Institutionen för systemteknik

Department of Electrical Engineering

Examensarbete

Institutionen för systemteknik
Department of Electrical Engineering

Separating the MPEG-2-Bitstream for Lossless Compression and Transmission

Francisco A. Rivas Portillo

LiTH-ISY-EX-3078

2000-06-09

TEKNISKA HÖGSKOLAN
LINKÖPINGS UNIVERSITET

Department of Electrical Engineering
Linköping University
S-581 83 Linköping, Sweden

Linköpings tekniska högskola
Institutionen för systemteknik
581 83 Linköping
Separating the MPEG-2 Bitstream for Lossless Compression and Transmission

Master Thesis written at Image Coding Group Linköping University by Francisco A. Rivas Portillo

Reg nr: LiTH-ISY-EX-3078

Supervisor: Ms. Astrid Lundmark, Image Coding Group

Examiner: Dr. Robert Forchheimer, Image Coding Group

Linköping, June 2000
The video coding standard of the Moving Picture Experts Group (MPEG-2) achieves data reduction by motion compensated prediction, discrete cosine transform (DCT), and variable length coding (VLC) of quantized DCT coefficients, motion data and signalling data.

Part of the generated MPEG-2 bitstream is data information (DI), that is, quantized VLC DCT coefficients and Motion Vectors (MV), and part is signalling information (SI), needed by the MPEG-2 decoder in order to display the bitstream. The ratio between the SI and the whole size of the bitstream can be very high. It is even possible that the SI is larger than the DI, degrading the efficiency of the compression process performed by the MPEG-2 coder.

One of the purposes of this Master Thesis is to implement a system to losslessly compress the SI contained in an MPEG-2 bitstream. The lossless compression is done by Huffman coding and Lempel Ziv Welch coding.

Another purpose will be to study the transmission of the losslessly compressed MPEG-2 bitstream through a noisy channel. This involves channel coding, which is performed by a convolutional encoder. The channel decoding process is done by a Viterbi decoder.
Acknowledgements

I would like to express my sincere gratitude to Ms. Astrid Lundmark, my supervisor, for her valuable help and guidance in the development of this work.

I am very grateful to Dr. Robert Forchheimer for allowing me to do this Master Thesis in the Image Coding Group.

I would also like to thank all the people that directly or indirectly have helped me in the fulfilment of this project.
# Table of Contents:

1. Introduction................................................................................. 1
   1.1 Motivation................................................................................ 1
   1.2 Objectives................................................................................ 1

2. MPEG-2 Standard......................................................................... 3
   2.1 Introduction.............................................................................. 3
   2.2 Video Fundamentals............................................................... 4
   2.3 Bit Rate Reduction Principles............................................... 5
   2.4 MPEG-2 Details...................................................................... 9
   2.5 Conclusions about the MPEG-2 Standard............................. 18

3. General Overview of the System................................................. 19
   3.1 Coder Side............................................................................. 19
   3.2 Decoder Side.......................................................................... 20

4. Splitting Program........................................................................ 23
   4.1 Experimental Set-up.............................................................. 23
   4.2 Bitstream Splitting............................................................... 25
   4.3 Reasons for Auxiliary Information....................................... 29
   4.4 SI, DI and PSNR Plots........................................................... 32

5. Lossless Compression of Signalling Information....................... 49
   5.1 Introduction........................................................................... 49
   5.2 Huffman Coding vs. Lempel Ziv Welch Coding................... 49
   5.3 Compressed MPEG-2 Bitstream Format.............................. 57

6. Convolutional Coding and Viterbi Decoding............................... 59
   6.1 Introduction........................................................................... 59
   6.2 Convolutional Coding.......................................................... 60
   6.3 Viterbi Decoding................................................................. 65
   6.4 Lossless Compressed Bitstream Channel Coding............... 72
   6.5 Recovering the Compressed MPEG-2 Bitstream............... 74

7. Reconstructing Program............................................................ 77
   7.1 Error-free Reconstructing Files............................................ 77
   7.2 Non-error-free Reconstructing Files.................................... 79

8. Conclusions.................................................................................. 83
9. Suggestions and Future Improvements .......................... 85

References ........................................................................ 87

Appendix A: Tables ......................................................... 89

  A.1 MPEG-2 Bitstream Structure ..................................... 89
  A.2 Lossless Compression Ratios ...................................... 92

Appendix B: Lossless Compression and Expansion Theory .... 97

  B.1 Huffman Coding and Decoding ............................... 97
  B.2 Lempel Ziv Welch Coding and Decoding ................. 101
1. Introduction

This Master Thesis (Examensarbete) has been realized at the Image Coding Group (ICG), in the Electrical Engineering Department (ISY), at Linköping University, Sweden. The work was done under the supervision of Ms. Astrid Lundmark.

The C source codes of the programs were implemented on a SUN Spark station, using emacs as text editor and gcc as C compiler.

The written report was done on a PC, using Microsoft Word 98 as text editor.

1.1 Motivation

The video coding standard of the Moving Picture Experts Group (MPEG-2) achieves data reduction by motion compensated prediction, discrete cosine transform (DCT) and variable length coding of quantized DCT (VLC DCT) coefficients, motion and signalling data [4]. See Chapter 2 for details.

The degree of compression achieved by MPEG-2 is very high. It can reduce an original video sequence to around 5% of its original size, depending on video content and acceptable distortion. Part of the generated MPEG-2 bitstream is data information, that is, quantized VLC DCT coefficients and Motion Vectors (MV), and part is signalling information (SI), needed by the MPEG-2 decoder in order to display the bitstream.

Depending on the bit rate that we are using to generate the bitstream and also on the nature of the input video sequence, the ratio between the SI and the whole size of the bitstream can be very high. It is even possible that the SI is larger than the DI, degrading the efficiency of the compression process performed by the MPEG-2 coder.

1.2 Objectives

One of the purposes of this Master Thesis is to implement a system to losslessly compress the SI contained in an MPEG-2 bitstream. This task requires a program to split the signalling and the data information from an input bitstream. We will see in Chapter 4 the amount of SI (in percentage) for different coding bit rates (0.1-1.0 Mbit/s), for video sequences with a fixed number of frames (30 frames). The other kind of experiment is to fix the bit rate (to 0.5 Mbit/s) and increase the number of frames.

The lossless compression is done by two already existing programs in the UNIX environment. They perform Huffman (pack) and Lempel Ziv Welch (gzip) coding. The reciprocal programs are called unpack and gunzip.
The operations described above are enough for storage purposes. But we are going to consider the possibility of transmitting the compressed splitted bitstream through a channel that corrupts that information. Chapter 6 it is shown how the channel coding is performed by a convolutional coder, and the channel decoding process is done by a Viterbi decoder. It is important to note that the channel coding description is done in a qualitative form, not quantitative.

Since we want to recover the original bitstream, a reconstructing (merging) program has been implemented. This is described in Chapter 7, and we will see that the key to success in this task is the auxiliary information generated by the splitting program, together with the MPEG-2 syntax. We will consider these two cases: error-free reconstructing files and non-error-free reconstructing files.

Chapter 8 deals with the most relevant conclusions and results found along the previous Chapters. In Chapter 9, several future improvements and suggestions are presented. The most interesting suggestion is related to the MPEG-2 standard itself, and the possible modifications we can introduce in the bitstream concept, in order to increase the degree of compression that the MPEG-2 standard can achieve.

The final Chapters of this thesis are appendices containing the tables from which the plots in Chapters 4 and 5 were made. There are also descriptions of the Huffman and Lempel Ziv Welch compression schemes.
2. The MPEG-2 Standard

2.1 Introduction

Recent progress in digital technology has made the widespread use of compressed digital video signals practical [1]. Standardisation has been very important in the development of common compression methods to be used in the new services and products that are now possible.

MPEG (Moving Picture Experts Group) was started in 1988 as a working group within ISO/IEC with the aim of defining standards for digital compression of audio-visual signals. MPEG’s first project, MPEG-1, was published in 1993 as ISO/IEC 11172. It is a three-part standard defining audio and video compression coding methods and a multiplexing system for interleaving audio and video data so that they can be played back together. MPEG-1 principally supports video coding up to about 1.5 Mbit/s giving quality similar to VHS and stereo audio at 192 kbit/s. It is used in the CD-i and Video-CD systems for storing video and audio on CD-ROM.

During 1990, MPEG recognized the need for a second, related standard for coding broadcast formats at higher data rates. The MPEG-2 standard is capable of coding standard-definition television at bit rates from about 3 to 15 Mbit/s and high-definition television at 15-30 Mbit/s. MPEG-2 extends the stereo audio capabilities of MPEG-1 to multichannel surround sound coding. MPEG-2 decoders also decode MPEG-1 bitstreams.

Drafts of the audio, video and systems specifications were completed in November 1993 and the ISO/IEC approval process was completed in November 1994. The final text was published in 1995.

MPEG-2 aims to be a generic video coding system supporting a diverse range of applications. Different algorithmic ‘tools’, developed for many applications, have been integrated into the full standard. To implement all the features of the standard in all decoders is unnecessarily complex, so a small number of subsets of the full standard, known as profiles and levels, have been defined. A profile is a subset of algorithmic tools and a level identifies a set of constraints on parameter values (such as picture size and bit rate). A decoder which supports a particular profile and level is only required to support the corresponding subset of the full standard and set of parameter constraints.
2.2 Video Fundamentals

Television services in Europe currently broadcast video at a frame rate of 25 Hz. Each frame consists of two interlaced fields, giving a field rate of 50 Hz. The first field of each frame contains only the odd-numbered lines of the frame (numbering the top frame line as line 1). The second field contains only the even-numbered lines of the frame and is sampled in the video camera 20 ms after the first field. It is important to note that one interlaced frame contains fields from two time instances. American television is similarly interlaced but with a frame rate of just under 30 Hz.

In video systems other than television, non-interlaced video is commonplace (for example, most computers output non-interlaced video). For non-interlaced video, all the lines of a frame are sampled at the same time instance. Non-interlaced video is also termed 'progressively scanned' or 'sequentially scanned' video.

The red, green and blue (RGB) signals coming from a colour television camera can be equivalently expressed as luminance (Y) and chrominance (UV) components. The chrominance bandwidth may be reduced relative to the luminance without significantly affecting the picture quality. For standard-definition video, CCIR recommendation 601 defines how the component (YUV) video signals can be sampled and digitalized to form discrete pixels. The terms 4:4:4, 4:2:2 and 4:2:0 are often used to describe the sample structure of the digital picture. 4:4:4 indicates that the chrominance is sampled using the same bit rate relative to the luminance. 4:2:2 indicates that the chrominance is horizontally subsampled by a factor of two relative to the luminance; 4:2:0 indicates that the chrominance is horizontally and vertically subsampled by a factor of two relative to the luminance.

The active region of a digital television frame, sampled according to CCIR recommendation 601, is 720 pixels by 576 lines for a frame rate of 25 Hz. Using 8 bits for each Y, U and V pixels, the uncompressed bit rates for 4:4:4, 4:2:2 and 4:2:0 signals are therefore:

- 4:4:4: $720 \times 576 \times 25 \times 8 + 720 \times 576 \times 25 \times (8 + 8) \approx 248$ Mbit/s
- 4:2:2: $720 \times 576 \times 25 \times 8 + 360 \times 576 \times 25 \times (8 + 8) \approx 166$ Mbit/s
- 4:2:0: $720 \times 576 \times 25 \times 8 + 360 \times 288 \times 25 \times (8 + 8) \approx 124$ Mbit/s

MPEG-2 is capable of compressing the bit rate of standard-definition 4:2:0 video down to about 3-15 Mbit/s. For digital terrestrial television broadcasting of standard-definition video, a bit rate of around 6 Mbit/s is thought to be a good compromise between picture quality and transmission bandwidth efficiency.
2. The MPEG-2 Standard

2.3 Bit Rate Reduction Principles

A bit rate reduction system operates by removing redundant information from the signal at the coder prior to transmission, and re-inserting it at the decoder. A coder and decoder pair are referred to as ‘codec’. In video signals, two distinct kinds of redundancy can be identified.

- **Spatial and temporal redundancy:** Pixel values are not independent, but are correlated with their neighbours both within the same frame and across frames. So, to some extent, the value of a pixel is predictable given the values of neighbouring pixels.

- **Psychovisual redundancy:** The human eye has a limited response to fine spatial detail and is less sensitive to detail near object edges or around shot-changes. Consequently, controlled impairments introduced into the decoded picture by the bit rate reduction process should not be visible to a human observer.

Two key techniques employed in an MPEG-2 codec are intraframe Discrete Cosine Transform (DCT) coding and Motion Compesated interframe prediction. These techniques have been successfully applied to video bit rate reduction prior to MPEG, notably for video conference systems at bit rates below 2 Mbit/s.

*Intraframe DCT coding*

**DCT:** A two-dimensional DCT is performed on small blocks (8x8 pixels) of each component of the picture to produce blocks of DCT coefficients. The DCT for an 8x8 block is defined as:

\[
E(u,v) = \frac{4C(u) \cdot C(v)}{16} \sum_{j=0}^{7} \sum_{k=0}^{7} e^{(j,k)} \cdot \cos \left( \frac{(2j + 1)u\pi}{16} \right) \cdot \cos \left( \frac{(2k + 1)v\pi}{16} \right)
\]

where \( j, k \) and \( u, v \) represent indices in the horizontal and vertical directions for the pixel and coefficient blocks, respectively; \( u, v, j, k = 0, 1, \ldots, 7 \); and

\[
C(w) = \frac{1}{\sqrt{2}} \quad \text{for} \ w = 0;
\]

\[
C(w) = 1 \quad \text{for} \ w = 1, 2, \ldots, 7
\]

The magnitude of each DCT coefficient indicates the contribution of a particular combination of horizontal and vertical spatial frequencies to the original picture block. The coefficient corresponding to zero horizontal and vertical frequency is called the DC coefficient. It is important to differentiate \( u \) and \( v \) from U and V, the chrominance components.

