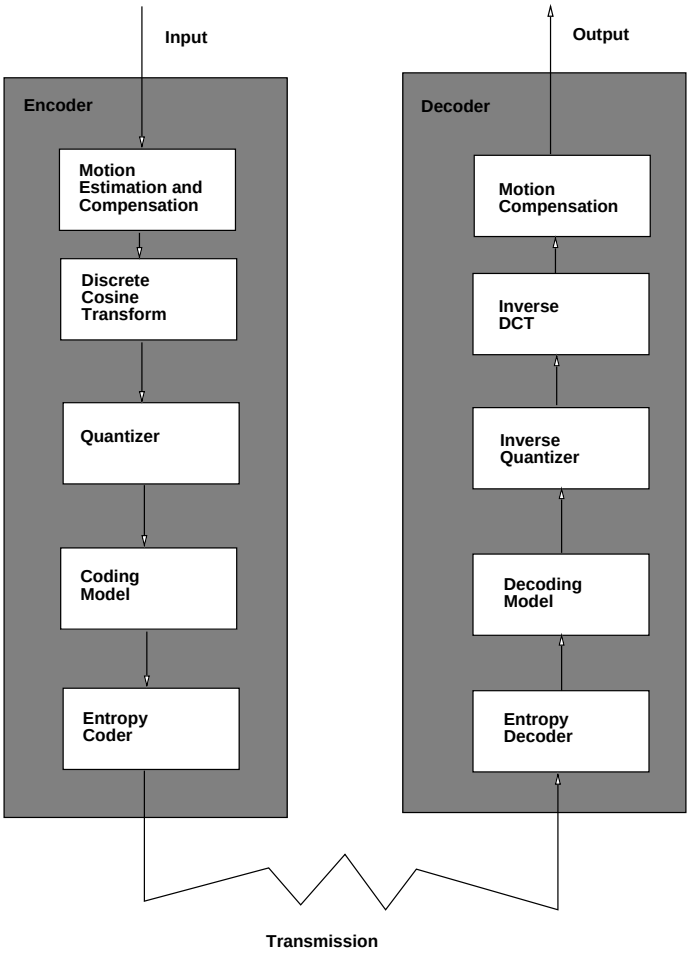

Figure 2.1: The flow of information from the encoder to the decoder for the MPEG system.

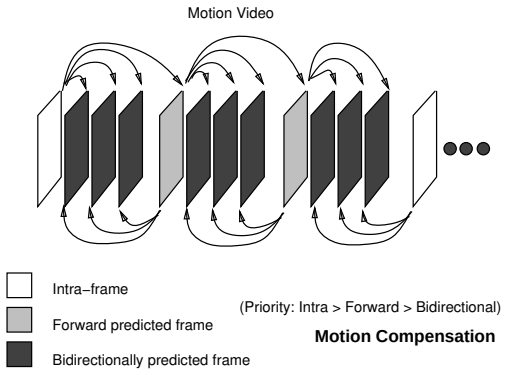

Figure 2.2: The motion compensation model for MPEG.

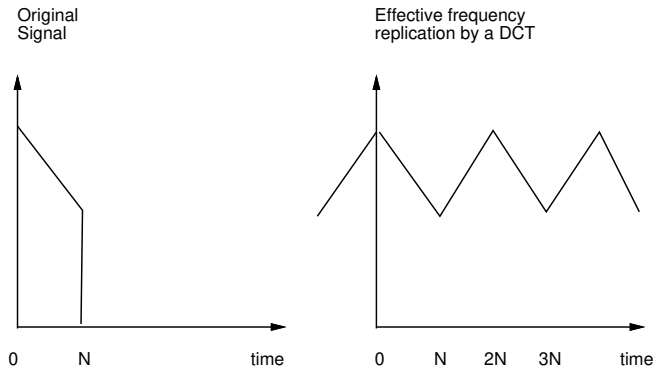

Figure 2.3: The DCT transform. The correspondence between the original windowed signal between 0 to N and the replication performed implicitly by the DCT to fill out times outside of the window.

2.5 Transform Stage

If the video energy of the image is of low spatial frequency– slowly varying, then a transform can be used to concentrate the energy into very few coefficients. To simplify the transform complexity, the image is divided into small blocks. The transform method chosen by CCITT is the two dimensional 8 by 8 DCT, a transform studied extensively for image compression.

Conceptually, a one dimensional DCT can be thought of as taking the Fourier Transform of an infinite sequence as shown in figure 2.3. This sequence consists of the data vector reflected across the axis and repeated indefinitely along the axes.

The two dimensional DCT can be obtained by performing a one dimensional DCT on the columns and a one dimensional DCT on the rows. An explicit formula[1] for the two dimensional 8 by 8 DCT can be written in terms of the pixel values, $f(i,j)$, and the frequency domain transform coefficients, $F(u,v)$.

$$F(u,v) = (1/4)C(u)C(v)\sum_{i=0}^{7}\sum_{j=0}^{7} f(i,j)\cos((2i+1)u\pi/16)\cos((2j+1)v\pi/16), \tag{2.3}$$

where

$$C(x) = \begin{cases} 1/\sqrt{2} & x = 0 \\ 1 & \text{otherwise} \end{cases} . \tag{2.4}$$

The transformed output from the 2D DCT will be ordered so that the mean value, the DC coefficient, is in the upper left corner and the higher frequency coefficients progress by distance from the DC coefficient. The higher vertical frequencies are represented by higher row numbers and the higher horizontal frequencies are represented by higher column numbers.

The inverse of the 2D DCT is written as

$$f(i,j) = (1/4)\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos((2i+1)u\pi/16)\cos((2j+1)v\pi/16). \tag{2.5}$$

As an example of the DCT, lets take the transform of a pizza. Assume that the pizza has only cheese topping so that the top of the pizza is a uniform color. Also assume that the pizza is infinitely thin (2 dimensional). The 8 by 8 input represented by the symbol O for the outside of the pizza and X for the inside of the pizza is

[1]The complexity of the DCT can be simplified tremendously. For example, in the one dimensional 8 element DCT, the multiplications can be reduced to 12, see [5]; and for the two dimensional 8 by 8 case, with scaled DCT's some multiplications can be combined with the quantization process, and the number of multiplications purely for the inverse DCT is reduced to 61, see [6].

```
OOOXXOOO
OXXXXXXO
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
OXXXXXXO
OOOXXOOO
```

If we assign $O = -10$ and $X = 10$, the pizza looks like

-10	-10	-10	10	10	-10	-10	-10
-10	10	10	10	10	10	10	-10
10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10
-10	10	10	10	10	10	10	-10
-10	-10	-10	10	10	-10	-10	-10

and by (2.3), the frequency domain representation, rounded to the nearest integer, is

40	0	-26	0	0	0	-11	0
0	0	0	0	0	0	0	0
-45	0	-24	0	8	0	-10	0
0	0	0	0	0	0	0	0
-20	0	0	0	20	0	0	0
0	0	0	0	0	0	0	0
-3	0	10	0	18	0	4	0
0	0	0	0	0	0	0	0

We can conclude that the DCT of a pizza doesn't resemble anything edible. But this example does illustrate the reduction in data caused by the transform: more zero coefficients and a concentration of energy around the upper left corner of the matrix.

Even though the energy distribution has changed, the total energy remains the same because the DCT is a unitary transformation (but because a balanced DCT involves the square root of the data length, the DCT is sometimes defined with unbalanced normalization).

Furthermore, since the DCT is unitary, the maximum value of each 8 by 8 DCT coefficient is limited to a factor of eight times the original values; and after rounding, an eight bit input value can be represented by an eleven bit transformed value. Coincidently, the IDCT of this block, when rounded to the nearest integer, yields the original pizza, but perfectly exact reconstruction is not always possible with an integer value DCT.

2.6 Quantization

The coefficients of the DCT are quantized to reduce their magnitude and to increase the number of zero value coefficients. The uniform quantizer is used for the MPEG method, with a different stepsize per DCT coefficient position.

The intraframe blocks are quantized with the DC and the AC terms separately; the AC and DC quantization are

$$C(0,0) = \lfloor (F(0,0) \pm 4)/8 \rfloor, \tag{2.6}$$

$$A(u,v) = \lfloor ((F(u,v) * 16) \pm Q(u,v)/2)/Q(u,v) \rfloor, \tag{2.7}$$

$$C(u,v) = \lfloor ((A(u,v) \pm Q_F)/2Q_F \rfloor. \tag{2.8}$$

where $C(u,v)$ is the quantized coefficient, $F(u,v)$ is the DCT frequency coefficient, $Q(u,v)$ is the quantizer stepsize, Q_F is the quantizing parameter (for rate control); and $\pm$ is positive for $F(u,v)$ positive, negative otherwise.

The inverse intra quantizer is

$$F(0,0) = 8C(0,0), \tag{2.9}$$

$$F(u,v) = C(u,v)Q_F Q(u,v)/8. \tag{2.10}$$

For forward predicted and interpolated blocks, the quantizer has a dead-band around zero, and is the same for both AC and DC components as

$$A(u,v) = \lfloor((F(u,v) * 16) \pm Q(u,v)/2)/Q(u,v)\rfloor, \tag{2.11}$$

Then if Q_F is odd

$$C(u,v) = A(u,v)/2Q_F, \tag{2.12}$$

and if Q_F is even

$$C(u,v) = (A(u,v) \pm 1)/2Q_F, \tag{2.13}$$

where $\pm$ is positive for $A(u,v) > 0$ positive, negative for $A(u,v) < 0$. The inverse quantizer is then

$$F(u,v) = (2F(u,v) \pm 1)Q_F Q(u,v)/16, \tag{2.14}$$

where $\pm$ is positive for $F(u,v) > 0$ positive, negative for $F(u,v) < 0$.

If any of the inverse quantization results in $F(u,v)$ even, then the reconstructed transform coefficient is rounded down to the next closest odd integer to zero. This prevents artifacts arising from even reconstructed coefficient values in the inverse DCT.

Quantization is the lossy stage in the MPEG coding scheme (note that the integer-DCT introduces very slight loss). If we quantize too coarse, we may end up with images that look "blocky," but if we quantize too fine, we may spend useless bits coding (essentially) noise.

2.7 Coding Model

The coding model rearranges the quantized DCT coefficients into a zig-zag pattern, with the lowest frequencies first and the highest frequencies last. The zig-zag pattern is used to increase the run-length of zero coefficients found in the block. The assumption is that the lowest frequencies tend to have larger coefficients and the highest frequencies are, by the nature of most pictures, predominantly zero.

As illustrated in figure 2.4, the first coefficient (0,0) is called the DC coefficient and the rest of the coefficients are called AC coefficients. The AC coefficients are traversed by the zig-zag pattern from the (0,1) location to the (7,7) location.

The quantized DC coefficients are encoded by the number of significant bits, followed by the bits themselves. The quantized AC coefficients usually contain runs of consecutive zeroes. Therefore, a coding advantage can be obtained by using a run-length technique. The AC coefficients are encoded based on the number of zeroes before the next non-zero coefficient, and the next non-zero coefficient. For frequently occurring combinations of zero-run-length/nonzero-coefficient, a unique variable length code is used. For the other codes, an ESCAPE variable length code allows the definition of the run-length of zeroes, and the level of the coefficients, as is.

The inverse run-length coder translates the input coded stream into an output array of AC coefficients. Based on the run-length code, it takes the current position in the output array and appends a number of zeroes followed by the next non-zero coefficient.

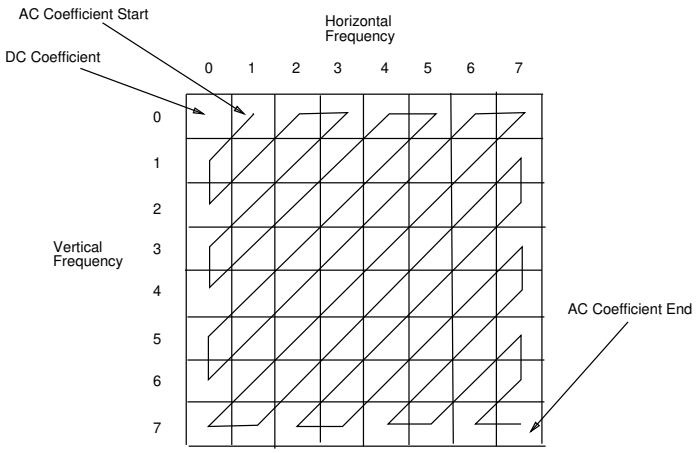

Figure 2.4: The zigzag pattern for reordering the two dimensional DCT coefficients into a one dimensional array.