The DCT does not directly reduce the number of bits required to represent the block. In fact for an 8x8 block of 8 bit pixels, the DCT produces an 8x8 block of 11 bit coefficients (the
range of coefficient values is larger than the range of pixel values). The reduction in the number of bits follows from the observation that, for typical blocks from natural images, the distribution of coefficients is non-uniform. The transform tends to concentrate the energy into the low-frequency coefficients, and many of the other coefficients are near zero. The bit rate reduction is achieved by not transmitting the near-zero coefficients and by quantising and coding the remaining coefficients as described below. The non-uniform coefficient distribution is a result of the spatial redundancy present in the original image block.

**Quantization:** The function of the coder is to transmit the DCT block to the decoder, in a bit rate efficient manner, so that it can perform the inverse transform to reconstruct the image. It has been observed that the numerical precision of the DCT coefficients may be reduced, while still maintaining good image quality at the decoder. The coder and the decoder have to use the same coefficients, though, otherwise errors will accumulate at the decoder. Quantization is used to reduce the number of possible values to be transmitted, reducing the required number of bits.

The degree of quantization applied to each coefficient is weighted according to the visibility of the resulting quantization noise to a human observer. In practice, this results in the high-frequency coefficients being more coarsely quantized than the low-frequency coefficients. Note that the quantization noise introduced by the coder is not reversible in the decoder, making this coding and decoding process 'lossy'. See Figures 2.1 and 2.2.

![Human Vision Sensitivity vs. Spatial Frequency](image)

**Figure 2.1** Human Vision Sensitivity vs. Spatial Frequency, showing that the human perception of noise in pictures is not uniform but is a function of the spatial frequency. More noise can be tolerated at high spatial frequencies.

**Coding:** The serialization and coding of the quantized DCT coefficients exploits the likely clustering of energy into the low-frequency coefficients, and the frequent occurrence of zero-value coefficients. The block is scanned in a diagonal zigzag pattern, starting at the DC coefficient, to produce a list of quantized coefficient values, ordered according to the scan pattern. See Figure 2.3.
2. The MPEG-2 Standard

The list of values produced by scanning is entropy coded using a variable-length code (VLC). Each VLC codeword denotes a run of zeros followed by a non-zero coefficient of a particular level. VLC coding recognizes that short runs of zeros are more likely than long ones, and small coefficients are more likely than large ones. The VLC allocates codewords which have different lengths depending upon the probability with which they are expected to occur. To enable the decoder to distinguish where one code ends and the next begins, the VLC has the property that no complete code is a prefix of any other.

![Diagram](image1.png)

**Figure 2.2** The coefficients from the DCT are divided by constants that are a function of two-dimensional frequency, in the weighting process.

![Diagram](image2.png)

**Figure 2.3** Scan scheme of 8x8 quantized DCT coefficients.
To illustrate the variable-length coding process, consider the following example list of values produced by scanning the quantized coefficients from a transformed block:

12, 6, 6, 0, 4, 3, 0, 0, 0, ..., 0

The first step is to group the values into runs of (zero or more) zeros followed by non-zero value. Additionally, the final run of zeros is replaced with an end of block (EOB) marker. Using parentheses to show the groups, this gives:

(12), (6), (6), (0,4), (3) EOB

The second step is to generate the variable length code-words corresponding to each group (a run of zeros followed by a non-zero value) and the EOB marker. Table 1 shows an extract of the DCT coefficient VLC table common to both MPEG-1 and MPEG-2. MPEG-2 has an additional ‘intra’ VLC optimized for coding intra blocks (see section 2.4). Using the variable length code from Table 2-1 and adding spaces and commas for readability, the final coded representation of the example block is:

0000 0000 1101 00, 0010 0001 0, 0010 0001 0, 0000 0011 000, 0010 10, 10

<table>
<thead>
<tr>
<th>Length of run of zeros</th>
<th>Value of non-zero coefficient</th>
<th>Variable-length codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>0000 0000 1101 00</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>0010 0001 0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0000 0011 000</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0010 10</td>
</tr>
<tr>
<td>EOB</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2-1: Extract from the MPEG-2 DCT coefficient VLC table.

Motion-compensated interframe prediction

This technique exploits temporal redundancy by attempting to predict the frame to be coded from a previous ‘reference’ frame. The prediction cannot be based on a source picture because the prediction has to be repeatable in the decoder, where the source pictures are not available (the decoded pictures are not identical to the source pictures because the bit rate reduction process introduces small distortions into the decoded picture). Consequently, the coder contains a local decoder which reconstructs pictures exactly as they would be in the decoder, from which predictions can be formed.

The simplest interframe prediction of the block being coded is that which takes the co-sited (i.e. the same spatial position) block from the reference picture. Naturally this makes a good prediction for stationary regions of the image, but is poor in moving areas. A more sophisticated method, known as motion-compensated interframe prediction, is to offset any translational motion which has occurred between the block being coded and the reference frame, and to use a shifted block from the reference frame as a prediction, see Figure 2.4.
2. The MPEG-2 Standard

Motion-compensated interframe prediction technique, showing how the motion vector is used together with the DCT VLC coefficients to code the actual block.

One method of determining the motion that has occurred between the block being coded and the reference frame is a 'block-matching' search in which a large number of trial offsets are tested by the coder using the luminance component of the picture. The 'best' offset is selected on the basis of the minimum error between the block being coded and the prediction.

The bit rate overhead of using motion-compensated prediction is the need to convey the motion vectors required to predict each block to the decoder. For example, using MPEG-2 to compress standard-definition video to 6 Mbit/s, the motion vector overhead could account for about 2 Mbit/s during a picture making heavy use of motion-compensated prediction.

2.4 MPEG-2 Details

Codec structure

In an MPEG-2 system, the DCT and motion-compensated prediction are combined, as shown in Figure 2.5. The code subtracts the motion-compensated prediction from the source picture to form a 'prediction error' picture. The prediction error is transformed with the DCT, the coefficients are quantized and these quantized values coded using VLC. The coded luminance and chrominance prediction error is combined with 'side information' required by the decoder, such as motion vectors and synchronising information, and formed into a bitstream for transmission. Figure 2.6 shows an outline of the MPEG-2 video bitstream structure.
In the decoder, the quantized DCT coefficients are reconstructed and inverse transformed to produce the prediction error. This is added to the motion-compensated prediction generated from previously decoded pictures to produce the decoded output.

![Diagram of motion-compensated DCT coder and decoder](image)

**Figure 2.5** Motion-compensated DCT coder and decoder

**Key words:**
- DCT: Discrete Cosine Transform
- IDCT: Inverse Discrete Cosine Transform
- Q: Quantization
- IQ: Inverse Quantization
- VLC: Variable Length Coder
- VLD: Variable Length Decoder
- MCP: Motion Compensated Prediction

In an MPEG-2 codec, the motion-compensated predictor shown in Figure 2.5 supports many methods for generating a prediction. For example, the block may be ‘forward predicted’ from a previous picture, ’backward predicted’ from a future picture, or ’bidirectionally predicted’ by averaging a forward and backward prediction. The method used to predict the block may change from one block to the next. Additionally, the two fields within a block may be predicted separately with their own motion vector, or together using a common motion vector. Another option is to make a zero-value prediction, such that the source image block rather than the prediction error block is DCT coded. For each block to be coded, the coder chooses between these prediction modes, trying to maximize the decoded picture quality within the constraints of the bit rate. The choice of prediction mode is transmitted to the decoder, with the prediction error, so that it may regenerate the correct prediction.
2. The MPEG-2 Standard

A sequence consists of all the pictures that follow a sequence header until a sequence_end_code occurs. Encoding and displaying parameters are transmitted with the sequence header. The sequence header can be repeated in order to allow random access, but all the data elements of the repeated sequence header, except those concerning quantisation matrices, must have the same values as in the first sequence header. The repeated sequence header must precede in the bitstream either an I-picture or a P-picture (described below together with B-pictures). In the case that random access is performed to a P-picture, it is possible that the decoded pictures may not be correct.

*Picture types*

In MPEG-2, three ‘picture types’ are defined. The picture type defines which prediction modes may be used to code each block.

‘Intra’ pictures (I-pictures) are coded without reference to other pictures. Moderate compression is achieved by reducing spatial redundancy, but not temporal redundancy. They can be used periodically to provide access points in the bitstream where decoding can begin.

‘Predictive’ pictures (P-pictures) can use the previous I- or P-picture for motion compensation and may be used as a reference for further prediction. Each block in a P-picture can be either predicted or intra-coded. By reducing spatial and temporal redundancy, P-pictures offer increased compression compared to I-pictures.
'Bidirectionally-predictive’ pictures (B-pictures) can use the previous and next I- or P-pictures for motion compensation, and offer the highest degree of compression. Each block in a B-picture can be forward, backward or bidirectionally predicted or intra-coded. To enable backward prediction from a future frame, the coder reorders the pictures from natural 'display' order to 'bitstream' order so that the B-picture is transmitted after the previous and next pictures it references. This introduces a reordering delay dependent on the number of consecutive B-pictures. See Figure 2.7.

![Figure 2.7 Typical MPEG-2 picture dependency.](image)

The different picture types typically occur in a repeating sequence, termed a 'group of pictures' or GOP. A typical GOP in display order is:

\[ B_1 B_2 I_3 B_4 B_5 P_6 B_7 B_8 P_9 B_{10} B_{11} P_{12} \]

The corresponding bitstream order is:

\[ I_3 B_1 B_2 P_6 B_4 B_5 P_9 B_7 B_8 P_{12} B_{10} B_{11} \]

A regular GOP structure can be described with two parameters: N, which is the number of pictures in the GOP, and M, which is the spacing of P-pictures. The GOP given here is described as \( N = 12 \) and \( M = 3 \). MPEG-2 does not insist on a regular GOP structure. For example, a P-picture following a shot-change may be poorly predicted since the reference picture for prediction is completely different from the picture being predicted. Thus, it may be beneficial to code it as an I-picture instead.

The GOP layer allows random access because the first picture after the GOP header is an I-picture, which means that it does not need any reference to any other picture. The GOP layer is optional, i.e. it is not mandatory to put any GOP header in the bitstream. In the header there is also the timecode of the first picture of the GOP to be displayed.

The decoding process, as the GOP header is immediately followed by an I-picture, can begin at that point of the bitstream. Anyway it is possible that some B-pictures, following such I-pictures in the bitstream, have references coming from the previous GOP and can not
2. The MPEG-2 Standard

be correctly decoded. In this case the GOP is called an Open GOP because some references from the previous GOP exist; if a random access to such a GOP is performed, some B-pictures should not be displayed. A GOP is called a Closed GOP when either there are no B-pictures immediately following the first I-picture or such B_pictures do not have any references coming from the previous GOP (in this case a GOP header flag must be set).

For a given decoded picture quality, coding using each picture type produces a different number of bits. In a typical example sequence, a coded I-picture was three times larger than a coded P-picture, which was itself 50% larger than a coded B-picture.

The next layer, after the picture layer, in the hierarchy is the slice. A slice is a portion of image of 16 lines x (n x16) pixels. Each slice is coded independently from the other slices of the picture. Therefore the slice layer allows error confinement because, when errors in the bitstream are detected, the decoder can try to continue the decoding process looking for the next slice header. The video decoding process at the slice layer has these steps: decode slice_vertical_position, decode quantiser_scale_code and decode all the macroblocks that compose the slice.

A macroblock is a portion of image that consists of 16x16 pixels (four blocks). At the macroblock layer motion compensation and prediction are performed and it is possible to change the quantisation step. It must be noticed that, if the picture is an interlaced frame picture, the odd lines of the macroblock belong to the first field and the even lines to the second field. The video decoding process at the macroblock layer has these steps:

Decode the macroblock mode and the possible quantiser_scale_code.

-) If it is an Intra macroblock:

   Decode the blocks which the macroblock consists of.

-) If it is a Non-Intra macroblock:

   Decode the prediction mode and the motion vectors.
   Produce the suitable prediction for the macroblock.
   Decode the blocks which the macroblock consists of, obtaining the prediction errors values.
   Add the prediction errors values to the prediction.

Buffer control

By removing much of the redundancy from the source images, the coder outputs a variable bit rate. The bit rate depends on the complexity and predictability of the source picture and the effectiveness of the motion-compensated prediction.
For many applications, the bitstream must be carried in a fixed bit-rate channel. In these cases, a buffer store is placed between the coder and the channel. The buffer is filled at a variable rate by the coder, and emptied at a constant rate by the channel. To prevent the buffer from under- or overflowing, a feedback mechanism acts to adjust the average coded bit rate as a function of the buffer fullness. For example, the average coded bit rate may be lowered by increasing the degree of quantization applied to the DCT coefficients. This reduces the number of bits generated by the variable-length coding, but increases distortion in the decoded image. The decoder must also have a buffer between the channel and the variable-rate input to the decoding process. The size of the buffers in the coder and decoder must be the same.