Phrase	Symbol	Frequency
Because	b	1
I'm	I	6
Bad	B	15
Come on	C	2
It	i	1
Really	R	6
You know	Y	4

Table 2.1: The frequency of words in the song *Bad*.

2.8 Entropy Coding

The block codes from the DPCM and run-length models can be further compressed using entropy coding. For the MPEG method, the Huffman coder[7] is used to compress the data closer to symbol entropy. One reason for using the Huffman coder is that it is easy to implement in hardware. To compress data symbols, the Huffman coder creates shorter codes for frequently occurring symbols and longer codes for occasionally occurring symbols.

For example, let's encode an excerpt from Michael Jackson's song *Bad*[2].

```
Because I'm bad, I'm bad-- come on
Bad, bad-- really, really bad
You know I'm bad, I'm bad--
you know it
Bad, bad-- really, really bad
You know I'm bad, I'm bad-- come on, you know
Bad, bad-- really, really bad
```

The first step in creating Huffman codes is to create a table assigning a frequency count to each phrase. In the above lyrics, ignoring capitalization, this is shown in table 2.1.

Initially designate all symbols as leaf nodes for a tree. Now starting from the two least weight nodes, aggregate the pair into a new node. For example, in the above frequency chart, *Because* and *It* would be

[2] 1987 ©MJJ Productions, Inc.

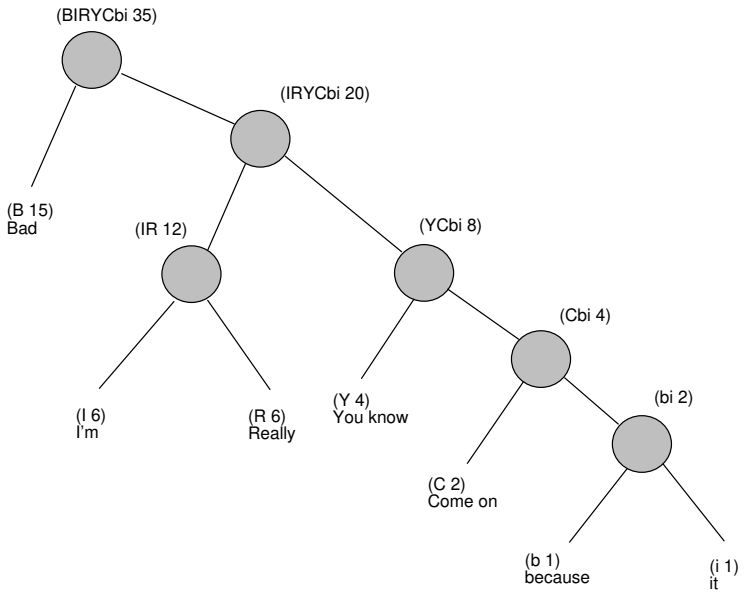

Figure 2.5: The Huffman tree for the lyrics to *Bad*.

Phrase	Symbol	Frequency	Code Length	Code
Because	b	1	5	00001
I'm	I	6	3	011
Bad	B	15	1	1
Come on	C	2	4	0001
It	i	1	5	00000
Really	R	6	3	010
You know	Y	4	3	001

Table 2.2: The Huffman codes for the words in *Bad*.

aggregated first. Repeat this process for the new set until the entire symbol set is represented by a single node. The result is shown in figure 2.5.

A Huffman code can be generated for each symbol by attaching a binary digit to each branch. Let's assign the binary digit 1 to the left branches and the binary digit 0 to the right branches. The symbol code is generated by following the path of branches from the top node to the symbol leaf node. In the case of the word *Bad*, the Huffman code is 1, and for the word *I'm*, the Huffman code is 011. A full table for each symbol is shown in table 2.2.

To encode a symbol, just output the code word to the bit stream. For example, output the code word 010 for *Really*. To decode from a stream of bits, start from the top node, and follow the left branch or right branch depending on the value taken from the bit stream; continue until a leaf node is reached. The decoded symbol is the symbol associated with that leaf.

In the above example, the first few bits in the output bit stream are (bars are inserted for the reader's convenience)

00001—011—1—011—1—...

Our coding efficiency can be calculated by comparing the number of bits required to realize the lyrics. For the Huffman code above, the bit length is 15(1) + 16(3) + 2(4) + 2(5) = 81 bits. In comparison, for a three bit code, the bit length is 35(3) = 105 bits; and for an ideal 7 symbol code, the bit length is $35(\log_2 7) = 98.3$ bits. The Huffman coder compresses the lyrics by about 20 percent; but this figure does not include the cost of

transmitting the initial Huffman code table to the decoder.

In truth, Huffman coding is not exactly used in MPEG. A true Huffman code is based on a full tree (at least for binary alphabets). The MPEG code tables are often not full trees, and thus are more properly described as "variable-length codes" rather than Huffman codes, though we do not make this distinction in this report.

2.9 Layering

The MPEG model is based on several layers, each of these layers designed to give structure to the coding model. We review the layers before describing the actual compression algorithms to give a tour of the hierarchy of information.

The topmost layer is the video sequence layer. It consists of the important signaling information - the dimensions of the frame; the pel-aspect ratio; the picture-rate; the quantization matrices used, if not the default.

The next layer is the group-of-pictures layer. It consists of timing and reconstruction information for a group of pictures; typically it is inserted at regular intervals to aide in editing, but there is no requirement that it should.

Right below is the picture layer, which describes the location of the current video frame (with respect to the group-of-pictures layer) as well as the picture type, current buffer fullness, temporal reference, and other information.

The slice layer is used to divide up the picture into pieces that can survive significant errors. These slices are placed end-to-end in a raster scan across the image; they consist of macroblocks, which are described next.

The macroblock layer is the smallest integral image unit in the coding process. It is composed from the input image as follows. The input image is composed of a luminance and two chrominance components in a ratio of 4:1:1 for Y, U, and V. These component frames are divided up into 8x8 blocks. Since there are four times as many luminance blocks than each chrominance component block, a logically consistent "Macroblock" codes four luminance blocks followed by a block for each of the two chrominances. All component blocks within a macroblock share similar properties of motion displacement; they are coded together as well. The order of transmission starts from the four blocks of the luminance to the two chrominances. For the YUV system the transmission pattern would look like

```
Y1 Y2           Y5 Y6
Y3 Y4 U1 V1     Y7 Y8 U2 V2 ...
```

The block layer represents individual 8x8 blocks of the image of which the macroblocks consist.

2.10 Details on encoding

The PVRG-MPEG software codec algorithms are based on the early MPEG simulation models, though there are some differences. We shall describe the encoding algorithms that we use.

Our software MPEG encoder can be modeled as in figure 2.6. The encoder calculates the motion compensation vectors for a set of *original* frames in the frame store. A selection method (described below) determines which one of the motion vectors to use. Once the motion vector is selected, the current frame is compensated, and the difference is coded through a transform and a quantizer. Both the motion vectors used and the coded output bits are entropy coded before being interleaved onto the final output bit-stream.

The decoding algorithm is simple. It does the inverse quantization, inverse transform, and compensates with the motion vectors by the previously transmitted frames in the frame store. One should note that, while the motion vectors are generated using the original frames, motion compensation is done on the decoder by the reconstructed frames. Clearly the encoder must perform the decoding as well to find the proper differences, thus the encoder is more sophisticated than the decoder.

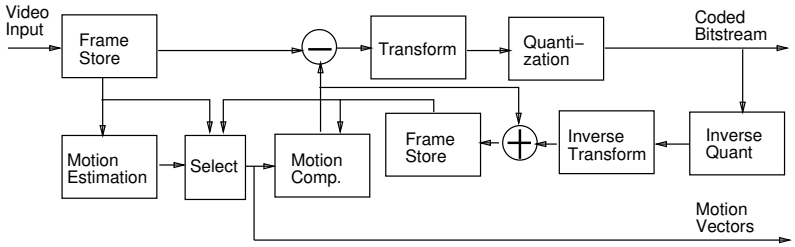

Figure 2.6: The lossy portion of the encoding algorithm for the PVRG-MPEG software codec.

2.11 Motion Estimation

In our MPEG software codec implementation, the motion estimation compares a 16 by 16 superblock (consisting of four eight by eight blocks) in the luminance throughout a small search area of a previously transmitted intra-frame or predicted frame. The range for such comparison is between ± 8 pixels based on the luminance component of the image at half-pel accuracy. The displacement with the smallest absolute superblock difference, determined by the sum of the absolute values of the pixel-to-pixel difference throughout the block, is considered the motion-compensation vector for that particular macroblock. (The chrominance motion vector is just the luminance motion vector divided in half).

For frame distances greater than one, the motion estimation is telescopic; that is, the search area for the current frame starts at the best motion vector obtained for replenishment of the previous frame. The internal code uses a radially increasing circular search starting from a motion vector of (0,0) with short circuit evaluation, so the precise description of range is not easy.

The motion compensated block is the difference between the best-match block and the block to be coded. Whether the motion compensated block is used, or the original block is used for coding is based on the following decisions.

For interpolated frames, the direction of prediction is chosen based on the smallest squared error for either forward compensation, or backward compensation, or bi-directional compensation (where the bi-directionally predicted block is the mean of both forward and backward prediction).

The predicted frames has a choice between zero displacement motion compensation (sometimes called interframe compensation) and motion vector compensation. If the energy of the motion compensated block with a zero displacement (bd) is roughly less than the energy of the motion compensated block with best-match displacement (dbd), then zero displacement motion compensation is used, otherwise motion vector compensation is used.

For both interpolated frames and predicted frames, the decision whether to use compensation at all is made by the following criterion. If motion compensated image block's variance (VAR) is less than the original block's variance (VAROR), then motion compensation/interframe compensation is used, otherwise no compensation is used and the block is coded as is. The threshold functions for these decisions are shown in figure 2.7.

2.12 Rate Control

For our MPEG software codec, rate control is provided by a simple buffer-proportional feedback to the quantizer step-size. The intra-frame and forward predicted-frame quantizers are generally kept at half the quantization size as the interpolated frame quantizers. This is to keep the replenishment frames of high enough quality to avoid excess bits wasted for the interpolated frames. Unfortunately, this method of rate control does not adequately encompass the different relative rates for intraframes and interpolated frames. Rate control is extremely important to obtaining the best possible quality for encoding. Clearly much better algorithms than this one exist, some

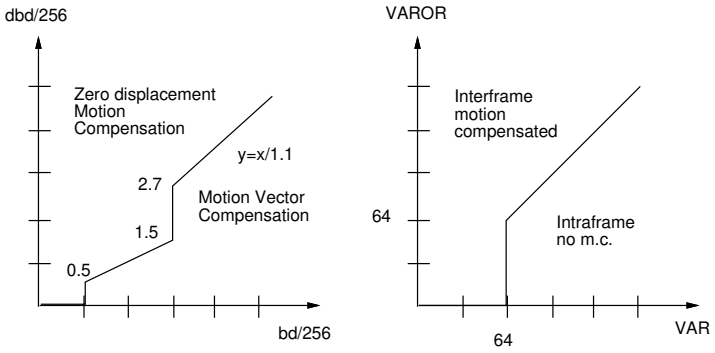

Figure 2.7: The default decision functions for enabling motion compensation and deciding whether motion vectors should be sent.

involving image adaptive quantization; the interested reader might look at [9].

2.13 DC Intraframe Mode

There is a special MPEG transmission mode which has substantial compression with high loss; this mode is called DC intraframe mode. The DC intraframe coding is a sequence of intraframes with only the DC transform components of each block coded - no AC transform components are quantized or coded. No predicted or interpolated frames are used, and this mode is used for the *entire* video sequence, if used at all.

Chapter 3

Calling Parameters

The calling parameters are specified for the UNIX operating system. According to standard convention, when we describe an option it is placed between square brackets.

3.1 Encoder

The calling parameters to start the mpeg encoder is

```
mpeg [-d] [-NTSC] [-CIF] [-QCIF] [-PF] [-NPS] [-MBPS mbps] [-UTC]
     [-a StartNumber] [-b EndNumber]
     [-h HorizontalSize] [-v VerticalSize]
     [-f FrameInterval] [-g GroupInterval]
     [-4] [-c] [-i MCSearchLimit] [-o] [-p PictureRate]
     [-q Quantization] [-r Target Rate]
     [-s StreamFile]  [-x Target Filesize] [-y]
     [-z ComponentFileSuffix i]
     ComponentFilePrefix1 [ComponentFilePrefix2]
-NTSC (352x240)  -CIF (352x288) -QCIF (176x144) base filesizes.
-PF is partial frame encoding/decoding...
    is useful for files horizontalxvertical sizes not multiple of 16
    otherwise files are assumed to be multiples of 16.
-NPS is not-perfect slice (first/end) blocks not same...
-MBPS mbps: is macroblocks per slice.
-UTC: Use time code - forces frames to equal time code value
-d enables the decoder
-a is the start filename index. [inclusive] Defaults to 0.
-b is the end filename index. [inclusive] Defaults to 0.
 overiding -NTSC -CIF -QCIF
Dimension parameters:
  -h gives horizontal size of active picture.
  -v gives vertical size of active picture.
  -f gives frame interval - distance between intra/pred frame (def 3).
  -g gives group interval - frame intervals per group (def 2).

-4 used to specify if DC intraframe mode is used for encoding.
-c is used to give motion vector prediction. (default: off).
-i gives the MC search area: between 1 and 31 (default 15).
-o enables the interpreter.
-p gives the picture rate (see coding standard; default 30hz).
```

```
-q denotes Quantization, between 1 and 31.
-r gives the target rate in bits per second.
-s denotes StreamFile, which defaults to ComponentFilePrefix1.mpg
-x gives the target filesize in bits. (overrides -r option.)
-y enables Reference DCT.
-z gives the ComponentFileSuffixes (repeatable).
```

We examine these parameters, case by case, below.

- `-NTSC` indicates that the source frame size for the Luminance component is 352x240 and for the Chrominance components is 176x120. This is the default size.
- `-CIF` indicates that the source frame size for the Luminance component is 352x288 and for the Chrominance components is 176x144.
- `-QCIF` indicates that the source frame size for the Luminance component is 176x144 and for the Chrominance components is 88x72.
- `-PF` indicates that the component sizes specified by `-h` and `-v` (or read from the compressed stream) are not multiples of 16 and furthermore that the input files (or output files) should be exactly that specified by the component size `-h` and `-v` (or as read from the compressed stream). Otherwise the input and output files will be the component size rounded up to a multiple of 16 in each dimension.
- `-NPS` The NPS option allows the encoder to encode without having the restriction that the last macroblock in a slice cannot be skipped. For coding purposes, the first macroblock in a slice must be coded, although the decoder will probably handle that case as well.
- `-MBPS` This specifies the macroblocks per slice. If left unspecified then the number of macroblocks per slice is the number of macroblocks on one row of the image.
- `-UTC` Forces the frame number of the image to match that of the Group-of-Pictures time code.
- `-a` denotes the following value to be the start of the filename index [inclusive]. Defaults to 0.
- `-b` denotes the following value to be the end of the filename index [inclusive]. Defaults to 0.
- `-h` is the horizontal size of the picture in terms of the luminance component, used for a non `-NTSC`, `-CIF`, `-QCIF` filesize. This value is coded directly to the compressed stream. If it is not a multiple of 16, and `-PF` is not used, then the input *file* dimension is rounded up to a multiple of 16.
- `-v` is the vertical size of the picture in terms of the luminance component, used for a non `-NTSC`, `-CIF`, `-QCIF` filesize. This value is coded directly to the compressed stream. If it is not a multiple of 16, and `-PF` is not used, then the input *file* dimension is rounded up to a multiple of 16.
- `-f` is the number of interpolated frames between each intra/predicted frame. `FrameInterval-1` is the number of interpolated frames within each frame interval.
- `-g` is the number of frame intervals per group-of-pictures. With a group interval of 1 then there are no predicted frames. The value `GroupInterval-1` is the number of predicted frames in each group of pictures.
- `-4` is used to DC Intraframe mode where only the DC components of each picture block is coded.
- `-i` is the diameter of the motion estimation search. The diameter must be inclusively between 1 and 15, designating a region from interframe (0,0) search, to (approximately) full search $(\pm 7.5, \pm 7.5)$, on a half pel grid. The internal motion estimation search is in a radial pattern starting from (0,0) outwards and is somewhat difficult to explain. The motion estimation telescopes for a frame distance greater than one; in other words, the previous frame's motion vector is used as a starting point for the new search.

- `-o` signals that the program interpreter will read the control algorithms from the standard input.
- `-p` specifies the picture rate by a special code. It must be one of the coding standard specifications: 0=forbidden; 1=23.976 Hz; 2=24 Hz; 3=25 Hz; 4=29.97 Hz; 5=30 Hz; 6=50 Hz; 7=59.94 Hz; 8=60 Hz (default 5=30Hz).
- `-q` gives a value for the quantization *not in the presence of rate control, which automatically changes the quantization values*. If rate control is specified, this parameter gives a value for the initial quantization of the first frame (which the program usually takes an educated guess at).
- `-r` specified a rate for the coded stream. If this is enabled, a buffer model is used to limit the size of the coding stream. Note this is given in bits per second.
- `-s` specifies the filename to store the coded image. If unspecified it defaults to `ComponentFilePrefix0.mpg`.
- `-x` gives a target filesize for the compressed stream. This overrides the rate option `-r`, if specified. This is specified in bits. The actual implementation of this technique is through calculating the equivalent bit-rate. Since the buffer usually has some contents at the end of coding, the final filesize is larger than actually designated.
- `-y` enables the double-precision floating point Reference DCT. The default is the Chen DCT.
- `-z` denotes the component file suffixes in sequential order. For example `-z .y.clr -z .u.clr -z .v.clr` indicates that the luminance and the two chrominance files end with a suffix of `.y.clr`, `.u.clr`, and `.v.clr`, in that order. If unspecified, the suffixes default to `.Y`, `.U`, and `.V`.
- `ComponentFilePrefix`*n* must be specified. In general, the other component files share the same prefix and will default to `ComponentFilePrefix1` if not explicitly specified. However, in some cases the prefix is what changes in the file and we can specify this individually.

The following is an example of specifying the file prefixes: Suppose that the luminance and two chrominances are specified with filenames such as `Lum.`*i*`.mum` `Chrom1.`*i*`.lily` `Chrom2.`*i*`.snapdragon` where *i* is a numerical index that changes with each frame. Then we use the following options:

```
Lum. Chrom1. Chrom2. -z .mum -z .lily -z .snapdragon
```

to specify the prefix and the suffix of the files correctly. Again, the convention that we use is `.Y`, `.U`, `.V`.

3.1.1 What are the files?

The input and output files are raw raster-scan files in the YUV color space stored in unsigned 8 bit characters. There are no headers on these files, so a 352x240 file has a length of exactly 84480 bytes. Formulas for the conversion between colorspaces are given in Chapter 2.

3.1.2 Encoding Examples

The simplest coding example is as follows. Suppose we had a sequence of input files `foo0.Y, foo0.U, foo0.V, foo1.Y, foo1.U, foo1.V, foo2.Y, foo2.U, foo2.V` of size 352x240 Y; 176x120 U; 176x120 V.

```
mpeg -a 0 -b 2 foo
```

would take input files `foo0.Y, foo0.U, foo0.V, foo1.Y, foo1.U, foo1.V, foo2.Y, foo2.U, foo2.V` and place them in the compressed stream `foo.mpg`. If one wanted to add rate control of 500 kbits/second and put the output in the file `foo.mpg`.

```
mpeg -a 0 -b 2 foo -s foo.mpg -r 500000
```

For a fixed filesize of 500kbits, use the following command:

```
mpeg -a 0 -b 10 foo -s foo.mpg -x 500000
```

Let's look at luminance dimensions not a multiple of 16. Suppose that the `foo` files were 160x120 Y; 80x60 U; 80x60 V. (We use the `-PF` option to indicate that the sources are not rounded up to the suitable 16 pel boundaries.)

```
mpeg -a 0 -b 2 -h 160 -v 120 foo -s foo.mpg -x 500000 -PF
```

Suppose the video sequence is supposed to be just intra-frames. Then the group interval would be 1 and the frame interval would be 1. The following command does this.

```
mpeg -a 0 -b 10 foo -s foo.mpg  -g 1 -f 1
```

Suppose no predicted frames are desired, just intraframes and interpolated frames, with a frame interval of 6. The following command can be used.

```
mpeg -a 0 -b 10 foo -s foo.mpg  -g 1 -f 6
```

3.2 Decoder

The decoding parameters are fewer than the encoding parameters because much of the information is already present in the compressed stream. The decoder is enabled with the specification `-d`.

```
mpeg  -d -s StreamFile [-a StartNumber] [-y] [-PF] [-UTC]
     [-z ComponentFileSuffix-i]
     ComponentFilePrefix-1 [ComponentFilePrefix-2]
     [ComponentFilePrefix-3]
```

The `StartNumber` parameter designates the start of numbering for the frames decoded.
Specifying any of the other encoder parameters will have no effect.

3.2.1 Decoding Examples

In the first example in section 3.1.2, the command

```
mpeg -d -s foo.mpg bar
```

would put the output files in `bar0.Y, bar0.U, bar0.V, bar1.Y, bar1.U, bar1.V, bar2.Y, bar2.U, bar2.V`. If one wanted to offset the files by 10, one can type

```
mpeg -d -s foo.mpg bar -a 10
```

which would put the output files in `bar10.Y, bar10.U, bar10.V, bar11.Y, bar11.U, bar11.V, bar12.Y, bar12.U, bar12.V`.

Finally, with the non-integral value example before, supposing that the `foo` files were encoded 160x120 Y; 80x60 U; 80x60 V. (We use the `-PF` option to indicate that the destination files should *not* be rounded up to the suitable 16 pel boundaries.)

```
mpeg -d -s foo.mpg -PF
```

3.3 Program Trace

Let's examine a typical output from coding the first seven frames of the color NTSC-CIF sequence table tennis.