MPEG-2 defines the maximum decoder (and hence coder) buffer size, although the coder may choose to use only part of this. The delay through the coder and decoder buffer is equal to the buffer size divided by the channel bit rate. For example, an MPEG-2 coder operating at 6 Mbit/s with a buffer size of 1.8 Mbit would have a total delay through the coder and decoder buffers of around 300 ms. Reducing the buffer size will reduce the delay, but may affect picture quality if the buffer becomes too small to accommodate the variation in bit rate from the coder VLC.

Profiles and levels

MPEG-2 video is an extension of MPEG-1 video. MPEG-1 was targeted at coding progressively scanned video at bit rates up to about 1.5 Mbit/s. MPEG-2 provides extra algorithmic ‘tools’ for efficiently coding interlaced video and supports a wide range of bit rates. MPEG-2 also provides tools for ‘scalable’ coding where useful video can be reconstructed from pieces of the total bitstream. The total bitstream may be structured in layers, starting with a base layer (that can be decoded by itself) and adding refinement layers to reduce quantization distortion or improve resolution.

A small number of subsets of the complete MPEG-2 tool-kit have been defined, known as profiles and levels. A profile is a subset of algorithmic tools and a level identifies a set of constraints on parameter values (such as picture size or bit rate). The profiles and levels defined to date fit together such that a higher profile or level is a superset of a lower one. A decoder which supports a particular profile and level is only required to support the corresponding subset of algorithmic tools and set of parameter constraints.
Two non-scalable profiles are defined by the MPEG-2 specification.

The *simple profile* uses no B-frames, and hence no forward or interpolated prediction. Consequently, no picture reordering is required (picture reordering would add about 120 ms to the coding delay). With a small coder buffer, this profile is suitable for low-delay applications such as video conferencing where the overall delay is around 100 ms. Coding is performed on a 4:2:0 video signal.

The *main profile* adds support for B-pictures and is the most widely used profile. Using B-pictures increases the picture quality, but adds about 120 ms to the coding delay to allow for the picture reordering. Main profile decoders will also decode MPEG-1 video.

**Details of scalable profiles**

The *SNR profile* adds support for enhancement layers of DCT coefficient refinement, using the ‘signal to noise (SNR) scalability’ tool. Figure 2.8 shows an example SNR-coder and decoder.

![SNR-scalable video coder and decoder.](image)

**Key words:**

DCT: Discrete Cosine Transform  
IDCT: Inverse Discrete Cosine Transform  
Q: Quantization  
IQ: Inverse Quantization  
VLC: Variable Length Coder  
VLD: Variable Length Decoder  
MCP: Motion Compensated Prediction

The codec operates in a similar manner to the non-scalable codec shown in Figure 2.5, with the addition of an extra quantization stage. The coder quantizes the DCT coefficients to a given accuracy, variable-length codes them and transmits them as the lower-level or 'base
layer’ bitstream. The quantization error introduced by the first quantizer is itself quantized, variable-length coded and transmitted as the upper-level or ‘enhancement-layer’ bitstream. Side information required by the decoder, such as motion vectors, is transmitted only in the base layer.

The base-layer bitstream can be decoded in the same way as the non-scalable case shown in Figure 2.5. To decode the combined base and enhancement layers, both layers must be received, as shown in Figure 2.8. The enhancement-layer coefficient refinements are added to the base-layer coefficient values following inverse quantization. The resulting coefficients are then decoded in the same way as in the non-scalable case.

The SNR profile is suggested for digital terrestrial television as a way of providing graceful degradation.

The spatial profile adds support for enhancement layers carrying the coded image at different resolutions, using the ‘spatial scalability’ tool. Figure 2.9 shows a coder and a decoder which perform spatial scalability. The W block is an adaptative weighting function.

Spatial scalability is characterized by the use of decoded pictures from a lower layer as a prediction in a higher layer. If the higher layer is carrying the image at a higher resolution, then the decoded pictures from the lower layer must be sample-rate converted to the higher resolution by means of an ‘up-converter’.

In the coder shown in Figure 2.9, two coder loops operate at different picture resolutions to produce the base and enhancement layers. The base-layer coder produces a bitstream which may be decoded in the same way as in the non-scalable case. The enhancement-layer coder is offered the ‘up-converted’ locally-decoded, pictures from the base layer, as a prediction for the upper-layer block. This prediction is in addition to the prediction from the upper-layer’s motion-compensated predictor. The adaptive weighting function, W in Figure 2.9 (coder side), selects between the prediction from the upper and lower layers.

As with SNR scalability, the lower-layer bitstream can be decoded in the same way as in the non-scalable case. To decode the combined lower and upper layers, both layers must be received, as shown in Figure 2.9 (decoder side). The lower layer is decoded first and the ‘up-converted’ decoded pictures offered to the upper-layer decoder for possible use as a prediction. The upper-layer decoder selects between its own motion-compensated prediction and the ‘up-converted’ prediction from the lower layer, using a value for the weighting function, W, transmitted in the upper-layer bitstream.

The spatial profile is suggested as a way to broadcast a High-Definition TV (HDTV) service with a main-profile compatible standard-definition service.
The high profile adds support for coding a 4:2:2 video signal and includes the scalability tools of the SNR and spatial profile.

**Details of levels**

MPEG-2 defines four levels of coding parameters constraints. Table 2-2 shows the constraints on picture size, frame rate, bit rate and buffer size for each of the defined levels.

<table>
<thead>
<tr>
<th>Level</th>
<th>Max. frame width (pixels)</th>
<th>Max. frame height (lines)</th>
<th>Max. frame rate (Hz)</th>
<th>Max. bit rate (Mbit/s)</th>
<th>Buffer size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>352</td>
<td>288</td>
<td>30</td>
<td>4</td>
<td>475 136</td>
</tr>
<tr>
<td>Main</td>
<td>720</td>
<td>576</td>
<td>30</td>
<td>15</td>
<td>1 835 008</td>
</tr>
<tr>
<td>High-1440</td>
<td>1440</td>
<td>1152</td>
<td>60</td>
<td>60</td>
<td>7 340 032</td>
</tr>
<tr>
<td>High</td>
<td>1920</td>
<td>1152</td>
<td>80</td>
<td>80</td>
<td>9 781 248</td>
</tr>
</tbody>
</table>

**Table 2-2** MPEG-2 levels: picture size, frame-rate and bit-rate constraints.
Note that the constraints are upper limits and that the codecs may be operated below these limits (e.g. a high-1440 decoder will decode 720 pixels by 576 lines pictures).

In broadcasting terms, standard-definition TV requires main level and high-definition TV requires high-1440 level.

2.5 Conclusions about the MPEG-2 Standard

MPEG-2 has been very successful in defining a specification serving a range of applications, bit rates, qualities and services.

The major interest is in the main profile at main level (MP@ML) for applications such as digital satellite television broadcasting (terrestrial, satellite and cable), video-on-demand services and desktop video systems. Several manufacturers have already announced MP@ML single-chip decoders and multichip encoders. Prototype equipment supporting the SNR and spatial profiles has also been constructed for use in broadcasting field trials.

The specification only defines the bitstream syntax and decoding process. Generally, this means that any decoders which conform to the specification should produce nearly identical output pictures. However, decoders may differ in how they respond to errors introduced in the transmission channel. For example, an advanced decoder might attempt to conceal faults in the decoded picture if it detects errors in the bitstream.

For a coder to conform to the specification, it only has to produce a valid bitstream. This condition alone has no bearing on the picture quality through the codec, and there is likely to be a variation in coding performance between different coder designs. For example, the coding performance may vary depending on the quality of the motion-vector measurement, the techniques for controlling the bit rate, the methods used to choose between the different prediction modes, the degree of picture preprocessing and the way in which the quantizer is adapted according to the picture content.

The picture quality through an MPEG-2 codec depends on the complexity and predictability of the source pictures. Real-time decoders have demonstrated generally good quality standard-definition pictures at bit rates around 6 Mbit/s. As experience of MPEG-2 coding increases, the same picture quality may be achievable at lower bit rates.
The purpose of this Chapter is to briefly describe the different blocks which perform the splitting of the MPEG-2 bitstream into signalling and data information (and the generation of auxiliary information), the lossless compression (Huffman and Lempel Ziv Welch schemes) of the signalling and auxiliary information, the channel coding of the data information and the compressed signalling and auxiliary information, and the reciprocal of these operations. That is, the recovering of the data information and the compressed signalling and auxiliary information (using the side information due to the channel coding), the expansion of the compressed signalling and auxiliary information, and the merging of the signalling and data information with the help of the auxiliary information, in order to recover the original MPEG-2 bitstream (or a corrupted version of it).

### 3.1 Coder Side

Figure 3.1 shows the coder side of the system. The input will be an MPEG-2 bitstream file. This file is generated using the 'mpeg2encode’ program, with the help of a parameter file. Detailed descriptions of these aspects are given in Chapter 4.

![Diagram](image_url)

**Figure 3.1** Coder side of the system.

**Keywords:**
- AI: Auxiliary Information
- CAI: Compressed AI
- CCAI: Channel-encoded CAI
- DCCAI: Disturbed CCAI
- LC: Lossless Compressor
- SI: Signalling Information
- CSI: Compressed SI
- CCSI: Channel-encoded CSI
- DCCSI: Disturbed CCSI
- CC: Convolutional Coder
- DI: Data Information
- CCSI: Channel-encoded CSI
- CCDI: Disturbed CCDI

The first block of the coder side is the Splitting Program (SP). It takes as input the MPEG-2 bitstream, and splits it into Signalling and Data Information (SI and DI, respectively). DI is related to the VLC DCT coefficients and the Motion Vectors. SP also generates Auxiliary Information (AI), used at the decoder side to reconstruct the original bitstream (or a corrupted version of it). A detailed description of the SP block, and the special characteristics of SI, DI and AI is found in Chapter 4.

The second block(s) in this scheme is the Lossless Compressor (LC). It will perform either Huffman or Lempel Ziv Welch (LZW) compression. AI and DI are compressed...
Separating the MPEG-2 Bitstream for Lossless Compression and Transmission

separately, to produce CAI (Compressed AI) and CSI (Compressed SI). DI is not compressed because it is already source encoded by the MPEG-2 standard. A detailed description of Huffman and LZW compression is given in Appendix B. Interesting results about the degree of compression achieved are found in Chapter 5. It is clear that we will succeed in our purpose of compressing the original MPEG-2 bitstream if \( \frac{CSI + CAI + DI}{SI + DI} < 1 \).

The third step in the coding process is to add redundancy to DI, CAI and CSI, in order to send this information through a noisy channel. This is the purpose of the CC block, which performs Convolutional Coding of the three sources of information separately, for reasons that will be explained in Chapter 6. Then, we have at the output of CC the next: CCDI (Channel-encoded DI), CCSI (Channel-encoded CSI) and CCAI (Channel-encoded CAI).

Finally, this information is sent through a channel that will corrupt the three sources of information. Then, disturbed versions of CCDI, CCSI and CCAI arrive to the decoder side. These corrupted sources of information are, as shown in Figure 3.1, DCCDI, DCCSI and DCCCAI.

3.2 Decoder side

Figure 3.2 shows the decoder side of the system. As we can expect, the inputs to the decoder side are the corrupted sources of information DCCDI, DCCSI and DCCCAI.

![Decoder side of the system](image)

**Key words:** AI: Auxiliary Information  
CAI: Compressed AI  
DCCAI: Disturbed CAI  
LE: Lossless Expansion  
SI: Signalling Information  
CSI: Compressed SI  
DCCSI: Disturbed CSI  
VD: Viterbi Decoder  
DCCDI: Disturbed CCDI  
DCCDI: Disturbed CCDI

The structure of the decoder side of the system is a reciprocal version of the coder side. Then, the next block(s) performs the Viterbi Decoding (VD) of DCCDI, DCCSI and DCCCAI. It is important to note that Figure 3.2 assumes that the VD is able to detect all the errors introduced by the channel into the three sources of information. That is, we recover CAI, CSI and DI. This is not a very realistic situation, because although Viterbi decoding is very efficient for large depths \((K\) parameter) in the convolutional encoder, the efficiency degrades
3. General Overview of the System

for small $K$s. For the moment, we will assume perfect recovery, and we will deal with a more realistic situation in Chapter 6.

Once we get CAI, CSI and DI from the VD, the next step is to expand the losslessly compressed information with the help of the Lossless Expansion (LE) block. Now we have AI, SI and DI.

The last step will be, of course, to reconstruct the original MPEG-2 bitstream (as it was at the beginning of the coder side). This is what the Merging Program (MP) does, with the help of AI. The concrete manner to do this is explained in Chapter 7. It is important to recall that really we are going to reconstruct a corrupted version of the original bitstream, due to the noise introduced by the channel. However, even in this case, there are ways to allow the MPEG-2 decoder to handle these erroneous bitstreams, without side information [4].
4. Splitting Program

In this Chapter we are going to describe, with a high degree of detail, how the MPEG-2 bitstream is splitted (or separated) into signalling information (SI) and data information (DI), with the execution of the splitting program (SP). We will show the different kinds of auxiliary information (AI) we need to generate, in order to reconstruct the bitstream at the decoder side. At the end of the Chapter are included several plots with information about the amount of SI we can find in the different bitstreams, for different values of the coding bit rate. Another kind of experiment will be to fix the bit rate, and vary the number of frames. Plots (related to these experiments) showing information about the Peak Signal to Noise Ratio (PSNR) of the Luminance component (Y) of the samples (bitstreams) are also included.