We used the command

```
mpeg -a 0 -b 6 tennis/stennis. -s short.mpg
```

which generated the following output:

```
Image Dimensions: 352x240   MPEG Block Dimensions: 352x240
START>SEQUENCE
START>Intraframe: 0
Total No of Bits:   223477  Bits for Frame:   223477
MB Attribute Bits:    660  MV Bits:      0   EOB Bits:   3960
Y Bits:  196206  U Bits:    9386  V Bits:   12432  Total Bits:  218024
MV StepSize: 4.000000  MV NumberNonZero: 23.665152  MV NumberZero: 40.334848
Code MType:     0    1
Macro Freq:   330    0
Y     Freq:  1320    0
UV    Freq:   660    0
Comp: 0  MSE: 19.91  SNR: 29.19  PSNR: 35.14 Entropy: 6.65
Comp: 1  MSE: 5.09  SNR: 33.73  PSNR: 41.06 Entropy: 4.06
Comp: 2  MSE: 5.13  SNR: 36.30  PSNR: 41.03 Entropy: 4.74
Loading file: tennis/stennis.0.Y
Loading file: tennis/stennis.1.Y
Loading file: tennis/stennis.2.Y
Loading file: tennis/stennis.3.Y
Doing Forward: 1
Doing Forward: 2
Doing Forward: 3
Doing Backward: 2
Doing Backward: 1
START>Predicted: 3
Total No of Bits:   369237  Bits for Frame:   145760
MB Attribute Bits:   3110  MV Bits:    349   EOB Bits:   3358
Y Bits:  128457  U Bits:    4189  V Bits:    5940  Total Bits:  138586
MV StepSize: 4.000000  MV NumberNonZero: 13.789394  MV NumberZero: 50.210606
Code MType:     0    1    2    3    4    5    6
Macro Freq:    63  247    1   19    0    0    0
Y     Freq:   231  946    0   76    0    0    0
UV    Freq:   114  274    0   38    0    0    0
Comp: 0  MSE: 10.23  SNR: 32.07  PSNR: 38.03 Entropy: 6.57
Comp: 1  MSE: 4.88  SNR: 33.90  PSNR: 41.24 Entropy: 3.93
Comp: 2  MSE: 4.74  SNR: 36.65  PSNR: 41.37 Entropy: 4.63
```

```
START>Interpolated: 1
Total No of Bits:   386314  Bits for Frame:    17077
MB Attribute Bits:   3019  MV Bits:   1320   EOB Bits:    704
Y Bits:   12222  U Bits:      99  V Bits:     333  Total Bits:   12654
MV StepSize: 8.000000  MV NumberNonZero: 0.901010  MV NumberZero: 63.098990
Code MType:     0    1    2    3    4    5    6    7    8    9   10
Macro Freq:    75  200    1    6   14   32    2    0    0    0    0
Y     Freq:     0  209    0   17    0   97    8    0    0    0    0
UV    Freq:     0   11    0    2    0    4    4    0    0    0    0
Comp: 0  MSE: 19.44  SNR: 29.30  PSNR: 35.24 Entropy: 6.59
Comp: 1  MSE: 5.21  SNR: 33.62  PSNR: 40.96 Entropy: 3.96
Comp: 2  MSE: 5.75  SNR: 35.81  PSNR: 40.54 Entropy: 4.66
START>Interpolated: 2
Total No of Bits:   406219  Bits for Frame:    19905
MB Attribute Bits:   3528  MV Bits:   1580   EOB Bits:    850
Y Bits:   14628  U Bits:       5  V Bits:     190  Total Bits:   14823
MV StepSize: 8.000000  MV NumberNonZero: 0.986869  MV NumberZero: 63.013131
Code MType:     0    1    2    3    4    5    6    7    8    9   10
Macro Freq:    55  234    5   20    1   15    0    0    0    0    0
Y     Freq:     0  311    0   64    0   39    0    0    0    0    0
UV    Freq:     0    9    0    2    0    0    0    0    0    0    0
Comp: 0  MSE: 21.28  SNR: 28.90  PSNR: 34.85 Entropy: 6.58
Comp: 1  MSE: 5.16  SNR: 33.66  PSNR: 41.00 Entropy: 3.93
Comp: 2  MSE: 5.62  SNR: 35.91  PSNR: 40.63 Entropy: 4.64
Loading file: tennis/stennis.3.Y
Loading file: tennis/stennis.4.Y
Loading file: tennis/stennis.5.Y
Loading file: tennis/stennis.6.Y
Doing Forward: 1
Doing Forward: 2
Doing Forward: 3
Doing Backward: 2
Doing Backward: 1
START>Intraframe: 6
Total No of Bits:   628441  Bits for Frame:   222222
MB Attribute Bits:    660  MV Bits:      0   EOB Bits:   3960
Y Bits:  195319  U Bits:    9209  V Bits:   12328  Total Bits:  216856
MV StepSize: 4.000000  MV NumberNonZero: 23.657576  MV NumberZero: 40.342424
Code MType:     0    1
Macro Freq:   330    0
Y     Freq:  1320    0
UV    Freq:   660    0
Comp: 0  MSE: 19.83  SNR: 29.19  PSNR: 35.16 Entropy: 6.64
Comp: 1  MSE: 5.01  SNR: 33.79  PSNR: 41.13 Entropy: 4.06
Comp: 2  MSE: 5.25  SNR: 36.22  PSNR: 40.93 Entropy: 4.76
START>Interpolated: 4
Total No of Bits:   642636  Bits for Frame:    14195
MB Attribute Bits:   3238  MV Bits:   1720   EOB Bits:    478
Y Bits:    9270  U Bits:       6  V Bits:     516  Total Bits:    9792
MV StepSize: 8.000000  MV NumberNonZero: 0.771212  MV NumberZero: 63.228788
Code MType:     0    1    2    3    4    5    6    7    8    9   10
Macro Freq:   170  132    0    1    6   21    0    0    0    0    0
Y     Freq:     0  155    0    4    0   56    0    0    0    0    0
UV    Freq:     0   17    0    1    0    6    0    0    0    0    0
Comp: 0  MSE: 16.92  SNR: 29.88  PSNR: 35.85 Entropy: 6.54
Comp: 1  MSE: 4.59  SNR: 34.17  PSNR: 41.51 Entropy: 3.89
Comp: 2  MSE: 5.69  SNR: 35.86  PSNR: 40.58 Entropy: 4.64
START>Interpolated: 5
Total No of Bits:   668143  Bits for Frame:    25507
MB Attribute Bits:   4048  MV Bits:   1640   EOB Bits:   1202
Y Bits:   19183  U Bits:      27  V Bits:     354  Total Bits:   19564
MV StepSize: 8.000000  MV NumberNonZero: 1.186364  MV NumberZero: 62.813636
Code MType:     0    1    2    3    4    5    6    7    8    9   10
Macro Freq:    38  201    6   71    3   11    0    0    0    0    0
Y     Freq:     0  389    0  169    0   25    0    0    0    0    0
UV    Freq:     0    9    0    4    0    5    0    0    0    0    0
Comp: 0  MSE: 22.99  SNR: 28.55  PSNR: 34.51 Entropy: 6.57
```

```
Comp: 1  MSE: 4.85  SNR: 33.93  PSNR: 41.28 Entropy: 3.94
Comp: 2  MSE: 5.63  SNR: 35.91  PSNR: 40.63 Entropy: 4.67
END>SEQUENCE
Number of buffer overflows: 0
```

An itemized annotation of the sequence follows.

- **For image size:**
  - `Image Dimensions:` The dimensions of the image read from the stream.
  - `MPEG Block Dimensions:` The dimensions of the image rounded up to the nearest macroblock, which is the actual image dimension transmitted by MPEG.
- **For rate control:**
  - `Rate:` This specifies the rate in kilobits per second of the coding stream.
  - `QDFact:` This is the division factor of the Buffer to find the next quantization step.
  - `QOffs:` This is the offset of the quantization. Normally this is 1 so that there is no *zero* quantization stepsizes.
- `Total No of Bits:` represents the total number of bits coded so far. Of course the final stream file rounds the number of bits up to an integral multiple of eight.
- `Bits for Frame:` represents the number of bits to code the present frame.
- `Buffer Contents:` represents the level currently in the buffer.
- `Buffer Size:` represents the size of the buffer. Defined as the greater of the targeted rate per second/4 or 320 kbits.
- `MB Attribute Bits:` represents the number of bits coded for macroblock attributes (this includes Motion Vectors).
- `MV Bits:` represents the number of bits necessary to code the Motion Vectors.
- `EOB Bits:` represents the number of bits coding End of Blocks.
- `Y Bits:` represents the number of bits spent on coding luminance coefficients (excluding End of Blocks).
- `U Bits:` represents the number of bits spent on coding U chrominance coefficients (excluding End of Blocks).
- `V Bits:` represents the number of bits spent on coding V chrominance coefficients (excluding End of Blocks).
- `Total Bits:` is the sum of the Y Bits, the U Bits, and the V Bits.
- `MV StepSize:` represents the mean value of the quantizer step-size parameter (not the step-size itself) as it is adapted throughout the coding.
- `MV NumberNonZero:` represents the mean value of non-zero coefficients per block in the coding.
- `MV NumberZero:` represents the mean value of zero coefficients per block in the coding. This value should be (64 - `MV NumberNonZero`).
- `Code MType:` represents the coding method in numeric order horizontally for the following frame types: for intraframe blocks as `Intra, Intra+Q`; for predicted frames `MC+Pattern, Inter+Pattern, MC, Intra, MC+Pattern+Q, Inter+Patter+Q, Intra+Q`; for interpolated frames `FMC+BMC, FMC+BMC+Pattern, BMC, BMC+Pattern, FMC, FMC+Pattern, Intra, FMC+BMC+Pattern+Q, FMC+Pattern+Q, BMC+Pattern+Q, Intra+Q`.

  - `Macro Freq:` represents the macroblock frequency for each single coding method.
  - `Y Freq:` represents the luminance frequency for each single coding method.
  - `UV Freq:` represents the chrominance frequency for each single coding method.

- `Comp:` represents the component number (reading horizontally.)
  - `MSE:` represents the Mean Squared Error of that particular component.
  - `MRSNR:` represents the (mean-removed) signal to noise ratio weighted on the original signal's variance.
  - `SNR:` represents the signal to noise ratio weighted on the original signal's squared value.
  - `PSNR:` represents the peak signal to noise ratio, the precise metric used by mpeg. The signal is weighted on maximum signal power, i.e. 255^2.
  - `Entropy:` is the probabilistic independent source signal entropy as calculated by $\sum p \log p$.

- `Number of buffer overflows:` represents the number of times the buffer overflows and a zero motion vector, zero coefficient block is forced to be coded.

3.4 Return Values

The return value on successful completion is a 0. (Warning not all of these are followed exactly.) Other values indicate errors as follows:

```
ERROR_NONE 0
ERROR_BOUNDS 1            /*Input Values out of bounds */
ERROR_HUFFMAN_READ 2      /*Huffman Decoder finds bad code */
ERROR_HUFFMAN_ENCODE 3    /*Undefined value in encoder */
ERROR_MARKER 4            /*Error Found in Marker */
ERROR_INIT_FILE 5         /*Cannot initialize files */
ERROR_UNRECOVERABLE 6     /*No recovery mode specified */
ERROR_PREMATURE_EOF 7     /*End of file unexpected */
ERROR_MARKER_STRUCTURE 8  /*Bad Marker Structure */
ERROR_WRITE 9             /*Cannot write output */
ERROR_READ 10             /*Cannot write input */
ERROR_PARAMETER 11        /*System Parameter Error */
ERROR_MEMORY 12           /*Memory exceeded */
```

Chapter 4

The Program Interpreter

The program interpreter is used to define the filenames and create new routines to use for coding decisions.

4.1 Terminology

We use the following terminology to specify the grammar of the language.

- Braces $\{\}$ are used to delimit expressions. In the `Definition:` section, a set of values between two braces indicates a range of occurrences of the prefix. For example, $a\{3,5\}$ indicates 3 to 5 occurrences of a. Furthermore, in the `Syntax:` section, a pair of double braces with empty contents represents the bottom of the stack. In the `Example:` section contents indicated within a brace indicate the standard output response to a command.
- The * operator is the Kleene operator and designates zero or more occurrences of the prefix. Basically, a*, means zero or more occurrences of a, the set $\{\epsilon, a, aa, aaa, aaaa, ...\}$.
- The + operator designates one or more occurrences of the prefix. Thus a+ is the set $\{a, aa, aaa, ...\}$–the same as aa*.
- Brackets [] are an overloaded definition. In some instances it is used to delimit an optional argument. Thus [a] means that the occurrence of "a" is optional. This is consistent with Extended Backus-Naur Form (EBNF). In terms of the `Definition` they also indicate a range. For example, [a-z] represents all elements from a to z, and [^a] means all characters not equivalent to a. This is consistent with UNIX lexical convention. Finally, these are also valid tokens.
- Parentheses () are an overloaded definition. It is used for grouping in the lexical context where a | indicates an "or" function. For example $(a|b)$ indicates that either a or b can be used. For cases involving an id within the parenthesis, it represents a particular stack element on the internal stack.

4.2 Tokens

A token is the basic unit or word in the vocabulary of the program interpreter. It can be of several sorts.

A token can be an English word or a concatenated English word such as "STREAMNAME." For English words the program interpreter is case insensitive. That means "StReamNaME," "STREAMNAME," and "streamname" are all identical to the interpreter.

A token can be an integer, which is defined by the following nomenclature:

```
Definition: Digit -> [0-9]
            HexDigit -> ({Digit}|[a-fA-F])
```

```
            OctalDigit -> [0-7]
            DecInteger -> {Digit}+
            HexInteger -> 0[xX]{HexDigit}+
            OctInteger -> 0[oO]{OctalDigit}+
            HexInteger2 -> {HexDigit}+[Hh]
            OctInteger2 -> {OctalDigit}+[BCObco]
            CharInteger -> '([^\\]|\\([\n^\n]|{OctalDigit}{1,3}))'
Syntax:     Decimal-Integer = digits
            Hex-Integer = 0xhexdigits 0Xhexdigits
                    (hexdigits)h (hexdigits)H
            Octal-Integer = 0ooctdigits 0Ooctdigits
                    (octdigits)b (octdigits)B
                    (octdigits)c (octdigits)C
                    (octdigits)o (octdigits)O
            Character-Integer = 'character'

Example:    10 0x10 10H 10B '\n' 'G'
```

A token can be a real number, defined as below

```
Definition: ScaleFactor [eE][-+]?{Digit}+
            Real1 ({Digit}+"."{Digit}*({ScaleFactor})?)
            Real2 ({Digit}*"."{Digit}+({ScaleFactor})?)
            Real3 ({Digit}+{ScaleFactor})
            Real {Real1}|{Real2}|{Real3}
Syntax:     C-Real and Fortran-Real
Example:    .25 0.25e12
```

A token can be a string, which is defined according to the following nomenclature:

```
Definition: String -> \"([^\"]|\\\")*\"
Syntax:     "non-escaped-double-quote-character"
Example:    "Hello George"
            "Lord of the Flies"
            "Quotes \"\" are fun\n"
```

A token can be a comment (not a regular expression because it nests) and can be written by C's matching comment pairs "/* Comment-text */".

A token can be braces [], denoting some sort of array or repeated structure.

4.2.1 Special Characters

A special character in the definition of a character-integer or a string can be written with the \ escape character.

The escape character singles out the following character sequence as being special. The following *one characters* represent common text control characters.

- **b:** backspace.
- **i:** (horizontal) tab.
- **n:** newline.

- **v:** (vertical) tab.
- **f:** form feed.
- **r:** carriage return.
- **0:** null.

If the character is greater than one single digit (and less than or equal to three digits), it must be an octal code. An octal code is written as a sequence of two or more octal digits. Care must be taken when imbedding octal codes inside other numerical digits. Thus `\466` is undefined (exceeds 255) and `"\0466"` is the equivalent of `"&6"`.

Otherwise, the character after the \ is taken verbatim.

4.3 Comments

First of all, we start out by defining what a comment is. A comment begins with a `/*` and ends with a `*/`. They nest, hence, the following examples are valid comments.

```
Definition: S -> S S |
                 /* (string!=*/) S (string!=*/) */ |
                 /* (string!=*/) */
Syntax:     None.
Example:    E/* Comment /* These comments,
            unlike those of C, nest... *//**/ */
            /* Simple Comment */
```

4.4 Aborting

The entire program can be aborted by the following command to the interpreter, causing an unconditional exit.

```
Syntax: ABORT
Example: ABORT
```

4.5 Structure Definition

The structure must be specified before the streams can be opened and such structure definitions cannot be placed as valid program commands. The specification of the stream name is given by

```
Definition: S -> STREAMNAME string
Syntax:     STREAMNAME Filename

Example:    streamname "Output.Compressed"
```

Each “image” component, or color, resides on a separate file with a given prefix and suffix. This is specified by

```
Definition: S -> COMPONENT
             ( integer [string string] |
              [ {integer [string string]}+ ] )
Syntax:     COMPONENT Index [Fileprefix FileSuffix]
            COMPONENT [
            Index1 [Fileprefix1 FileSuffix1]
                        ...
            Indexn [Fileprefixn FileSuffixn]
                        ]
Example:    component 0 ["name" ".Y.clr"]
            component [0 ["name" ".Y.clr"]
                1 ["name" ".U.clr"]
                2 ["name" ".V.clr"]]
```

The image type can be changed to CIF, QCIF, or NTSC by the following commands.

```
Syntax: CIF
Syntax: QCIF
Syntax: NTSC
Example: CIF
```

The quantization for non-rate controlled images and the initial quantization for rate controlled images is changed by

```
Definition: QUANTIZATION integer
Syntax:     QUANTIZATION number
Example:    QUANTIZATION 10
```

The number for the quantization should be between 1 and 31 inclusively.

4.6 Invalid commands

The following commands have been disabled because of lack of support.

The picture rate and frame skip definitions which change the rate control and file access for programs can be changed by

```
Definition: (PICTURERATE | FRAMESKIP) integer
Syntax:     PICTURERATE rate
Syntax:     FRAMESKIP interval
Example:    picturerate 7 frameskip 2
```

Also the motion compensation search limit can be changed by

```
Definition: SEARCHLIMIT integer
Syntax:     SEARCHLIMIT range
Example:    searchlimit 15
```

The range acceptable is between 1 and 15, representing a motion compensated search of [0,0] to [$\pm$7.5,$\pm$7.5].

4.7 Printing

The printing commands display the status of the program to the standard output device. They are all valid program commands. The PRINTPROGRAM command prints out the program to the standard output.

```
Definition: S -> PRINTPROGRAM integer
Syntax:     PRINTPROGRAM index
Example:    open 0 0 sto TYPE3 exit close printprogram 0
            {0: VAL 0.000000
             2: STO 4
             4: EXIT}
```

The printstack command outputs the current contents of the stack.

```
Definition: S -> PRINTSTACK
Syntax:     PRINTSTACK
Example:    {} 2.0 printstack {2.0}
```

The ECHO command prints out a string to the standard output. The echo command is not imbeddable in a program.

```
Definition: S -> ECHO string
Syntax:     ECHO "printable-characters"
Example:    echo "Play it, Sam, play it again."
               {Play it, Sam, play it again.}
```

4.8 Arithmetic Commands

All arithmetic commands operate on double precision numbers which are stored on an internal stack much like the operator of RPN (Reverse Polish Notation) on a HP calculator. The reason that we use double precision floating point representation is that it represents higher precision at insignificant overhead. These commands are all valid program operations.

A value is entered onto the stack by an integer or a real token. The integer value is forced to a real by the interpreter.

```
Syntax: integer -> (real)
Syntax: real -> (real)
Example: 1 2.3 .4 printstack {1.0 2.3 0.4}
```

These math commands operate on the top element of the stack.

```
Syntax: (stack) NEG  -> (-stack)
Syntax: (stack) SQRT  -> (sqrt(stack))
Syntax: (stack) ABS  -> (abs(stack))
Syntax: (stack) FLOOR -> (greatest_integer(stack))
Syntax: (stack) CEIL -> (least_integer(stack))
Syntax: (stack) ROUND -> (nearest_integer(stack))
Example:  3.5 round sqrt -3.0 abs printstack {2.0 3.0}
```

The math commands operate on the top two elements of the stack.

```
Syntax: (stack1) (stack2) ADD -> (stack1+stack2)
Syntax: (stack1) (stack2) SUB -> (stack1-stack)
Syntax: (stack1) (stack2) MUL -> (stack1*stack2)
Syntax: (stack1) (stack2) DIV -> (stack1/stack2)
Example:  3 5 add 1 sub printstack  {7.0}
```

The logical commands return a 1.0 or a 0.0 depending on whether the logical operations are true or false. For the purpose of logical values, a zero value is considered a false and a non-zero value is considered true.

```
Syntax: (stack) NOT  -> (!(stack))
Syntax: (stack1) (stack2) AND -> ((stack1)&&(stack2))
Syntax: (stack1) (stack2) OR -> ((stack1)||(stack))
Syntax: (stack1) (stack2) XOR -> ((stack1)^^(stack2))
Syntax: (stack1) (stack2) LT -> (stack1<stack2)
Syntax: (stack1) (stack2) LTE -> (stack1<=stack2)
Syntax: (stack1) (stack2) GT -> (stack1>stack2)
Syntax: (stack1) (stack2) GTE -> (stack1>=stack2)
Syntax: (stack1) (stack2) EQ -> (stack1==stack2)
Example:  3 5 gte 3 3 eq printstack {0.0 1.0}
```

4.9 Stack Commands

The stack commands manipulate the elements of the stack.

```
Syntax: (stack) DUP -> (stack) (stack)
Syntax: (stack) POP -> -
Syntax: (stack1) (stack2) EXCH -> (stack2) (stack1)
Syntax: (stack1) ... (stackn) COPY n ->
               (stack1) ... (stackn)
               (stack1) ... (stackn)
Syntax: (stackr) ... (stack1) ROLL r n ->
               (stackn-1modr+1)...(stacknmodr+1)
Syntax: (stackn)...(stack1) INDEX n -> (stackn)...(stack1) (stackn)
Syntax: {} (stack)* CLEAR -> -
Example: 1 dup dup printstack {1.0 1.0 1.0} pop pop pop printstack {}
```

4.10 Memory Access Commands

There is a central location where all of the memory values are located. The size of this memory defaults to 1024 registers, each register containing a double value and indexed by the location in Memory (from 0 to 1023).

```
Syntax: (stack) STO Index -> -
Syntax: RCL Index -> (value of memory: Index)
Example: 10 sto 1 rcl 1 printstack {10.0}
```

4.11 Program Access Commands

To open and close a program we use `OPEN` and `CLOSE`. To exit the program (one must be designated explicitly), we use the command `EXIT`. Any program command between the `OPEN` and the `CLOSE` are part of the body of the program, thus `OPEN` and `CLOSE` are not valid program operators though `EXIT` is valid. Each program can store up to 500 commands.

```
Syntax: OPEN programnumber
Syntax: EXIT
Syntax: CLOSE
Example: open 0 1 1 add exit close
```

In general, the `programnumber` varies between 0 and 9. The first allocation (0) is reserved for the coding method choice algorithm. The second location (1) is reserved for the quantization algorithm. The other locations can be used as executable routines by the first two locations.

To execute a program, we use the the command `EXE`. This is a valid program command and programs can call each other. It has form of

```
Syntax: EXE programnumber
Example: open 0 1 exit close printstack {} exe 0 printstack {1.0}
         clear open 2 exe 0 exe 0 exe 0 exit close exe 2
         printstack {1.0 1.0 1.0}
```

4.12 Labels and Equivalences

A label is a string identifier of a particular point in the program and follows the same syntax for a normal program identifier (e.g. case independent). When defining a program, a label associates a position with the following command so that a branch can identify with that program location. We usually denote a label with a identifier followed by a colon; for example, `MYLABEL:`.

An equivalence is similar to a label but works for memory locations. To define an equivalence, use

```
Syntax: (stack) EQU identifier
Example: 24 equ x
```

Then for all programs afterwards, every store and recall using `x` uses the memory location 24.

Although a label is strictly a program-related concept, once defined, an equivalence can be used throughout the program interpreter.

4.13 Branch Commands

```
Syntax: - GOTO Label -> -
Syntax: (stack) IFG Label -> -
Syntax: (stack) IFNG Label -> -
Example:  open 5
          ifg DONE:
          100
          DONE: exit
          close
          1.0 exe 5 printstack {100.0}
```

The branch commands occur only in program fragments. The `GOTO` command transfers the program control to the instruction following the label designated. The `IFG` command transfers control to the Label if the top value on the stack is non-zero. The `IFNG` command transfers command to the label if the top value on the stack is zero.

4.14 Reserved Memory Locations

The first 20 memory locations are reserved and should not be used by the program. They are represented by the following labels and cannot be renamed for other purposes.