4.1 Experimental Set-up

The first step in order to (give as) input an MPEG-2 bitstream to the SP, is to generate the bitstreams. They are generated from video sequence files (normally named videosequence.yuv), each one containing 300 frames. Each frame is composed of three components: the luminance component (Y), and the chrominance components (U and V). The horizontal and vertical sizes of the Y component are 352 pixels and 288 pixels, respectively. The U and V components have horizontal and vertical sizes that are half of the Y component’s; that is, 176 and 144 pixels, respectively. Each pixel has a size of one byte (8 bits). Then, the size of one frame is:

- Y component: 352x288 pixels (bytes) = 101376 pixels (bytes)
- U component: 176x144 pixels (bytes) = 25344 pixels (bytes)
- V component: 176x144 pixels (bytes) = 25344 pixels (bytes)

Frame size = 101376 + 25344 + 25344 = 152064 pixels (bytes)

Thus, the video sequence files have a size around 40-50 Mbytes. Concretely, these video sequences are:

- akiyo.yuv: This video sequence shows a woman giving the news.
- mad.yuv: This one shows a woman with her child.
- containe.yuv: This one shows a ship moving over the see.

The MPEG-2 bitstreams are generated using the program ’mpeg2encode’, with the help of a parameter file. For the first class of experiments, we do not want to generate the bitstream using the whole video sequences (300 frames). Instead, we decided to generate the bitstreams using only 30 frames from each video sequence file. We will use a higher number of frames
for the second class of experiments. If we choose a display rate of 30 frames/second, the experiments will be performed on one second of video sequence. The program ’sample’ was used to pick up 30 frames from the original (300 frames) video sequence files. This program has two inputs: the start frame and the end frame. After checking the video sequences described above, suitable samples were found with these start and end frames. This is shown in Table 4-1:

<table>
<thead>
<tr>
<th>Video Seq. Name</th>
<th>Start frame</th>
<th>End frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>akiyo.yuv</td>
<td>201</td>
<td>230</td>
</tr>
<tr>
<td>mad.yuv</td>
<td>141</td>
<td>170</td>
</tr>
<tr>
<td>containe.yuv</td>
<td>251</td>
<td>280</td>
</tr>
</tbody>
</table>

Table 4-1 Suitable start and end frames for each video sequence file, in order to generate the MPEG-2 bitstreams.

The ’sample’ program will generate 30 files (frame000.yuv, frame001.yuv, ..., frame029.yuv), each one containing one frame of the chosen video sequence, according to the start and end frames.

We are close to be ready to begin the experiments (the use of the SP program). But it is important to speak a little bit more about the ’mpeg2encode’ program. A normal call to this program is:

    mpeg2encode parameter_file bitstream.m2v

Currently, the ’mpeg2encode’ program is rather limited, in the sense that it can only generate 4:2:0 chromatic format bitstreams (4:2:2 and 4:4:4 chromatic formats not allowed yet), and it does not support scalable extensions. It also has more limitations, like not supporting concealment motion vectors. So, the generated bitstreams (inputs to the SP) will be of 4:2:0 chromatic format, and without any kind of scalability.

A benefit of the SP program is, in this sense, that it was implemented following the MPEG-2 standard [2] rather than the source code of the ’mpeg2encode’ program. This makes it general, and it can support any kind of chromatic formats, and any kind of scalability.

The coding parameters (which will determine, for instance, the nature of the generated bitstream) can be modified by editing the parameter_file (usually bitstream.par). There are a lot of coding parameters [3]. We will focus our attention on those that are most relevant for the experimental set-up. These parameters are:

- Name of source frame files: A printf format string defining the name of the input files. For our case, it will be frame0%03d. Then, the encoder looks for files: frame000, frame001, frame002, …
4. Splitting Program

- Input picture file format: A number defining the format of the source input frames. In our case, it must be set to '1' (.yuv format).
- Number of frames: This defines the length of the sequence in integer units of frames. In our case, it must be set to '30' for the first class of experiments, and will vary in the range 60-300 for the second class of experiments.
- N: Number of frames in a 'Group of Pictures' (GOP). This defines the distance between I-frames (and GOP headers). We set this parameter to '12'.
- M: I-frame / P-frame distance, that is, the distance between consecutive I or P frames. We set this to '3'. N has to be a multiple of M.
- Picture format: This parameter is set to '0', because we want 'frame picture coding', in which both fields of a frame are coded simultaneously.
- Horizontal size: Pixel width of the frames. Set to '352'.
- Vertical size: Pixel height of the frames. Set to '288'.
- Frame rate code: Defines the frame rate. Set this parameter to '5', that corresponds to 30 frames / second.
- Bit rate: A positive floating point value specifying the target bit rate. It will take 10 values for the first class of experiment: 0.1, 0.2, 0.3, ..., 1.0 Mbit/s. And for the second class of experiment it will be fixed to 0.5 Mbit/s. These values are chosen for reasons explained below.

Once these parameters are saved, the 'mpeg2encode' program is executed, and an MPEG-2 bitstream file (bitstream.m2v) is generated. This will serve as the input for the splitting program.

4.2 Bitstream splitting

The bitstream splitting program is called 'cod', and a normal call to this program is

`cod bitstream.m2v`

The key to check the bitstream, and to recognize the different kinds of signalling and data information (SI and DI) is the MPEG-2 syntax, described with no ambiguity in the standard document. In this document, the structure, at bit level, of the different layers is shown, that is, the structure of the video sequence, group of pictures (GOP), picture, slice, macroblock and block layers. The relationship of the different layers, that is, how a video sequence encloses GOP’s, how GOP’s enclose pictures, how pictures enclose slices, how slices enclose
macroblocks, and how macroblocks enclose blocks and motion vectors information, is also shown. From the syntax, it is very easy to identify which bits are signalling information, and which ones are data information. The purpose of the detailed description of the syntax is to implement an MPEG-2 decoder. We do not want to decode the bitstream, we want to split it. But for our aim we can use the structures (i.e. functions) described in the standard. In fact, if we take a look at the source code of the ‘cod’ program (cod.c), it has a very similar structure compared to the MPEG-2 syntax.

There are several kinds of SI. We can find SI in all the layers, except in the block layer, where all the information is DI, that is, the quantized VLC DCT coefficients. We can also find DI in the macroblock layer, that is, motion vectors information. In order to store this information separately, we will define seven files: four files will be SI files, two files will be DI files, and the last one is the AI (auxiliary information) file. Figure 4.1 shows a scheme of the SP, with the output files it will generate.

![Figure 4.1 Scheme of the Splitting Program (SP), showing the files (SI, DI and AI) generated.](image)

**Keywords:** HL=High Level; SLC=Slice; MB=Macroblock; MV=Motion Vectors; DCT=Discrete Cosine Transform (coefficients)

The High Level SI file stores the information related to the video sequence, GOP and picture layers. It also contains stuffing information. In order to keep constant bit rate, the MPEG-2 encoder inserts stuffing bytes (and bits) along the bitstream. This stuffing information should not be taken into account by an MPEG-2 decoder, but we have to keep it, in order to reconstruct the original bitstream at the decoder side. We decided to group together the SI related to the video sequence, GOP and picture layers because of the small size of the samples. Even in this case, the High Level file is always the smallest SI file. In future
4. **Splitting Program**

experiments, with a large amount of frames, we can consider separately the picture SI, only with small and easy modifications of the SP. The rest of the SI files contain information related to the slice and the macroblock layers, and the motion vectors.

The other relevant files to consider are the DI files, which store information about the quantized VLC DCT coefficients and about the motion vectors, respectively. It is important to recall that there exist both data and signalling information related to the motion vectors, and they have to be stored separately.

In order to make things easier, we are going to assign a number to each of the SI and DI files. These are the assignments:

- High Level SI …………… File no. 1
- Slice SI ………………… File no. 2
- Macroblock SI…………..File no. 3
- Motion Vectors SI……….File no. 4
- Quantized VLC DCT DI…File no. 5
- Motion Vectors DI………..File no. 6

The third type of information generated is placed in the auxiliary information (AI) file. This file contains information related to the number of slices per picture, the number of macroblocks per slice, and the amount of stuffing bits and bytes. This will be further described in 4.3.

The key function to separate (or split) the information from the bitstream is the 'get_put_bits' (**gbp** in the splitting program) function. It takes any number of bits from the bitstream, and places them in the file indicated at the input of the function. It has this form:

\[
gbp (ofile, n, pb)
\]

where `ofile` is one of the five files where we store either signalling or data information, `n` is the number of bits taken from the input (bitstream) file (and placed in one of the five files), and `pb` stands for 'position inside the byte', an integer variable varying in the range (0…7) which represents the current bit position inside the current address (byte), from which we fill each output (DI and SI) file. There are six of this class of variable: one for the input file (`pbi`), and five for the DI and SI files.

A typical call to the **gbp** function should be **gbp (ofile3, 5, &pb3)**. This call will take the next 5 bits from the bitstream, and will place them into the SI file no. 3, (that is, into the macroblock SI file) from position pb3 inside the current byte. The symbol `&` means that we pass the variable pb3 to the **gbp** function as reference (not as value) because we need to update its value each time we call the **gbp** function. Then, the **gbp** function is able to get (from
the input) and put (to the output) information, this is important, at bit level. That is, we do not need to take the bits from the first bit of one byte, and place them from the first bit of one byte. The function allows to take bits from the bitstream at any position inside a byte, and place them at any position inside a byte. This is done with the help of the bitwise operators included in the C programming language. Figure 4.2 shows schematically how two consecutive calls are done to the gpb function:

\[
gpb (ofile5, 7, &pb5);
\]

\[
gpb (ofile1, 4, &pb1);
\]

\[
\begin{array}{c}
\text{Byte n-1} \\
\text{Byte n} \\
\text{Byte n+1}
\end{array}
\]

\[
\begin{array}{c}
0 0 1 \\
0 0 1 1 \\
0 0 1 1 1 0
\end{array}
\]

\[
\begin{array}{c}
0 0 0 1 0 1 1 \\
0 0 1 1 1 0
\end{array}
\]

\[
\begin{array}{c}
0 0 1 1 1 \\
0 0 1 1 1 0
\end{array}
\]

\[
\begin{array}{c}
\text{File no.5 (DI)} \\
\text{File no. 5 (DCT DI)}
\end{array}
\]

\[
\begin{array}{c}
\text{File no.1 (High Level SI)}
\end{array}
\]

**Figure 4.2** How two consecutive calls to the gpb function split the bitstream file, and fill two of the output files.

For this example, we can consider that the 'position inside the input file' is equal to 6 (pbi = 4), and that pb5 = 2 and pb1 = 3. Once both instructions are executed, the VLC DCT DI file is filled with the next 4 bits of the input file, and the next 7 ones are placed in the High Level SI file. Of course, pbi, pb5 and pb1 are updated to 1, 1 and 7, respectively.
4.3 Reasons for Auxiliary Information

Once the MPEG-2 bitstream is separated into signalling and data information files, we need to use auxiliary information, stored in the AI, in order to reconstruct it again. There are three sources of AI:

- Number of macroblocks per slice. It comes from the way in which macroblocks are enclosed by slices. Here we show a fragment of the MPEG-2 syntax, related to the slice layer:

```c
slice () {
  slice_start_code;
  .....................
  do {
    macroblock ();
  } while ((nextbits () != '000 0000 0000 0000 0000')
  next_start_code ();
}
```

This piece of code, concretely, the 'do-while' loop means that when we are at the macroblock layer, if we find the sequence '000 0000 0000 0000 0000'\(_2\) then the decoder knows that this group of macroblocks enclosed by the current slice are finished, and the checking of the next slice (if there is) begins. Taking into account this fact, it is very easy to put together the SI related to the macroblock layer, following the MPEG-2 syntax, as mentioned before. The problem arises when we want to reconstruct the original bitstream at the decoder side. There is no way to know whether the next macroblock SI being checked belongs to the current slice, or to the next one, unless we keep track about the number of macroblocks per slice. This is done by defining a counter (macroblocks\_per\_slice) and an array of integers (number\_of\_macroblocks[N]) where we place the number of macroblocks per slice. While we are in the 'do-while' loop mentioned above, the counter is incremented by one, and once we leave this loop, we write in the next element of the array the current value of the counter. Then, at the decoder side, we have only to replace the 'do-while' loop by a 'for' loop, executed the number of times found in the elements of the array previously created and filled in the execution of the Separating Program. After several experiments, it was found that the number of macroblocks per intra picture is constant, but it is variable for predictive and bidirectional pictures.

- Number of slices per picture. This second source of AI has a very similar nature compared with the previous one, but in a higher level. Let us take a look to the MPEG-2 syntax function where we find the slices:
Separating the MPEG-2 Bitstream for Lossless Compression and Transmission

```c
picture_data () {
    do {
        slice ();
    } while (nextbits () == slice_start_code)
    next_start_code ();
}
```

Again, we have a 'do-while' loop, where all the slices of the current picture being checked are enclosed. It is not difficult to put together the SI associated with each slice. The problem is at the decoder side, because actually we do not know whether the current slice belongs to the next picture, or the the current one. Then, we define a counter (slices_per_picture) and an array of integers (number_of_slices[M]) and keep track of the number of slices per picture, similarly to the number of macroblocks per slice. And finally we replace in the Reconstructing Program the 'do-while' loop by a 'for' loop, executed the number of times found in the next element of the array. After several experiments, it was found that the number of slices per picture is constant for all types of pictures, and equal to 18, for this kind of video sequence. But we do not assume any knowledge in advance about the bitstream, because the program was implemented to be as general as possible, and the number of slices per picture will be different for other kinds of video sequences, with different horizontal and vertical sizes.