- `SQUANT` The quantization coefficient for the slice. This is always transmitted per slice and should be modified by the quantization routine.
- `MQUANT` The quantization coefficient specified on the macroblock level which changes the quantization for all further macroblocks until the next slice definition. Currently, quantization on this fine level is not supported.
- `PTYPE` This is the picture type. It is 1,2,3,4 depending on being an intra-frame, predicted frame, interpolated frame, or DC Intracoded frame. Ptype 0 is not allowed.
- `MTYPE` this is the macroblock type. It can be of several classes depending on the picture type. This should be changed by the block coding determination routine.
  - **PType=1** Picture type of intra frame.
    - **MType=0** Intraframe block coded.
    - **MType=1** `MQUANT` used; Intraframe block coded.
  - **PType=2** Picture type of predicted frame.
    - **MType=0** Forward prediction; Coded Block Pattern.
    - **MType=1** Interframe-Compensation; Coded Block Pattern.
    - **MType=2** Forward prediction; No data coded.
    - **MType=3** Intraframe block coded.
    - **MType=4** `MQuant` used; Forward prediction; Coded Block Pattern.
    - **MType=5** `MQuant` used; Interframe-Compensation; Coded Block Pattern.
    - **MType=6** `MQuant` used; Intraframe block coded.
  - **PType=3** Picture type of interpolated frame.
    - **MType=0** Interpolative prediction; No data coded.
    - **MType=1** Interpolative prediction; Coded Block Pattern.
    - **MType=2** Backward prediction; No data coded.
    - **MType=3** Backward prediction; Coded Block Pattern.
    - **MType=4** Forward prediction; No data coded.
    - **MType=5** Forward prediction; Coded Block Pattern.
    - **MType=6** Intraframe block coded.
    - **MType=7** `MQuant` used; Interpolative prediction; Coded Block Pattern.
    - **MType=8** `MQuant` used; Forward prediction; Coded Block Pattern.
    - **MType=9** `MQuant` used; Backward prediction; Coded Block Pattern.
    - **MType=10** `MQuant` used. Intraframe block coded.
  - **PType=4** DC Intraframe (no block types).
- `BD` The absolute error difference between the previous frame's block and the current frame's block/256.
- `FDBD` The absolute error difference between the previous reference frame's motion compensated (displaced) block and the current block/256. This is only present in predicted or interpolated frame types.
- `BDBD` The absolute error difference between the next reference frame's motion compensated (displaced) block and the current block/256. This is only present in the interpolated frame type.

- IDBD The absolute error difference between the previous and next reference frame's motion compensated (displaced) block and the current block/256. This is only present in the interpolated frame type.
- VAROR The variance of the original block/256.
- FVAR The variance of the previous frame motion compensated block/256.
- BVAR The variance of the previous frame motion compensated block/256.
- IVAR The variance of the previous frame motion compensated block/256.
- DVAR The variance of zero motion vector replenishment with the previous frame/256.
- RATE The rate in kbits/second to be transmitted by the program.
- BUFFERSIZE The size of the buffer in bits.
- BUFFERCONTENTS The current contents of the buffer in bits.
- QDFACT The buffer division factor to obtain a quantization. Our value is QDFACT = (RATE/230).
- QOFFS The quantization offset after the division factor. Our value is QOFFS = 1.

4.15 Reserved Program Locations

The program location 0 represents the "coding algorithm" and the program location 1 represents the "quantization control algorithm". The quantization program is called only if the rate-control is enabled.

The "coding algorithm" is called at every macroblock step and (intentionally, can be changed in souce code) only affects the internal MTYPE. All the above named variables are set prior to calling the "coding algorithm" with reference values.

The "quantization control algorithm" is called every slice interval and (intentionally, can be changed in code) only affects the internal SQUANT. Only the SQUANT, MQUANT, PTYPE, MTYPE, RATE, BUFFERSIZE, BUFFERCONTENTS, QDFACT, QOFFS variables are set prior to calling the "quantization control algorithm."

4.16 Command List

A summary of all of the commands and their possible occurrences are as follows ([] indicates a non-supported function right now).

Program and Top-Level	Top-Level Only	Program Only
ADD	ECHO	EXIT
SUB	OPEN	GOTO
MUL	CLOSE	IFG
DIV	EQU	IFNG
NOT	STREAMNAME	(ID)
AND	COMPONENT	
OR	[PICTURERATE]	
XOR	[FRAMESKIP]	
LT	QUANTIZATION	
LTE	[SEARCHLIMIT]	
EQ	NTSC	
GT	QCIF	
GTE	CIF	
NEG		
SQRT		
ABS		
FLOOR		
CEIL		
ROUND		
DUP		
POP		
EXCH		
CLEAR		
PRINTSTACK		
PRINTIMAGE		
PRINTFRAME		
COPY		
INDEX		
STO		
RCL		
EXE		
ABORT		
PRINTPROGRAM		
ROLL		
(INTEGER)		
(REAL)		

4.17 Examples

Let's define some decision routines as code for the program interpreter.

4.17.1 Intraframe Coding

If we want to use intraframe coding only, we might write the following C-code.

```
if (PType==1)
  MType = 0;
else if (PType==2)
  MType = 3;
else if (PType==3)
  MType = 6;
```

but we can use the following program

```
/* This program sets intraframe decoding for the sequence regardless of */
/* type of picture frame.  It first checks the picture type, and sets */
/* the macroblock type accordingly */
open 0
  rcl PTYPE
  1 GT
  IFG NEXT1:
  0 sto MTYPE            /* Picture type intra */
  exit
NEXT1:
  rcl PTYPE
  2 GT
  IFG NEXT2:
  3 sto MTYPE            /* Picture type predicted */
   exit
NEXT2:
  6 sto MTYPE            /* Picture type interpolated */
exit
close
```

4.17.2 Reference Block Coding

The reference coding control for the predicted picture type is given by

```
if (PType==2)
  {
    if (((BD < 3.0) && (FDBD > (BD*0.5))) ||  /* Decide between inter/mc */
        ((FDBD > (BD/1.1))))
      {
        MType = 1;                            /* Inter mode */
        if ((DVAR > (double) 64)&&(DVAR > VAROR)) /* error to high, intra */
    MType = 3;
      }
     else
      {
        MType = 0;                             /* MC Mode */
        if ((FVAR > (double) 64)&&(FVAR > VAROR)) /* error to high, intra */
    MType = 3;
      }
  }
```

we require the following program lines to change just the predicted picture type.

```
/* The coding decision used for the predicted frame.  We only change
things if the picture type is for predicted frames.  Otherwise, we don't
change anything and the default C-code routine decision holds. */
open 0
  rcl PTYPE
  2 eq
  ifng END:            /* Not predicted, so don't bother */
  rcl BD 3.0 lt rcl FDBD rcl BD 0.5 mul gt and
  rcl FDBD rcl BD 1.1 div gt or
  ifng DOMC:
  1 sto MTYPE
```

```
  rcl DVAR 64 gt
  rcl DVAR rcl VAROR gt and ifng END:
  3 sto MTYPE
  exit
DOMC:
  0 sto MTYPE
  rcl FVAR 64 gt
  rcl FVAR rcl VAROR gt and ifng END:
  3 sto MTYPE
END: exit
close
```

4.17.3 Reference Rate Control

To figure the simple reference rate control, we use the following routine

```
  SQuant=(BufferContents/QDFact)+QOffs;
  if ((PTYPE == P_INTRA)||(PTYPE==P_PREDICTED)) SQUANT = SQUANT/2;
  if (SQuant<1) SQuant=1;
  else if (SQuant>31) SQuant=31;
```

This has 1/2 smaller quantizer stepsizes for intra and predicted frames than interpolated frames. But we can modify it by the following program which has a more similar stepsize of quantization (1/1.5 smaller) for intra and predicted frames than interpolated frames. This gives a slightly evener rate, though may not be better in a rate-distortion sense.

```
/* The rate control that is in the C code. */
open 1
  rcl BUFFERCONTENTS rcl QDFACT div rcl QOFFS add
  rcl PTYPE 1 eq
  rcl PTYPE 2 eq
  or ifng CLIP:
  1.5 div
CLIP:
  dup 1 lt ifng CHECKHI:  /* Clip <1 */
  pop 1
  goto END:
CHECKHI:
  dup 31 gt ifng END: /* Clip >31 */
  pop 31
END:
  sto SQUANT
  exit
close
```

4.18 Enabling the Program Interpreter: An example

A simple program interpreter example involves coding by intra-frame only. This requires setting the macroblock type to 0 at each coding decision. Suppose we had a sequence of input files `foo0.Y, foo0.U, foo0.V, foo1.Y, foo1.U, foo1.V, foo2.Y, foo2.U, foo2.V` of size 352x240 Y; 176x120 U; 176x120 V; and the file `test.intra` contained the program discussed earlier:

```
/* This program sets intraframe decoding for the sequence regardless of */
/* type of picture frame.  It first checks the picture type, and sets */
/* the macroblock type accordingly */
open 0
  rcl PTYPE
  1 GT
  IFG NEXT1:
  0 sto MTYPE            /* Picture type intra */
  exit
NEXT1:
  rcl PTYPE
  2 GT
  IFG NEXT2:
  3 sto MTYPE            /* Picture type predicted */
   exit
NEXT2:
  6 sto MTYPE            /* Picture type interpolated */
exit
close
```

The command

```
mpeg -a 0 -b 2 foo -s foo.mpg -o < test.intra
```

would use the program above to set the macroblock types to intra-frame coding. Multiple programs can be concatenated together (e.g. using the `cat` command).

Chapter 5

Program Documentation

5.1 Program Flow

The program consists of twelve files. A brief description of the files and their contents are as follows:

- `mpeg.c` This file contains most of the major blocks of the mpeg routines. The entry point `main()` resides in this file as well as the main mpeg decoder and encoder calling routines.
- `lexer.c` This file contains most of the parsing code to drive the program interpreter. The important routine here is called `parser()` and reads in programs from the standard input. The definitions for the program interpreter (maximum program lines (2000), number programs (10), etc, are defined here).
- `transform.c chendct.c` These files contain the DCT transforms, the reference DCT and the Chen DCT. The quantization routines and the zig-zag routines are in the file `transform.c`.
- `codec.c` This file contains the routines to encode the basic DCT block run-length structure.
- `marker.c` This file contains the code to read and write the mpeg markers, e.g. MBS code, etc.
- `io.c me.c` This file contains the top-level memory management for the component files. The images and frame buffers are held in-memory. The block movement and block manipulation routines are found in `io.c`, while the memory comparison and motion estimation routines are found in `me.c`.
- `stat.c` This file contains the routines to figure statistics from the images held in memory.
- `huffman.c` The Huffman encoding and decoding routines are placed here along with the routines to construct suitable Huffman finite state machines from a length, codeword, and associated value.
- `stream.c` The bit-level stream interface is in this file. It is used to access the compressed file.
- `mem.c` The memory management routines are here. That involves routines to save and store and manipulate large areas of memory.

5.2 Functional Description

In order to examine the program on a routine-by-routine basis, we have provided the following documentation which itemizes the routines in each file and describes, briefly, their purpose.

5.2.1 mpeg.c

```
int main(int, char**);
extern void MpegEncodeSequence(void);
extern void MpegDecodeSequence(void);
extern void MpegEncodeIPBDFrame(void);
extern void MpegDecodeIPBDFrame(void);
extern void PrintImage(void);
extern void PrintFrame(void);
extern void MakeImage(void);
extern void MakeFrame(void);
extern void MakeFGroup(void);
extern void LoadFGroup(int);
extern void MakeFStore(void);
extern void MakeStat(void);
extern void SetCCITT(void);
extern void CreateFrameSizes(void);
extern void Help(void);
extern void MakeFileNames(void);
extern void VerifyFiles(void);
extern int Integer2TimeCode(int);
extern int TimeCode2Integer(int);

static void MpegEncodeDSequence(void);
static void ExecuteQuantization(int *);
static void CleanStatistics(void);
static void CollectStatistics(void);

static void MpegEncodeSlice(int);
static void MpegEncodeMDU(void);
static void MpegFindMType(void);
static void MpegCompressMType(void);
static void MpegWriteMType(void);

static void MpegEncodeDFrame(void);
static void MpegDecodeDFrame(void);
static void MpegDecodeSaveMDU(void);
static void MpegDecompressMDU(void);
```

- **main() (mpeg.c:351)** is the first routine called by program activation. It parses the input command line and sets parameters accordingly.
- **MpegEncodeSequence() (mpeg.c:507)** encodes the sequence defined by the CImage and CFrame structures, startframe and lastframe.
- **MpegEncodeDSequence() (mpeg.c:641)** encodes the DC intraframe sequence defined by the CImage and CFrame structures, startframe and lastframe.
- **ExecuteQuantization() (mpeg.c:701)** references the program in memory to define the quantization required for coding the next block.
- **CleanStatistics() (mpeg.c:738)** cleans/initializes the type statistics in memory.
- **CollectStatistics() (mpeg.c:761)** is used to assemble and calculate the relevant encoding statistics. It also prints these statistics out to the screen.

- **MpegEncodeIPBDFrame() (mpeg.c:809)** is used to encode a single Intra, Predicted, Bidirectionally predicted, DC Intra frame to the opened stream.
- **MpegEncodeMDU() (mpeg.c:936)** encodes the current MDU. It finds the macroblock type, attempts to compress the macroblock type, and then writes the macroblock type out. The decoded MDU is then saved for predictive encoding.
- **MpegFindMType() (mpeg.c:969)** makes an initial decision as to the macroblock type used for MPEG encoding.
- **MpegWriteMType() (mpeg.c:1220)** writes a macroblock type out to the stream. It handles the predictive nature of motion vectors, etc.
- **MpegCompressMType() (mpeg.c:1312)** makes sure that the macroblock type is legal. It also handles skipping, zero CBP, and other MPEG-related macroblock stuff.
- **MpegEncodeDFrame() (mpeg.c:1422)** encodes just the DC Intraframe out to the currently open stream. It avoids full DCT calculation.
- **MpegDecodeSequence() (mpeg.c:1527)** decodes the sequence defined in the CImage and CFrame structures; the stream is opened from this routine.
- **MpegDecodeIPBDFrame() (mpeg.c:1733)** is used to decode a generic frame. Note that the DC Intraframe decoding is farmed off to a specialized routine for speed.
- **MpegDecompressMDU() (mpeg.c:1873)** is used to decompress the raw data from the stream. Motion compensation occurs later.
- **MpegDecodeSaveMDU() (mpeg.c:1939)** is used to decode and save the results into a frame store after motion compensation.
- **MpegDecodeDFrame() (mpeg.c:2024)** decodes a single DC Intraframe off of the stream. This function is typically called only from MpegDecodeIPBDFrame().
- **PrintImage() (mpeg.c:2107)** prints the image structure to stdout.
- **PrintFrame() (mpeg.c:2129)** prints the frame structure to stdout.
- **MakeImage() (mpeg.c:2160)** makes an image structure and installs it as the current image.
- **MakeFrame() (mpeg.c:2183)** makes a frame structure and installs it as the current frame structure.
- **MakeFGroup() (mpeg.c:2215)** creates a memory structure for the frame group.
- **LoadFGroup() (mpeg.c:2235)** loads in the memory structure of the current frame group.
- **MakeFstore() (mpeg.c:2269)** makes and installs the frame stores for the motion estimation and compensation.
- **MakeStat() (mpeg.c:2289)** makes the statistics structure to hold all of the current statistics. (CStat).
- **SetCCITT() (mpeg.c:2303)** just sets the width and height parameters for the QCIF, CIF, NTSC-CIF frame sizes.
- **CreateFrameSizes() (mpeg.c:2337)** is used to initialize all of the frame sizes to fit that of the input image sequence.
- **Help() (mpeg.c:2413)** prints out help information about the MPEG program.
- **MakeFileNames() (mpeg.c:2462)** creates the filenames for the component files from the appropriate prefix and suffix commands.

- **VerifyFiles() (mpeg.c:2483)** checks to see if the component files are present and of the correct length.
- **Integer2TimeCode() (mpeg.c:2536)** is used to convert a frame number into a SMPTE 25bit time code as specified by the standard.
- **TimeCode2Integer() (mpeg.c:2611)** is used to convert the 25 bit SMPTE time code into a general frame number based on 0hr 0min 0sec 0pic.

5.2.2 codec.c

```
extern void EncodeAC(int, int *);
extern void CBPEncodeAC(int, int *);
extern void DecodeAC(int, int *);
extern void CBPDecodeAC(int, int *);
extern int DecodeDC(DHUFF *);
extern void EncodeDC(int, EHUFF *);
```

- **EncodeAC() (codec.c:75)** encodes the quantized coefficient matrix input by the first Huffman table. The index is an offset into the matrix.
- **CBPEncodeAC() (codec.c:130)** encodes the AC block matrix when we know there exists a non-zero coefficient in the matrix. Thus the EOB cannot occur as the first element and we save countless bits...
- **DecodeAC() (codec.c:227)** decodes from the stream and puts the elements into the matrix starting from the value of index. The matrix must be given as input and initialized to a value at least as large as 64.
- **CBPDecodeAC() (codec.c:289)** decodes the stream when we assume the input cannot begin with an EOB Huffman code. Thus we use a different Huffman table. The input index is the offset within the matrix and the matrix must already be defined to be greater than 64 elements of int.
- **DecodeDC() (codec.c:390)** decodes a dc element from the stream.
- **EncodeDC() (codec.c:425)** encodes the coefficient input into the output stream.

5.2.3 huffman.c

```
extern void inithuff(void);
extern int Encode(int, EHUFF *);
extern int Decode(DHUFF *);
extern void PrintDhuff(DHUFF *);
extern void PrintEhuff(EHUFF *);
extern void PrintTable(int *);

static DHUFF *MakeDhuff(void);
static EHUFF *MakeEhuff(int);
static void LoadETable(int *, EHUFF *);
static void LoadDTable(int *, DHUFF *);
static int GetNextState(DHUFF *);
static void AddCode(int, int, int, DHUFF *);
```

- **inithuff() (huffman.c:91)** initializes all of the Huffman structures to the appropriate values. It must be called before any of the tables are used.
- **MakeDhuff() (huffman.c:209)** constructs a decoder Huffman table and returns the structure.
- **MakeEhuff() (huffman.c:228)** constructs an encoder huff with a designated table-size. This table-size, n, is used for the lookup of Huffman values, and must represent the largest positive Huffman value.
- **LoadETable() (huffman.c:254)** is used to load an array into an encoder table. The array is grouped in triplets and the first negative value signals the end of the table.
- **LoadDHUFF() (huffman.c:280)** is used to load an array into the DHUFF structure. The array consists of trios of Huffman definitions, the first one the value, the next one the size, and the third one the code.
- **GetNextState() (huffman.c:299)** returns the next free state of the decoder Huffman structure. It exits an error upon overflow.
- **Encode() (huffman.c:319)** encodes a symbol according to a designated encoder Huffman table out to the stream. It returns the number of bits written to the stream and a zero on error.
- **Decode() (huffman.c:352)** returns a symbol read off the stream using the designated Huffman structure.
- **AddCode() (huffman.c:399)** adds a Huffman code to the decoder structure. It is called everytime a new Huffman code is to be defined. This function exits when an invalid code is attempted to be placed in the structure.
- **PrintDHUFF() (huffman.c:495)** prints out the decoder Huffman structure that is passed into it.
- **PrintEhuff() (huffman.c:517)** prints the encoder Huffman structure passed into it.
- **PrintTable() (huffman.c:540)** prints out 256 elements in a nice byte ordered fashion.

5.2.4 io.c

```
extern void MakeFS(int);
extern void SuperSubCompensate(int *, int *, int *, IOBUF *, IOBUF *);
extern void SubCompensate(int *, IOBUF *);
extern void AddCompensate(int *, IOBUF *);
extern void Sub2Compensate(int *, IOBUF *, IOBUF *);
extern void Add2Compensate(int *, IOBUF *, IOBUF *);
extern void MakeMask(int, int, int *, IOBUF *);
extern void ClearFS(void);
extern void InitFS(void);
extern void ReadFS(void);
extern void InstallIob(int);
extern void InstallFSIob(FSTORE *, int);
extern void WriteFS(void);
extern void MoveTo(int, int, int, int);
extern int Bpos(int, int, int, int);
extern void ReadBlock(int *);
extern void WriteBlock(int *);
extern void PrintIob(void);

static void Get4Ptr(int, int *, unsigned char *, unsigned char *, unsigned char *, unsigned char *);
static void Get2Ptr(int, int *, unsigned char *, unsigned char *);
```

- **MakeFS() (io.c:73)** constructs an IO structure and assorted book-keeping instructions for all components of the frame.
- **SuperSubCompensate() (io.c:107)** subtracts off the compensation from three arrays, forward compensation from the first, backward from the second, interpolated from the third. This is done with a corresponding portion of the memory in the forward and backward IO buffers.
- **Sub2Compensate() (io.c:148)** does a subtraction of the prediction from the current matrix with a corresponding portion of the memory in the forward and backward IO buffers.
- **Add2Compensate() (io.c:185)** does an addition of the prediction from the current matrix with a corresponding portion of the memory in the forward and backward IO buffers.
- **SubCompensate() (io.c:225)** does a subtraction of the prediction from the current matrix with a corresponding portion of the memory in the target IO buffer.
- **AddCompensate() (io.c:246)** does an addition of the prediction from the current matrix with a corresponding portion of the memory in the target IO buffer.
- **CopyCFS2FS() (io.c:345)** copies all of the CFrame Iob's to a given frame store.
- **ClearFS() (io.c:360)** clears the entire frame store passed into it.
- **InitFS() (io.c:375)** initializes a frame store that is passed into it. It creates the IO structures and the memory structures.
- **ReadFS() (io.c:407)** loads the memory images from the filenames designated in the CFrame structure.
- **InstallIob() (io.c:436)** installs a particular CFrame Iob as the target Iob.
- **WriteFS() (io.c:455)** writes the frame store out.
- **MoveTo() (io.c:478)** moves the installed Iob to a given location designated by the horizontal and vertical offsets.
- **Bpos() (io.c:493)** returns the designated MDU number inside of the frame of the installed Iob given by the input gob, mdu, horizontal and vertical offset. It returns 0 on error.
- **ReadBlock() (io.c:510)** reads a block from the currently installed Iob into a designated matrix.
- **WriteBlock() (io.c:545)** writes a input matrix as a block into the currently designated IOB structure.
- **PrintIob() (io.c:579)** prints out the current Iob structure to the standard output device.

5.2.5 chendct.c

```
extern void ChenDct(int *, int *);
extern void ChenIDct(int *, int *);
```

- **ChenDCT() (chendct.c:74)** implements the Chen forward dct. Note that there are two input vectors that represent x=input, and y=output, and must be defined (and storage allocated) before this routine is called.
- **ChenIDCT() (chendct.c:192)** implements the Chen inverse dct. Note that there are two input vectors that represent x=input, and y=output, and must be defined (and storage allocated) before this routine is called.

5.2.6 lexer.c

```
extern void initparser(void);
extern void parser(void);
extern void Execute(int);

static int hashpjw(char *);
static LINK * MakeLink(int, char *, int);
static ID * enter(int, char *, int);
static char * getstr(void);
static void PrintProgram(void);
static void MakeProgram(void);
static void CompileProgram(void);
static int mylex(void);
```

- **initparser() (lexer.c:458)** is used to place the Reserved Words into the hash table. It must be called before the parser command is called.
- **hashpjw() (lexer.c:503)** returns a hash value for a string input.
- **MakeLink() (lexer.c:527)** is used to construct a link object. The link is used for the hash table construct.
- **enter() (lexer.c:564)** is used to enter a Reserved Word or ID into the hash table.
- **getstr() (lexer.c:618)** gets a string from the input. It copies the string to temporary storage before it returns the pointer.
- **parser() (lexer.c:700)** handles all of the parsing required for the Program Interpreter. It is basically a `while` statement with a very large case statement for every input. All unmatched values– strings, brackets, etc. are ignored.
- **PrintProgram() (lexer.c:1123)** prints out a program that is loaded as current.
- **MakeProgram() (lexer.c:1234)** makes a program from the input from mylex().
- **CompileProgram() (lexer.c:1375)** assigns values to the labels in a program.
- **mylex() (lexer.c:1454)** reads either from the yylex() routine or from the currently active program depending on what source is active.
- **Execute() (lexer.c:1535)** calls the program interpreter to execute a particular program location.