Figures 4.3 and 4.4 show clearly the last points about the auxiliary information related to the number of macroblocks per slice, and the number of slices per picture.

![Figure 4.3](image)

**Figure 4.3** Example for a MPEG-2 bitstream in a predictive picture, showing the different SI and DI parts.

**Key words:** qsc=quantizer_scale_code; ebs=extra_bit_slice; mai=macroblock_address_increment; mm=macroblock_modes; mv=motion_vectors; cbp=coded_block_pattern.

In this figure, each term 'blockn' encloses several quantized VLC DCT coefficients. This is how a piece of bitstream looks like before separating it in different files. Figure 4.4 shows how the SI associated with the slices and the macroblocks look like. The Motion Vector SI
and DI files, and the VLC DCT DI file are filled in a similar way, when executing the Separating Program.

**Slice SI File**

```
...... /Slice15 SI /Slice16 SI /Slice17 SI /Slice0 SI /Slice1 SI ...... /Slice17 SI /Slice0 SI ......
```

**MB SI File**

```
...... /MB4 SI /MB5 SI /MB0 SI /MB1 SI ...... /MB11 SI /MB0 SI ...... /MB7 SI /MB0 SI ......
```

*Figure 4.4* Structure of the Slice SI and Macroblock SI files. The number of slices per picture is constant, but not the number of macroblocks per slice, inside a P picture, as we can see.

So, to know that, for example, 'Slice0 SI' belongs to 'Picture 34' and not to 'Picture 33', we need auxiliary information. The same is true for the Macroblock SI file.

-) *Stuffing Information*

The last class of AI is related to the stuffing information ('zero' bits and bytes) placed along the bitstream in order to have constant bit rate, as mentioned earlier. When an MPEG-2 decoder finds stuffing information, it neglects it, because it is meaningless. The function used by the MPEG-2 syntax to remove any zero bit and zero byte stuffing and to locate the next 'start code' (special fixed code of length equal to 32 bits) has this form:

```plaintext
next_start_code () {
    while (!bytealigned ())
        zero_bit;
    while (nextbits () != '0000 0000 0000 0000 0000 0000 0000 0001')
        zero_byte;
}
```

This function checks whether the current position is byte aligned. If it is not, stuffing bits are present. After that any number of zero stuffing bytes may be present before the 'start code'. Therefore, start codes are always byte aligned and may be preceded by any number of stuffing bits. Every layer (except the macroblock and block layers) has its own 'start code', and the slice layer has several ones. The fact is that we do not know how many stuffing bits and bytes are placed along the bitstream, and if we want to reconstruct the original bitstream, we need to keep track about this. However, in some applications, a conversion from fixed bit rate to variable bit rate by means of stuffing removal is interesting, but this is not our case. Again, we define one counter (*number_of Stuffing*) and one array of integers (*stuffing[LF]*) where we place the number of stuffing information that we find when our own version of the *next_start_code* function is executed in the splitting program. And, at the decoder side, we
replace the 'while' loops, and when we have a call to the `next_start_code` function, we fill the reconstructed bitstream file with the number of bits indicated by the next element in the 'stuffing' array.

We have seen that the reasons for auxiliary information are essentially related to the problems introduced by 'do-while' and 'while' loops found at slice and picture layers in the MPEG-2 syntax. Anyway, the amount of generated AI is not large compared to the size of the original bitstream. In the worst cases, the (AI size)/(bitstream size) ratio is around 2-3%, and normally it is even lower. For the first class of experiments (fixed number of frames = 30, and variable bit rate in the range 0.1-1.0 Mbit/s), the size of the AI file is around 1200 bytes before compression, and around 500-600 bytes after compression, using either Huffman or Lempel Ziv Welch schemes. The size of the bitstream files for this type of experiment varies in the range 40 Kbytes to 120 Kbytes, so the generated AI is not a major problem when studying the degree of compression that we can achieve. For the second class of experiments (fixed bit rate = 0.5 Mbit/s, variable number of frames in the range 60 to 300 frames), the ratio between the AI size and the bitstream size is less than 2%.

4.4 SI, DI and PSNR Plots

In this section the results obtained for the two classes of experiments are shown. We will see the variation of SI, DI and PSNR vs. Bit Rate, for fixed number of frames. Then, the variation of SI, DI and PSNR vs. Number of Frames, for fixed bit rate. All the values used to draw the plots are found in Appendix A, together with other results not plotted. In this Appendix, there are also data about the amount of generated auxiliary information.

The ratio between the SI and the DI with respect to the bitstream size is analyzed in two different ways: First, the total SI and DI ratios, and after this, the I picture, P picture and B picture SI and DI ratios separately.

**SI, DI and PSNR vs. Bit Rate, with fixed number of frames**

For this class of experiments, the bit rate range was chosen to be 0.1-1.0 Mbit/s, because the PSNR of the Luminance Component (Y) of the pictures lies approximately in the range 30-40 dB, which corresponds to acceptable image quality. The number of frames was fixed to 30, because it corresponds to 1 second of video sequence, for a frame-rate of 30 frames/s.
### Video Sequence 'mad.yuv'

**Figure 4.5** Total SI (%) vs. Bit Rate (Mbit/s) for the video sequence 'mad.yuv'.

**Figure 4.6** DCT DI and MV DI vs. Bit Rate (Mbit/s) for the video sequence 'mad.yuv'.
Figure 4.7 I, P and B Pictures SI vs. Bit Rate (Mbit/s) for the video sequence 'mad.yuv'.

Figure 4.8 PSNR (Y component) vs. Bit Rate (Mbit/s) for the video sequence 'mad.yuv'.
4. Splitting Program

Figure 4.9 I, P and B Pictures DI vs. Bit Rate (Mbit/s) for the video sequence 'mad.yuv'.

Video Sequence 'akiyo.yuv'

Figure 4.10 Total SI vs. Bit Rate (Mbit/s) for the video sequence 'akiyo.yuv'.
Figure 4.11 DCT DI and MV DI vs. Bit Rate (Mbit/s) for the video sequence ‘akiyo.yuv’.

Figure 4.12 PSNR (Y component) vs. Bit Rate (Mbit/s) for the video sequence ‘akiyo.yuv’.
4. Splitting Program

**Figure 4.13** I, P and B Pictures SI vs. Bit Rate (Mbit/s) for the video sequence 'akiyo.yuv'.

**Figure 4.14** I, P and B Pictures DI vs. Bit Rate (Mbit/s) for the video sequence 'akiyo.yuv'.

---

37
Video Sequence 'containe.yuv'

**Figure 4.15** Total SI vs. Bit Rate (Mbit/s) for the video sequence 'containe.yuv'.

**Figure 4.16** DCT DI and MV DI vs. Bit Rate (Mbit/s) for the video sequence 'containe.yuv'.
Figure 4.17 PSNR (Y component) vs. Bit Rate (Mbit/s) for the video sequence ’containe.yuv’.

Figure 4.18 I, P and B Pictures SI vs. Bit Rate (Mbit/s) for the video sequence ’containe.yuv’.
Here, we are going to comment the results plotted above. Regarding to the 'total signalling information vs. bit rate' plots, we can see how the relative amount (RA) of SI decreases from around 30% to 15% when increasing the bit rate, for all the video sequences. This is a coherent result, because when we increase the bit rate, the MPEG-2 coder uses more bits to variable-length code each block, thus increasing the RA of DI. We can also see that this decreasing is not constant. Instead, there is a certain bit rate, around 0.4-0.6 Mbit/s, where the slope of the SI percentage curve is higher. If we take a look to the DCT DI-MV DI curves, there is a coincidence between that point and the bit rate for which the RA of DCT DI increases faster and the RA of MV DI decreases faster as well. For 'mad.yuv' and 'akiyo.yuv' it is interesting to note how the MV DI percentage is larger than the DCT DI percentage for low bit rates, but the situation changes when increasing the bit rate (there is a cross-over point). Regarding the PSNR plots, we can see a clear symmetry between them and the SI percentage plots, and how the PSNR improves when the bit rate is higher. This is logical, if we consider that the PSNR is the ratio between the maximum quantization level (in this case, $2^8 = 256$) and the mean square error (MSE) for each pixel. That is,

$$PSNR(dB) = 10 \log \frac{256^2}{E[x - \tilde{x}]^2},$$

where $x$ is the real value of the pixel, and $\tilde{x}$ is the quantized value. So, if the MPEG-2 coder uses more bits (higher bit rate) to code the blocks (8x8 pixels), it means that the quantization step is smaller, then the MSE will be lower and the PSNR will be higher. We can see how the
PSNR is more or less constant for low bit rates (0.1-0.3 Mbit/s). The reason could be that the coarsest quantizer is chosen all the time for these target bit rates.

With respect to the other kind of plots, that is the I, P and B pictures SI and DI percentages vs. the bit rate, there are interesting results. For all the sequences, the most SI-consuming pictures are the bidirectional (B) ones, as we could expect. The B pictures are the ones that obtain the highest compression, see Chapter 2. This means that they are coded with less amount of DI bits than I or P pictures. In the DI plots, for low bit rates, the RA of B-pictures DI is the highest because they constitute the majority of frames. The I-pictures (minority of the frames) are coded with a low amount of bits. We can see how the situation changes drastically when increasing the bit rate. For the SI, it is shown how, for I- and P-pictures, the RA of SI remains more or less constant, that is, the (absolute) size of SI for I- and P-pictures grows with the bit rate proportionally to the bitstream size. However, although the absolute amount of SI for B-pictures grows with the bit rate, the RA decreases, because this ‘growth rate’ is slower than the ‘growth rate’ of the bitstream size. Finally, it is possible to see strong similarities between the total SI and MV DI percentages curves and the SI and DI percentages curves of the B-pictures.

**SI, DI and PSNR vs. number of frames, with fixed bit rate**

In this class of experiments, the bit rate was chosen to be 0.5 Mbit/s, because it is the lowest bit rate for which all the generated bitstreams have a PSNR of the Y component greater than 30.0 dB (the generated bitstream using ’containe.yuv’ -30 frames-, with 0.4 Mbit/s has a PSNR equal to 27.7 dB, and with 0.5 Mbit/s, the PSNR is equal to 30.3 dB). The frame range was chosen to be 60-300, in steps of 60 frames.
*Video Sequence ‘mad.yuv’*

**Figure 4.20** Total SI, DCT DI & MV DI vs. #frames for the video sequence ‘mad.yuv’.

**Figure 4.21** I, P and B Pictures SI vs. #frames for the video sequence ‘mad.yuv’
4. Splitting Program

![Graph 1](image1.png)

**Figure 4.22** I, P and B Pictures DI vs. #frames for the video sequence 'mad.yuv'

![Graph 2](image2.png)

**Figure 4.23** PSNR (Y component) vs. #frames for the video sequence 'mad.yuv'
Video Sequence ‘akiyo.yuv’

Figure 4.24 Total SI, DCT DI and MV DI vs. #frames for the video sequence ‘akiyo.yuv’.

Figure 4.25 I, P and B Pictures SI vs. #frames for the video sequence ‘akiyo.yuv’
4. Splitting Program

**Figure 4.26** I, P and B Pictures DI vs. #frames for the video sequence ’akiyo.yuv’.

**Figure 4.27** PSNR (Y component) vs. #frames for the video sequence ’akiyo.yuv’.
**Video Sequence ’containe.yuv’**

**Figure 4.28** Total SI, DCT DI and MV DI vs. #frames for the video sequence ’containe.yuv’.

**Figure 4.29** I, P & B Pictures SI vs. #frames for the video sequence ’containe.yuv’.
4. Splitting Program

Figure 4.30 I, P and B Pictures DI vs. #frames for the video sequence 'containe.yuv'

Figure 4.31 PSNR (Y component) vs. #frames for the video sequence 'containe.yuv'.
The results shown in the plots about the second type of experiments give a clear idea of the regularity of the video sequences under test. In fact, these are fixed-camera sequences, and there are few moving people or objects in them. So, increasing the number of frames, the SI and DI files increase their size more or less at the same rate as the entire bitstream does. So the SI and DI percentages remain approximately constant from 60 to 300 frames. Similar results are shown for the I, P, B Pictures SI and DI. There is a maximum variation of \(~3\%\) for the I Pictures DI file in ‘mad.yuv’. The situation is the same for the PSNR of the Y component: hardly 1-2 dB of variation from 60 to 300 frames.

In this Chapter, the most interesting results belong to the first type of experiments. They provide knowledge of how the bitstream is constituted, and how it varies when increasing the bit rate, from two points of view: the amount of SI and DI distributed in the different layers, and the amount of SI and DI distributed between the different type of pictures: I, P and B.

The real purpose of the second type of experiment will be clear in the next Chapter, which deals with lossless compression. While increasing the bit rate, the compression ratio degrades, but increasing the number of frames, the compression ratio improves, and precisely this is one of our goals.
5. Lossless Compression of Signalling Information

5.1 Introduction

In this Chapter, results about lossless compression of the signalling (and the auxiliary) information are shown. The next section shows experimental results obtained with Huffman compression, and with Lempel Ziv Welch (LZW) compression. The results are based on both types of experiments, as they were described in the previous Chapter. All values can be found in tables from Appendix A, together with other data.

The degree of compression obtained will be equal to:

\[
CR(\%) = 100 \cdot \left( 1 - \frac{CSI + CAI + DI}{SI + DI} \right),
\]

where \(CR\) (%) is the Compression Ratio (in percentage), \(CSI\) is the Compressed Signalling Information, \(CAI\) is the Compressed Auxiliary Information, \(SI\) is the Signalling Information (before compression), and \(DI\) is the Data Information. It is clear then that \(SI + DI\) is equal to the size of the original bitstream.

The last section of this Chapter deals with the compressed MPEG-2 bitstream format, for storage purposes (no channel coding needed).

5.2 Huffman Coding vs. Lempel Ziv Welch Coding

In a similar way as in the previous Chapter, first we are going to show the compression results for the first type of experiment, that is, fixed number of frames (equal to 30), and variable bit rate (in the range 0.1-1.0 Mbit/s). Following these results are shown those related to the second type of experiment, that is, fixed bit rate (equal to 0.5 Mbit/s), and variable number of frames (in the range 60-300 frames).
Compression Ratio (%) vs. Bit Rate, with fixed number of frames

Video Sequence 'mad.yuv'

Figure 5.1 Huffman CR (%) vs. Bit Rate (Mbit/s) for the video sequence 'mad.yuv'.

Figure 5.2 LZW CR (%) vs. Bit Rate (Mbit/s) for the video sequence 'mad.yuv'.
5. Lossless Compression of Signalling Information

**Video Sequence 'akiyo.yuv'**

![Huffman Compression Ratio](image)

**Figure 5.3** Huffman CR (%) vs. Bit Rate (Mbit/s) for the video sequence 'akiyo.yuv'.

![LZW Compression Ratio](image)

**Figure 5.4** LZW CR (%) vs. Bit Rate (Mbit/s) for the video sequence 'akiyo.yuv'.

51
**Video Sequence 'containe.yuv'**

**Huffman Compression Ratio (#frames = 30)**

![Huffman Compression Ratio Graph](image)

*Figure 5.5* Huffman CR (%) vs. Bit Rate (Mbit/s) for the video sequence 'containe.yuv'.

**LZW Compression Ratio (#frames = 30)**

![LZW Compression Ratio Graph](image)

*Figure 5.6* LZW CR (%) vs. Bit Rate (Mbit/s) for the video sequence 'containe.yuv'.
5. Lossless Compression of Signalling Information

For this first type of experiment, we can see clearly how much better LZW compression is compared to Huffman compression. We can also see how, in both cases, the degree of compression degrades when increasing the bit rate. This is normal, because the lossless compression is only performed in the SI part of the bitstream, and the SI percentage decreases when increasing the bit rate. Then, we have less degree of freedom to achieve high compression ratios. But, even in the worst cases (Huffman compression, high bit rate) we have a little compression (less than 1%), which is a good result, in the sense that we expected expansion in these bad cases. It is also shown that the low amount of auxiliary information does not prevent compression.

An interesting observation is that the CR degradation is not constant, and it has a larger slope in the range 0.4-0.5 Mbit/s. Then, we can see a coincidence with the plots in the previous Chapters, in which the SI curves had similar shape, and the PSNR curves had a symmetrical shape. Maybe the 'mpeg2encode' program works well with bit rates over 0.4-0.5 Mbit/s, but not with bit rates below this range.

Compression Ratio (%) vs. Number of Frames, with fixed bit rate

*Video Sequence 'mad.yuv'

![Huffman Compression Ratio (Bit Rate = 0.5 Mbit/s)](image)

**Figure 5.7** Huffman CR (%) vs. #frames for the video sequence ‘mad.yuv’. 

53
Figure 5.8 LZW CR (%) vs. #frames for the video sequence 'mad.yuv'.

Video Sequence 'akiyo.yuv'

Figure 5.9 Huffman CR (%) vs. #frames for the video sequence 'akiyo.yuv'.
5. Lossless Compression of Signalling Information

**Figure 5.10** LZW CR (%) vs. #frames for the video sequence ‘akiyo.yuv’.

**Video Sequence ‘containe.yuv’**

**Figure 5.11** Huffman CR (%) vs. #frames for the video sequence ‘containe.yuv’.
As we could expect, increasing the number of frames, the degree of lossless compression that we can achieve is greater. For example, with 30 frames, at 0.5 Mbit/s, the LZW CR for ‘akiyo.yuv’ is 6.3%, but with 300 frames, at the same bit rate, the LZW CR is 8.5% This is because larger files are easier to compress for Huffman and LZW. If we take a look at the tables in Appendix A which correspond to lossless compression, all the SI (and the AI) files are more compressed when increasing the number of frames. Again, the performance of LWZ is better, but we can see that for both schemes, the compression increases slower for large number of frames.

There is a main limitation in order to achieve higher degrees of compression for both types of experiment: the macroblock SI file. We can see in the tables in Appendix A that this is the most SI consuming layer, though the degree of compression of these SI files is very low compared to other SI files (high level, slice, mv) much smaller in size. An explanation of this ‘bad compression’ behaviour of the macroblock SI files is due to the lack of redundancy in these SI files. This, together with the limited degree of freedom (we only focus our compression effort in the SI and AI files, not in the DI files, which are much larger) are the reasons that cause that the highest compression ratio is 14.0% for the first type of experiments (‘akiyo.yuv’, at 0.1 Mbit/s), and 9% for the second type (‘containe.yuv’, with 300 frames).
5.3 Compressed MPEG-2 Bitstream Format

This section is focused on storage purposes of the compressed MPEG-2 bitstream. This means that we are not going to consider any channel coding, which will be treated in Chapter 6. Figure 5.13 shows a schematic description of how an MPEG-2 bitstream is compressed. The first part of the block diagram is similar to the first part of the coder side of the system, shown in Figure 3.1. The Separating Program (SP) splits the input bitstream in six different files (four SI files and two DI files), plus an auxiliary file (AI). Then, the SI and AI files are losslessly compressed (LC). To put everything together in one file, we make use of the Formating Program (FP). This program takes as input the five compressed files (CSI’s and CAI) and the two DI files. Now it calculates the size of each of the files, and places this information in a ‘header’ (an array). We can use, for example, long integers (four bytes size) as elements of the header array to store the size of each of the seven files. Thus, the size of the ‘header’ will be $4 \times 7 = 28$ bytes. The order of size storage could be the same as that shown in Figure 5.13. We create a new file, bitstream.cm2v, where we first store the header array. Finally, we place in this new file the CAI, CSI and DI files, in the same order as their sizes are stored in the header.

![Diagram](image_url)

**Figure 5.13** Schematic description of how to create a compressed MPEG-2 file.
The recovery of the CAI, CSI and DI files will be done with a reciprocal version of the Formating Program, as shown in Figure 5.14.

**Figure 5.14** Recovery of the original MPEG-2 bitstream, from the compressed bitstream file, with the help of the Inverse Formating Program (IFP).

We first get the header, whose size and order of storage we already know. Then, it will be easy to split the compressed bitstream file into the seven original files. The five compressed files are expanded, and finally we obtain the original MPEG-2 bitstream with the help of the merging program.

This works well for storage purposes. What happens for transmission through a noisy channel? This is the subject of the next Chapter, dealing with convolutional coding and Viterbi decoding.
6. Convolutional Coding and Viterbi Decoding

6.1 Introduction

The purpose of forward error correction (FEC) is to improve the capacity of a channel by adding some carefully designed redundant information to the data being transmitted through the channel. The process of adding this redundant information is known as channel coding. Convolutional coding and block coding are the two major forms of channel coding. Convolutional codes operate on serial data, one or a few bits at a time. Block codes operate on relatively large (typically, up to a couple of hundred bytes) message blocks. There are a variety of useful convolutional and block codes, and a variety of algorithms for decoding the received encoded information sequences to recover the original data.

Convolutional coding with Viterbi decoding is a FEC technique that is particularly suited to a channel in which the transmitted signal is corrupted mainly by additive white gaussian noise (AWGN) [5]. AWGN can be seen as noise whose voltage distribution over time has characteristics that can be described using a Gaussian, or normal, statistical distribution, i.e. a bell curve. This voltage distribution has zero mean and a standard deviation that is a function of the signal-to-noise ratio (SNR) of the received signal. If it is assumed that the received signal level is fixed, then if the SNR is high, the standard deviation of the noise is small, and vice-versa. Many radio channels are AWGN channels, but many, particularly terrestrial radio channels also have other impairments, such as multipath, selective fading, interference, and atmospheric (lightning) noise. Transmitters and receivers can add spurious signals and phase noise to the desired signal as well. Although convolutional coding with Viterbi decoding might be useful in dealing with those other problems, it may not be the most optimal technique.

Convolutional codes are usually described using two parameters: the code rate and the constraint length. The code rate, $k/n$, is expressed as a ratio of the number of bits into the convolutional encoder ($k$) to the number of channel symbols output by the convolutional encoder ($n$) in a given encoder cycle. The constraint length parameter, $K$, denotes the "length" of the convolutional encoder, i.e. how many $k$-bit stages are available to feed the combinatorial logic that produces the output symbols. Closely related to $K$ is the parameter $m$, which indicates how many encoder cycles an input bit is retained and used for encoding after it first appears at the input to the convolutional encoder. The $m$ parameter can be thought of as the memory length of the encoder.
Viterbi decoding is one of two types of decoding algorithms used with convolutional coding. The other type is sequential decoding. Sequential decoding has the advantage that it can perform very well with long-constraint-length convolutional codes, but it has a variable decoding time. Viterbi decoding has the advantage that it has a fixed decoding time. It is well suited to hardware decoder implementation. But its computational requirements grow exponentially as a function of the constraint length, so it is usually limited in practice to constraint lengths of $K = 9$ or less.

### 6.2 Convolutional Coding

Convolutionally encoding the data is accomplished using a shift register and associated combinatorial logic that performs modulo-two addition. A shift register is merely a chain of flip-flops wherein the output of the $n^{th}$ flip-flop is tied to the input of the $(n+1)^{th}$ flip-flop. Every time the active edge of the clock occurs, the input to the flip-flop is clocked through to the output, and thus the data are shifted over one stage. The combinatorial logic is often in the form of cascaded exclusive-or gates. The exclusive-or gate performs modulo-two addition of its inputs. When $q$ two-input exclusive-or gates are cascaded, with the output of the first one feeding one of the inputs of the second one, the output of the second one feeding one of the inputs of the third one, etc., the output of the last one in the chain is the modulo-two sum of the $q + 1$ inputs. A convolutional encoder, for a rate $1/2$, $K = 3$, $m = 2$ code, is shown in Figure 6.1.

![Convolutional encoder, with $r = 1/2$, $K = 3$, $m = 2$.](image)

In this encoder, data bits are provided at a rate of $k$ bits per second. Channel symbols are output at a rate of $n = 2k$ symbols per second. The input bit is stable during the encoder cycle. The encoder cycle starts when an input clock edge occurs. When the input clock edge
occurs, the output of the left-hand flip-flop is clocked into the right-hand flip-flop, the previous input bit is clocked into the left-hand flip-flop, and a new input bit becomes available. Then the outputs of the upper and lower modulo-two adders become stable. The output selector (SEL A/B block) cycles through two states— in the first state, it selects and outputs the output of the upper modulo-two adder; in the second state, it selects and outputs the output of the lower modulo-two adder.

Let us consider an example input data stream, and the corresponding output data stream:

Let the input sequence be 01011001010001. Assume that the outputs of both of the flip-flops in the shift register are initially cleared, i.e. their outputs are zero. The first clock cycle makes the first input bit, a zero, available to the encoder. The flip-flop outputs are both zero. The inputs to the modulo-two adders are all zeroes, so the output of the encoder is 00. The second clock cycle makes the second input bit available to the encoder. The left-hand flip-flop clocks in the previous bit, which was a zero, and the right-hand flip-flop clocks in the zero output by the left-hand flip-flop. The inputs to the top modulo-two adder are 100, so the output is a one. The inputs to the bottom modulo-two adder are 10, so the output is also a one. So the encoder outputs 11 for the channel symbols. The third clock cycle makes the third input bit, a zero, available to the encoder. The left-hand flip-flop clocks in the previous bit, which was a one, and the right-hand flip-flop clocks in the zero from two bit-times ago. The inputs to the top modulo-two adder are 010, so the output is a one. The inputs to the bottom modulo-two adder are 00, so the output is zero. So the encoder outputs 10 for the channel symbols. And so on. Figure 6.2 shows the timing diagram which illustrates the process. After all of the inputs have been presented to the encoder, the output sequence will be:

00 11 10 00 01 10 01 11 11 10 00 10 11 00 11

Notice that the first bit in each pair is the output of the upper modulo-two adder; the second bit in each pair is the output of the lower modulo-two adder.

It can be seen from the structure of the rate 1/2, \( K = 3 \) convolutional encoder and from the example given above that each input bit has an effect on three successive pairs of output symbols. That is an extremely important point and that is what gives the convolutional code its error-correcting power. The reason why will become evident when we get into the Viterbi decoder algorithm.
It will be more useful to consider the encoder as a simple state machine. The example encoder has two bits of memory, so there are four possible states. Let us give the left-hand flip-flop a binary weight of $2^1$, and the right-hand flip-flop a binary weight of $2^0$. Initially, the encoder is in the all-zero state. If the first input bit is a zero, the encoder stays in the all-zero state at the next clock edge. But if the input bit is a one, the encoder transitions to the 10 state at the next clock edge. Then, if the next input bit is zero, the encoder transitions to the 01 state, otherwise, it transitions to the 11 state. Table 6-1 gives the next state given the current state and the input, with the states given in binary notation. This table is often called a state transition table. Table 6-2 lists the channel output symbols, given the current state and the input data. With these two tables, it is possible to describe the behavior of the example rate $\frac{1}{2}$, $K = 3$ convolutional encoder. Note that both of these tables have $2^{(K-1)}$ rows, and $2^k$ columns, where $K$ is the constraint length and $k$ is the number of bits input to the encoder for each cycle. These two tables will come in handy when discussing the Viterbi decoder algorithm.
6. Convolutional Coding and Viterbi Decoding

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input = 0:</th>
<th>Input = 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 6-1 Next state, given the current state and the input.