5.2.7 marker.c

```
extern void ByteAlign(void);
extern void WriteVEHeader(void);
extern void WriteVSHeader(void);
extern int ReadVSHeader(void);
extern void WriteGOPHeader(void);
extern void ReadGOPHeader(void);
extern void WritePictureHeader(void);
extern void ReadPictureHeader(void);
extern void WriteMBSHeader(void);
```

```
extern void ReadMBSHeader(void);
extern void ReadHeaderTrailer(void);
extern int ReadHeaderHeader(void);
extern int ClearToHeader(void);
extern void WriteMBHeader(void);
extern int ReadMBHeader(void);

static void CodeMV(int,int);
static int DecodeMV(int,int);
```

- **ByteAlign() (marker.c:128)** aligns the current stream to a byte-flush boundary. This is used in the standard, with the assumption that input device is byte-buffered.
- **WriteVEHeader() (marker.c:143)** writes out a video sequence end marker.
- **WriteVSHeader() (marker.c:159)** writes out a video sequence start marker. Note that Brate and Bsize are defined automatically by this routine.
- **ReadVSHeader() (marker.c:200)** reads in the body of the video sequence start marker.
- **WriteGOPHeader() (marker.c:258)** write a group of pictures header. Note that the TimeCode variable needs to be defined correctly.
- **ReadGOPHeader() (marker.c:277)** reads the body of the group of pictures marker.
- **WritePictureHeader() (marker.c:292)** writes the header of picture out to the stream. One of these is necessary before every frame is transmitted.
- **ReadPictureHeader() (marker.c:341)** reads the header off of the stream. It assumes that the first PSC has already been read in. (Necessary to tell the difference between a new picture and another GOP.)
- **WriteMBSHeader() (marker.c:380)** writes a macroblock slice header out to the stream.
- **ReadMBSHeader() (marker.c:403)** reads the slice information off of the stream. We assume that the first bits have been read in by ReadHeaderHeader... or some such routine.
- **ReadHeaderTrailer() (marker.c:423)** reads the trailer of the GOP, PSC or MBSC code. It is used to determine whether it is just a new group of frames, new picture, or new slice.
- **ReadHeaderHeader() (marker.c:480)** reads the common structure header series of bytes (and alignment) off of the stream. This is a precursor to the GOP read or the PSC read. It returns -1 on error.
- **ClearToHeader() (marker.c:513)** reads the header header off of the stream. This is a precursor to the GOP read or the PSC read. It returns -1 on error.
- **WriteStuff() (marker.c:544)** writes a MB stuff code.
- **WriteMBHeader() (marker.c:556)** writes the macroblock header information out to the stream.
- **ReadMBHeader() (marker.c:810)** reads the macroblock header information from the stream.

5.2.8 me.c

```
extern void initme(void);
extern void HPFastBME(int, int, MEM *, int, int, MEM *, int, int);
extern void BruteMotionEstimation(MEM *, MEM *);
extern void InterpolativeBME(void);

static int Do4Check(unsigned char *, unsigned char *, unsigned char *, unsigned char *, unsigned char *,
static int Do2Check(unsigned char *, unsigned char *, unsigned char *, int, int);
```

- **initme() (me.c:65)** initializes the motion estimation to the proper number of estimated frames, by FrameInterval.
- **HPFastBME() (me.c:187)** does a fast brute-force motion estimation with two indexes into two memory structures. The motion estimation has a short-circuit abort to speed up calculation.
- **BruteMotionEstimation() (me.c:470)** does a brute-force motion estimation on all aligned 16x16 blocks in two memory structures. It is presented as a compatibility-check routine.
- **InterpolativeBME() (me.c:500)** does the interpolative block motion estimation for an entire frame interval at once. Although motion estimation can be done sequentially with considerable success, the temporal and spatial locality of doing it all at once is probably better.

5.2.9 mem.c

```
extern void CopyMem(MEM *, MEM *);
extern ClearMem(MEM *);
extern SetMem(int, MEM *);
extern MEM *MakeMem(int, int);
extern void FreeMem(MEM *);
extern MEM *LoadMem(char *, int, int, MEM *);
extern MEM *LoadPartialMem(char *, int, int, int, int, MEM *);
extern MEM *SaveMem(char *, MEM *);
extern MEM *SavePartialMem(char *, int, int, MEM *);
```

- **CopyMem() (mem.c:35)** copies the entire contents of m2 to m1.
- **ClearMem() (mem.c:48)** clears a memory structure by setting it to all zeroes.
- **SetMem() (mem.c:61)** clears a memory structure by setting it to all a value.
- **MakeMem() (mem.c:74)** creates a memory structure out of a given width and height.
- **FreeMem() (mem.c:104)** frees a memory structure.
- **LoadMem() (mem.c:119)** loads an Mem with a designated width, height, and filename into a designated memory structure. If the memory structure is NULL, one is created for it.

- **LoadPartialMem() (mem.c:156)** loads an Mem with a designated width, height, and filename into a designated memory structure. The file is of pwidth and pheight, and if different than the width and height of the memory structure, the structure is padded with 128's. If the memory structure is NULL, one is created for it.
- **SaveMem() (mem.c:207)** saves the designated memory structure to the appropriate filename.
- **SavePartialMem() (mem.c:230)** saves the designated memory structure to the appropriate filename.

5.2.10 stat.c

```
extern void Statistics(FSTORE *, FSTORE *);
static void StatisticsMem(MEM *, MEM *, STAT *);
```

- **Statistics() (stat.c:30)** prints to `stdout` all the accumulated statistics on the memory structures (CFS and Iob).
- **StatisticsMem() (stat.c:50)** calculates the statistics beween a reference memory structure and another memory structure, storing it in a statistics structure.

5.2.11 stream.c

```
extern void readalign(void);
extern void mropen(char *);
extern void mrclose(void);
extern void mwopen(char *);
extern void mwclose(void);
extern void zeroflush(void);
extern int mgetb(void);
extern void mputv(int, int);
extern int mgetv(int);
extern long mwtell(void);
extern long mrtell(void);
extern void mwseek(long);
extern void mrseek(long);
extern int seof(void);
```

- **readalign() (stream.c:66)** aligns the read stream to the next byte boundary.
- **mropen() (stream.c:80)** opens up the stream for reading on a bit level.
- **mrclose() (stream.c:99)** closes a read bit-stream.
- **mwopen() (stream.c:112)** opens a bit stream for writing.
- **mwclose() (stream.c:133)** closes the write bit-stream. It flushes the remaining byte with ones, consistent with -1 returned on EOF.
- **zeroflush() (stream.c:152)** flushes out the rest of the byte with 0's.

- **mputb() (stream.c:169)** puts a bit to the write stream.
- **mgetb() (stream.c:183)** returns a bit from the read stream.
- **mputv() (stream.c:202)** puts a n bits to the stream from byte b.
- **mgetv() (stream.c:219)** returns n bits read off of the read stream.
- **mwtell() (stream.c:239)** returns the position in bits of the write stream.
- **mrtell() (stream.c:252)** returns the position in read bits of the read stream.
- **mwseek() (stream.c:265)** seeks to a specific bit position on the write stream.
- **mrseek() (stream.c:294)** seeks to a specific bit position on the read stream.
- **seof() (stream.c:310)** returns a -1 if at the end of stream. It mimics the `feof` command.

5.2.12 transform.c

```
extern void ReferenceDct(int *, int *);
extern void ReferenceIDct(int *, int *);
extern void TransposeMatrix(int *, int *);
extern void MPEGIntraQuantize(int *, int *, int);
extern void MPEGIntraIQuantize(int *, int *, int);
extern void MPEGNonIntraQuantize(int *, int *, int);
extern void MPEGNonIntraIQuantize(int *, int *, int);
extern void BoundIntegerMatrix(int *);
extern void BoundQuantizeMatrix(int *);
extern void BoundIQuantizeMatrix(int *);
extern void ZigzagMatrix(int *, int *);
extern void IZigzagMatrix(int *, int *);
extern void PrintMatrix(int *);
extern void ClearMatrix(int *);

static void DoubleReferenceDct1D(double *,double *);
static void DoubleReferenceIDct1D(double *, double *);
static void DoubleTransposeMatrix(double *, double *);
```

- **ReferenceDct() (transform.c:69)** does a reference DCT on the input (matrix) and output (new matrix).
- **DoubleReferenceDCT1D() (transform.c:109)** does a 8 point dct on an array of double input and places the result in a double output.
- **ReferenceIDct() (transform.c:130)** is used to perform a reference 8x8 inverse dct. It is a balanced IDCT. It takes the input (matrix) and puts it into the output (newmatrix).
- **DoubleReferenceIDct1D() (transform.c:169)** does an 8 point inverse dct on ivect and puts the output in ovect.
- **TransposeMatrix (transform.c:190)** transposes an input matrix and puts the output in newmatrix.
- **DoubleTransposeMatrix (transform.c:208)** transposes a double input matrix and puts the double output in newmatrix.

- **MPEGIntraQuantize() (transform.c:227)** quantizes the input matrix with a fixed DC quantize step and an AC quantize step; along with a variable quantization factor.
- **MPEGIntraIQuantize() (transform.c:262)** inverse quantizes the input matrix with a fixed DC quantize step and an AC quantize step; along with a variable quantization factor.
- **MPEGNonIntraQuantize() (transform.c:288)** quantizes the input matrix with a quantization matrix and a quantization factor.
- **MPEGNonIntraIQuantize() (transform.c:330)** inverse quantizes the input matrix with a quantization matrix and a quantization factor.
- **BoundIntegerMatrix (transform.c:358)** bounds the output matrix so that no pixel has a value greater than 255 or less than 0.
- **BoundQuantizeMatrix() (transform.c:377)** bounds the coefficients of a quantized matrix.
- **BoundIQuantizeMatrix() (transform.c:398)** bounds an inverse quantized matrix.
- **IZigzagMatrix() (transform.c:419)** performs an inverse zig-zag translation on the input imatrix and places the output in omatrix.
- **ZigzagMatrix() (transform.c:437)** performs a zig-zag translation on the input imatrix and puts the output in omatrix.
- **PrintMatrix() (transform.c:455)** prints an 8x8 matrix in row/column form.
- **ClearMatrix() (transform.c:478)** sets all the elements of a matrix to be zero.

Bibliography

[1] U.S. Senate, *Committee on Interstate and Foreign Commerce, Advisory Committee on Color Television.* "The Present Status of Color Television – *Report of the Advisory Committee on Color Television to the Committee on Interstate and Foreign Commerce*," United States Senate, 81st Congress, 2nd Session, Document No., 197, Washington D.C., 1950.

[2] Boris Townsend, *PAL Colour Television*, Cambridge at the University Press, 1970.

[3] Arun N. Netravali and Barry G. Haskell, *Digital Pictures*, Plenum Press, 1988.

[4] W. A. Chen, C. Harrison, and S. C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Trans. Commun.*, vol. **COM-25**, No. 9, pp. 1004-1011, September, 1977. (See also COM-31, pp. 121-123.)

[5] Byeong G. Lee, "A New Algorithm to Compute the Discrete Cosine Transform," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. **ASSP-32**, No. 6, pp. 1243-1245, December 1984.

[6] Ephraim Feig, "A fast scaled-DCT algorithm," *SPIE Image Processing Algorithms and Techniques*, vol. **1244**, pp. 2-13, 1990.

[7] David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proc. IRE*, pp. 1098-1101, September 1953.

[8] Didier Legall, "MPEG - A Video Compression Standard for Multimedia Applications," *Communications of the ACM*, vol. **34**, No. 4, pp. 47-58, April 1991.

[9] Atul Puri and R. Aravind, "Motion-Compensated Video Coding," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. **1**, No. 4, pp. 351-361, December 1991.

[10] H.261, RM8.

[11] MPEG, "MPEG Video Simulation Model Three (SM3)," Draft #1, MPEG90.

[12] MPEG, "Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbits/s. Part 2, Video." MPEG 91/185 (Santa Clara).

[13] Eric Hamilton, "JPEG File Interchange Format, Version 1.02," September 1, 1992.

Index