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input = 0:</th>
<th>Input = 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 6-2 Channel output symbols, given the current state and the input data.

The next step is to map the one/zero output of the convolutional encoder onto an antipodal baseband signaling scheme, which is simply a matter of translating zeros to +1:s and ones to –1:s. This can be accomplished by performing the operation \( y = I - 2x \) on each convolutional encoder output symbol.

Now we have to add noise to the transmitted channel symbols produced by the convolutional encoder, that involves generating Gaussian random numbers, scaling the numbers according to the desired energy per symbol to noise density ratio, \( E_s/N_0 \), and adding the scaled Gaussian random numbers to the channel symbol values.

In the AWGN channel, the signal is corrupted by additive noise, \( n(t) \), which has the power spectrum \( N_0/2 \) (W/Hz). The variance \( \sigma^2 \) of this noise is equal to \( N_0/2 \). If we set the energy per symbol \( E_s \) equal to 1, then \( \frac{E_s}{N_0} = \frac{1}{2\sigma^2} \). So, \( \sigma = \sqrt{\frac{1}{2(\frac{E_s}{N_0})}} \).

The last step to perform before the Viterbi decoding is to quantize the received channel symbols. An ideal Viterbi decoder would work with infinite precision, or at least with
floating-point numbers. In practical systems, we quantize the received channel symbols with one or a few bits of precision in order to reduce the complexity of the Viterbi decoder, not to mention the circuits that precede it. If the received channel symbols are quantized to one-bit precision (< 0V = 1, > 0V = 0), the result is called hard-decision data. If the received channel symbols are quantized with more than one bit of precision, the result is called soft-decision data. A Viterbi decoder with soft decision data inputs quantized to three or four bits of precision can perform about 2 dB better than one working with hard-decision inputs. The usual quantization precision is three bits. More bits provide little additional improvement.

The selection of the quantizing levels is an important design decision because it can have a significant effect on the performance of the link. The following is a very brief explanation of one way to set those levels. Let us assume our received signal levels in the absence of noise are -1V = 1, +1V = 0. With noise, our received signal has mean +/- 1 and standard deviation $\sigma = \sqrt{1/(2 \cdot (E_s/N_0))}$. Let us use a uniform, three-bit quantizer having the input/output relationship shown in Figure 6.3, where $D$ is a decision level that will be calculated shortly. The decision level, $D$, can be calculated according to the formula $D = 0.5 \cdot \sigma = 0.5 \cdot \sqrt{1/(2 \cdot (E_s/N_0))}$, where $Es/No$ is the energy per symbol due to noise density ratio.

![Figure 6.3 Input/Output relationship for an uniform, three-bit quantizer.](image-url)
6.3 Viterbi Decoding

The single most important concept to aid understanding of the Viterbi algorithm is the trellis diagram. Figure 6.4 shows the trellis diagram for our example rate 1/2, $K = 3$ convolutional encoder, for a 15-bit message.

![Trellis diagram for a 15-bit message, generated using a 1/2-rate, $K = 3$ convolutional encoder.](image)

The four possible states of the encoder are depicted as four rows of horizontal dots. There is one column of four dots for the initial state of the encoder and one for each time instant during the message. For a 15-bit message with two encoder memory flushing bits, there are 17 time instants in addition to $t = 0$, which represents the initial condition of the encoder. The solid lines connecting dots in the diagram represent state transitions when the input bit is a one. The dotted lines represent state transitions when the input bit is a zero. Notice the correspondence between the arrows in the trellis diagram and the state transition table discussed above. Also notice that since the initial condition of the encoder is State 00, and the two memory flushing bits are zeros, the arrows start out at State 00 and end up at the same state.

Figure 6.5 shows the states of the trellis that are actually reached during the encoding of the example 15-bit message.

![States of the trellis that are actually reached during the encoding of the 15-bit message.](image)

The encoder input bits and output symbols are shown at the bottom of the diagram. Notice the correspondence between the encoder output symbols and the output table discussed.
above. Let us look at that in more detail, using the expanded version of the transition between one time instant to the next shown in Figure 6.6.

\[ \text{Figure 6.6 Expanded version of the transition between one time instant to the next.} \]

The two-bit numbers labeling the lines are the corresponding convolutional encoder channel symbol outputs. Remember that dotted lines represent cases where the encoder input is a zero, and solid lines represent cases where the encoder input is a one. (In Figure 6.6, the two-bit binary numbers labeling dotted lines are on the left, and the two-bit binary numbers labeling solid lines are on the right.)

Now it is time to see how Viterbi decoding works. For this example, we are going to use hard-decision symbol inputs to keep things simple. Suppose we receive the above encoded message with two bit errors, as shown in Figure 6.7:

\[ \text{Figure 6.7 Trellis tree showing the received message with two errors, in } t = 2 \text{ and } t = 11. \]

Each time we receive a pair of channel symbols, we are going to compute a metric to measure the ‘distance’ between what we received and all of the possible channel symbol pairs we could have received. Going from \( t = 0 \) to \( t = 1 \), there are only two possible channel symbol pairs we could have received: 00, and 11. That is because we know that the convolutional encoder was initialized to the all-zeros state, and given one input bit = one or zero, there are only two states we could transition to and two possible outputs of the encoder. These possible outputs of the encoder are 00 and 11.
The metric we are going to use for now is the Hamming distance between the received channel symbol pair and the possible channel symbol pairs. The Hamming distance is computed by simply counting how many bits are different between the received channel symbol pair and the possible channel symbol pairs. The results can only be zero, one, or two. The Hamming distance (or other metric) values that are computed at each time instant for the paths between the states at the previous time instant and the states at the current time instant are called branch metrics. For the first time instant, we are going to save these results as 'accumulated error metric’ values, associated with states. For the second time instant on, the accumulated error metric will be computed by adding the previous accumulated error metric to the current branch metric.

At \( t = 1 \), we received 00. The only possible channel symbol pairs we could have received are 00 and 11. The Hamming distance between 00 and 00 is zero. The Hamming distance between 00 and 11 is two. Therefore, the branch metric value for the branch from State 00 to State 00 is zero, and for the branch from State 00 to State 10 it is two. Since the previous accumulated error metric values are equal to zero, the accumulated metric values for State 00 and for State 10 are equal to the branch metric values. The accumulated error metric values for the other two states are undefined. Figure 6.8 illustrates the results at \( t = 1 \):

![Figure 6.8 Accumulated Error Metric at \( t = 1 \).](image)

Now let us look what happens at \( t = 2 \). We received a 11 channel symbol pair. The possible channel symbol pairs we could have received in going from \( t = 1 \) to \( t = 2 \) are 00 going from State 00 to State 00, 11 going from State 00 to State 10, 10 going from State 10 to State 01, and 01 going from State 10 to State 11. The Hamming distance between 00 and 11 is two, between 11 and 11 is zero, and between 10 or 01 and 11 is one. We add these branch metric values to the previous accumulated error metric values associated with each state that we came from to get to the current states. At \( t = 1 \), we could only be at State 00 or State 10.
The accumulated error metric values associated with those states were 0 and 2 respectively. Figure 6.9 shows the calculation of the accumulated error metric associated with each state, at \( t = 2 \).

Figure 6.10 shows the situation for \( t = 3 \). Things get a bit more complicated here, since there are now two different ways that we could get from each of the four states that were valid at \( t = 2 \) to the four states that are valid at \( t = 3 \). We compare the accumulated error metrics associated with each branch, and discard the larger one of each pair of branches leading to a given state. If the members of a pair of accumulated error metrics going into a particular state are equal, we just save that value. The other thing that is affected is the predecessor-successor history we are keeping. For each state, the predecessor that survives is the one with the lower branch metric. The operation of adding the previous accumulated error metrics to the new branch metrics, comparing the results, and selecting the smallest accumulated error metric to be retained for the next time instant is called the add-compare-select operation.

At \( t = 17 \), the trellis looks like in Figure 6.11, with the clutter of the intermediate state history removed.
The decoding process begins with building the accumulated error metric for some number of received channel symbol pairs, and the history of what states preceded the states at each time instant $t$ with the smallest accumulated error metric. Once this information is built up, the Viterbi decoder is ready to recreate the sequence of bits that were input to the convolutional encoder when the message was encoded for transmission. This is accomplished by the following steps:

- First, select the state having the smallest accumulated error metric and save the state number of that state.
- Iteratively perform the following step until the beginning of the trellis is reached: Working backward through the state history table, for the selected state, select a new state which is listed in the state history table as being the predecessor to that state. Save the state number of each selected state. This step is called traceback.
- Now work forward through the list of selected states saved in the previous steps. Look up what input bit corresponds to a transition from each predecessor state to its successor state. That is the bit that was most likely encoded by the convolutional encoder.

![Final aspect of the trellis tree, at $t = 17$.](image)

Table 6-3 shows the accumulated metric for the full 15-bit (plus two flushing bits) example message at each time $t$. It is interesting to note that for this hard-decision-input Viterbi decoder example, the smallest accumulated error metric in the final state indicates how many channel symbol errors occurred. Table 6-4 shows the surviving predecessor states for each state at each time $t$. Table 6-5 shows the states selected when tracing the path back through Table 6-4. Using a table that maps state transitions to the inputs that caused them, we can now recreate the original message. Table 6-6 is what this table looks like for our example rate $1/2, K = 3$ convolutional code.
Separating the MPEG-2 Bitstream for Lossless Compression and Transmission

Table 6-3 Accumulated metric for the full 15-bit (plus two flushing bits) example message at each time t.

Table 6-4 Surviving predecessor states for each state at each time t.

Table 6-5 States selected when tracing the path back through Table 14-4.

Table 6-6 Mapping of the state transitions to the inputs that caused them, recreating the original message. x denotes an impossible transition from one state to another state.
Now we have all the tools required to recreate the original message from the message we received, shown in Table 6-7. The two flushing bits are discarded.

<table>
<thead>
<tr>
<th>t</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6-7 Recreation of the original message from the received message.

Figure 6.12 shows a plot of the Bit Error Rate (BER) vs. the SNR, for different values of K (trellis depth), and for uncoded channel [5].

![Plot of Bit Error Rate (BER) vs. SNR](image)

**Figure 6.12** Simulation results for rate = 1/2 convolutional coding with Viterbi decoding on an AWGN channel with various convolutional code constraint lengths.
6.4 Lossless Compressed Bitstream Channel Coding

The purpose of this section is to describe how effectively we apply the Convolutional Coding to the different files (DI, CSI and CAI) that we want to transmit through the noisy channel. We will consider that these files have different importance, in the sense that we will use different encoder length to convolutional encode them. In the same way that we generated a compressed MPEG-2 file using the Formating Program, for storage purposes, now we are going to generate a channel-encoded, compressed MPEG-2 file for transmission purposes. The scheme is shown in Figure 6.13, and it is more complex than that shown in the last Chapter.

![Diagram of the scheme of how to generate a convolutional encoded MPEG-2 bitstream.](image)

Figure 6.13 Scheme of how to generate a convolutional encoded MPEG-2 bitstream.

The input to the channel coding system are the DI files, and the lossless compressed SI and AI files (CSI:s and CAI). The Formating Program (FP) puts together the CSI:s and CAI files, and generates separately a header file (HD1) containing an array of five long integer elements, which store the size of each one of the five compressed files. Then, the size of HD1 is 5x4 bytes = 20 bytes. We do the same with the DI files: they are put together with the help of FP, and another header file (HD2) is generated, containing an array of two long integer elements, storing each one the size of each DI files. HD2 has a size of 2x4 bytes = 8 bytes.

The next step is to convolutional encode the CSI+CAI file, using 'Strong Convolutional Encoding' (SCE in Figure 6.13), that is, a convolutional coder with a high value for the $K$
parameter, because we want to strongly protect the information contained in this file against error introduced by the channel. We can see from Figure 6.12 that \( K = 9 \) is a suitable value. We get the CCSI+Ccai file. The coder rate can be set to 1/2, but there are other suitable values, like 1/3 or 2/3. In the same way, the DI file is convolutional encoded, using 'Weak Convolutional Encoding' (WCE in Figure 6.13), that is, a convolutional coder with a low value for the \( K \) parameter, because we can tolerate higher bit error probability in this file. We can even remove the WCE block and assume that the DI files are not channel-encoded.

Now we have both type of files (CDI and CCSI+Ccai), and the next step is to put them together, using again the FP, generating a third header file (HD3), containing an array of two long integer elements, storing each one the size of each file. HD3 has a size of 2x4 bytes = 8 bytes. We generate then the CCSI+Ccai+CDI file.

The three header files (HD1, HD2 and HD3) are put together. Now we do not need to use the FP, because they have well defined size (20, 8, 8 bytes, respectively). Then, the HD:s file has a size of 36 bytes. Because this information is very important (it is the key to do the reciprocal operations to those which were done in this scheme, after transmission), we are going to channel encode this HD:s file using again 'Strong Convolutional Encoding'. The result is the CHD:s file, with size equal to \( \frac{n}{k} \times 36 \) bytes, where \( \frac{n}{k} \) is the inverse of the coder rate, see Section 6.1. For example, if the coder rate is equal to 1/2, then the CHD:s file will have a size of 2x36 bytes = 72 bytes.

The final step is to put together this CHD:s file with the CCSI+Ccai+CDI file, in this order. Again, we do not need the FP program because CHD:s has a well defined size of \( \frac{n}{k} \times 36 \) bytes, and these bytes are the first in the convolutional encoded MPEG-2 bitstream, ready to be transmitted through the noisy channel.
6.5 Recovering the Compressed MPEG-2 Bitstream.

Once we send the channel-encoded compressed bitstream through the noisy channel, we have to recover this information at the decoder side. The scheme for this recovering should perform exactly the inverse operations shown in Figure 6.13. The inverse operation for convolutional coding is, in this case, Viterbi decoding of the corrupted transmitted source of information. Figure 6.14 shows these reciprocal operations.

![Diagram showing how to recover the compressed MPEG-2 bitstream.](image)

The decoder side 'knows' that the first $\frac{n}{k} \times 36$ bytes of the channel-encoded compressed bitstream correspond to CHD:s, and the rest is CCSI+CCAI+CDI. Then, the first step is to decode these $\frac{n}{k} \times 36$ bytes. In order to allow a reconstruction of the (corrupted) compressed bitstream, we are going to assume that the Viterbi decoder is able to detect and correct all the possible errors found in the CHD:s file. From CHD:s we get again the HD:s file (size=36 bytes), and from this file, we get easily HD1, HD2 and HD3, because we know that the first one has a size of 20 bytes, the second has a size of 8 bytes, and the third one has a size of 8 bytes.
6. Convolutional Coding and Viterbi Decoding

Now we use the HD3 file to split the CCSI+CCAI+CDI file into CCSI+CCAI and CDI, with the help of the Inverse Formating Program, IFP. Then, we perform Viterbi decoding of both files, getting the DI and CSI+CAI files, or a corrupted version of them.

Finally, we use HD2 to split the DI file into the DCT DI and MV DI files, using the IFP, and we use HD1 to recover the HL CSI, SLC CSI, MB CSI, MV CSI and CAI files, arriving to the same situation that we had before channel-encode the compressed bitstream, unless errors are placed in this recovered files, because the Viterbi decoder was not able to detect (or to detect and not correct) errors introduced by the noisy channel.

In the next Chapter several problems related to errors introduced by the noisy channel in the transmitted bitstream are described. Possible solutions to these problems are also shown.
7. Reconstructing Program

This Chapter is the reciprocal of Chapter 4, where the splitting of the MPEG-2 bitstream was described. The first Section of this Chapter assumes that the files used to reconstruct the original bitstream (SI:s files, DI:s files and AI file) are error free (storage purposes). The next Section will deal with problems related to errors in those files due to transmission through a noisy channel.

7.1 Error-free Reconstructing Files

The starting point in order to reconstruct the original MPEG-2 bitstream are the seven files involved in this task, as it is shown in Figure 7.1. They are the same files generated by the Splitting Program, as it is shown in Figure 4.1.

![Scheme of the Merging Program](image)

**Keywords:** HL=High Level; SLC=Slice; MB=Macroblock;
MV=Motion Vectors; DCT=Discrete Cosine Transform (coeff.)

The bitstream merging (or reconstructing) program is called ‘decod’, and a normal call to this program is

```
decod bitstream.rec
```

The file `bitstream.rec` will store the reconstructed bitstream. The merging program (MP) is reciprocal with respect to the splitting program (SP). They both have the same structure, but they do inverse operations. While the SP has a fixed input file (the MPEG-2 bitstream) to be checked, and seven files to fill in (four SI files, two DI files and the AI file), the MP has these seven files as fixed inputs, and an output file to fill in (the `bitstream.rec` file), according to the MPEG-2 syntax.
The key to reverse the operations in the MP is to modify the ‘get_put_bits’ (gpb) function, described in Section 4.2. For example, a typical call to this function should be gpb(ifile5, 3, &pb3), which means that we take the next 3 bits of the input file no.5 (the DCT DI file), and we place them in the output file (bitstream.rec). Again, we assign numbers to the different (input) files in the same way as we did in Section 4.2. In order to show clearly how the MP is a reciprocal version of the SP, we are going to consider the same two consecutive calls that we wrote in Section 4.2. Figure 7.2 shows schematically how these calls are done to the gpb function.

\[ gpb (ifile5, 7, &pb5); \]
\[ \text{-------------------} \]
\[ gpb (ifile1, 4, &pb1); \]

![Figure 7.2](image)

**Figure 7.2** How two consecutive calls to the gpb function fill in the bitstream file.

As a final point to this Section, we are going to describe how the auxiliary information (AI) is used by the reconstructing program. From Section 4.3 we know that there are three sources of AI related to the number of macroblocks per slice, the number of slices per picture, and the stuffing information. We said that the reasons for AI for the two first cases arises from the ‘do-while’ loops, and they will be replaced by ‘for’ loops in the MP as it is shown:

Number of macroblocks per slice.

\[ \text{slice} () \{ \]
\[ \text{slice_start_code;} \]
\[ \text{-------------------} \]
\[ \text{number_mb = number_macroblocks [mb_pos];} \]
\[ \text{mb_pos ++;} \]
\[ \text{for (i=0; i < number_mb; i++) macroblock ();} \]
\[ \text{next_start_code ();} \]
\[ \} \]
7. Reconstructing Program

Number of slices per picture:

```c
picture_data () {

    .................
    number_slc = number_of_slices [pic];
    for (i=0; i < number_slc; i++) slice ();
    pic ++;
    next_start_code ();
}
```

The ‘for’ loops are executed the number of times stored in the ‘counters’ arrays number_macroblocks [], and number_of_slices [].

Regarding to the third type of AI, the next_start_code function will be like this in the reconstructing program:

```c
next_start_code () {

    .................
    number_stf = stuffing [stf_pos];
    gpbi (ifile1, number_stf, &pbi1);
    stf_pos ++;
}
```

That is, the stuffing array tell us the amount of bits (number_stf ) that we have to take from the file no. 1 (HL SI file) each time the next_start_code () is called.

7.2 Non-error-free Reconstructing Files

The starting point of this Section is the Figure 6.14, where we recover the CSI files, the DI files and the CAI file. We are going to assume that the compressed files are error free, that is, the Viterbi decoder was able to detect and correct all the errors found in those files, because of the strong channel-coding applied to them (K parameter of the convolutional encoder equal to a high value). So, we can expand these files to get the original SI and AI files. But the DI files are corrupted, and we are going to have problems to reconstruct the original MPEG-2 bitstream.

When an MPEG-2 decoder is decoding a corrupted MPEG-2 bitstream, and it detects errors (because it is not able to find a match in the variable length code tables) in those parts of the bitstream corresponding to the quantized VLC DCT coefficients, or the Motion Vectors, one simple way of undertaking this situation is to neglect the information between
the error and the ‘next start code’. The decoding process can continue, though part of the bitstream is not displayed, because all the information (signalling and data information) is in the same bitstream file, generated by the coder according to the MPEG-2 syntax.

We can apply a similar technique to the reconstructing program, but we have to remember that our situation is different, because now the bitstream is splitted in six different files (four SI files, and two DI files). If we detect one error in the VLC DCT DI file, the MP is not able to know which file contains the ‘next start code’, that is, which file is the next to check. The same problem arises with errors found in the MV DI file. There are two possible solutions to this problem:

- The first solution is an attempt to solve the problem described in the last paragraph. It consists of copying in to the DCT DI file the ‘next start code’ that the splitting program (SP) finds after checking the blocks of the last macroblock of one slice. We can find four types of ‘next start code’, depending on where the SP is checking inside the bitstream: it can be either ‘slice start code’ (if it is checking the last macroblock of one slice of one picture), or ‘picture start code’ (if it is checking the last macroblock of the last slice of one picture), or ‘group start code’ (if it is checking the last macroblock of the last slice of the last picture of one group of pictures), or ‘sequence end code’ (if it is checking the last macroblock of the last slice of the last picture of the last group of pictures). We do something similar with the motion vectors (MV DI) file. Then, when the merging program (MP) is reconstructing the bitstream, if it detects an error in the DCT DI file (that is, it does not find a match in the variable length code tables) it takes all the information between the error and the ‘next start code’ (neglecting the 32 bits corresponding to that ‘next start code’), and places it in the reconstructed bitstream file. We have to do the same with the information contained in the MV DI file, until the ‘next start code’ is found. Depending on the value of this ‘next start code’, the MP is able to know which file is the next to be checked, either HL SI (which contains ‘picture start code’, ‘group start code’ and ‘sequence end code’, among other information), or SLC SI (which contains ‘slice start code’, among other information). An important point is that the ‘for’ loop associated with the number of macroblocks per slice has to be executed until the end (in order to take the right information from the MB SI file, once the MP is checking the next slice to the erroneous slice), but in these executions of the macroblock () function we do not take any number of bits from neither the MV DI file, nor the DCT DI file, because it was already taken, once the error was detected. So, the MP can continue reconstructing a corrupted version of the original MPEG-2 bitstream, and it is clear that an MPEG-2 decoder will not be
able to decode the information between the error and the end of the slice where the error was found.

If the MP does not find any error when checking the DI files, it simply neglects the 32 bits corresponding to the ‘next start code’ that it will find after the blocks of the last macroblock of one slice.

The main drawback of this solution is that the new version of the DI files (containing ‘next start codes’ in proper places) can have erroneous bits not only in the variable length codes, but also in those ‘next start codes’. Then, the MP will not be able to detect the right ‘next start code’, and it can not continue reconstructing the bitstream.

A minor drawback is that this new version of the DI files are greater than the ‘real’ DI files. Concretely, they will have $32 \times \text{Total\_Number\_of\_Slices}$ bits more than the ‘real’ DI files, degrading the compression ratio that we can reach.

- The second solution is to consider a new type of auxiliary information generated by the splitting program (SP), and stored in a different file (AI2). This file contains two arrays of integers: one array related to the VLC DCT DI file, and another array related to the MV DI file. The elements of these arrays are the number of bits that the SP takes each time it is checking the block layer (in the case of the first array), and the number of bits that the SP takes each time it is checking the motion vectors (in the case of the second array). The length of these arrays will be equal to the number of macroblocks in the bitstream, because each macroblock will have associated one element of the first array, and one element of the second array. For example, a fragment of these arrays can be like this:

```
1\(^{st}\) array: no. bits for the VLC DCT {..., 24, 65, 51, 89, ...}
2\(^{nd}\) array: no. bits for the MV {..., 8, 0, 6, 12, ...}
```

This means that macroblock \(n-1\) contains 24 bits of block information (that is, quantized VLC DCT coefficients), and 8 bits of motion vectors information. And macroblock \(n\) contains 65 bits of block information and 0 bits of motion vectors information (there is no motion compensation).

We can losslessly compress the AI2 file (generating CAI2), and convolutionally encode it, with \(K\) parameter equal to 9, in order to recover the original file free of errors, after Viterbi decoding. Then, the merging program does not have to check the corrupted DI files to
continue with the reconstruction of the bitstream. It only has to check the value of the next element in those arrays stored in AI2, and take the corresponding number of bits from the DI files. Thus, it is possible to reconstruct a corrupted version of the original MPEG-2 bitstream. If the MP finds in the next element of the second array the value ‘0’ (there is no motion compensation), it means that it does not have to take any bit this time.

The main drawback of this solution is that we increase the amount of AI, degrading the compression ratio that we can reach, that is, increasing the ratio $\frac{CSI + DI + CAI}{SI + DI}$. We can even have expansion of the original bitstream in the worst cases (Huffman compression, high bit rate).
8. Conclusions

This Chapter is a summary of the most relevant results and conclusions that were presented in the previous Chapters.

- We have seen that it is worth to implement a program to split an MPEG-2 bitstream file in different signalling and data information files (SI and DI), in order to know how the structure of the bitstream is. This information is presented in the form of ratios: the ratio of the size of the different SI and DI files with respect to the size of the original bitstream.

The generated bitstreams belong to these two types of experiments:
1. Fixed number of frames (equal to 30) and variable bit rate (from 0.1 to 1.0 Mbit/s).
2. Variable number of frames (from 60 to 300) and fixed bit rate (equal to 0.5 Mbit/s).

An auxiliary information (AI) file (related to the number of slices per picture, the number of macroblocks per slice, and the stuffing information) is generated by the splitting program (SP) in order to allow the merging program (MP) to reconstruct the original bitstream.

- Regarding lossless compression, we have seen that increasing the bit rate, the compression ratio (CR) decreases for both Huffman coding and Lempel Ziv Welch (LZW) coding. And increasing the number of frames, the CR increases for both lossless compression methods. But we always had compression, even in the worst cases (Huffman coding, high bit rate). LZW coding was better than Huffman coding in all the experiments. The macroblock signalling information (MB SI) file was the most limiting in order to reach a high CR, because of the low compression, compared with the size (high MB SI size with respect to the bitstream size).

A possible format for the compressed bitstream was shown, and how to losslessly compress an MPEG-2 bitstream, which is suitable for storage purposes.

- We have seen how it is possible to transmit the compressed bitstream through a noisy channel, introducing proper channel coding, concretely convolutional coding, to the compressed SI and AI files (CSI and CAI), and the DI files. The most important files (CSI and CAI) can be channel coded with a $K$ parameter (depth of the convolutional coder) equal to a high value ($Strong\ Convolutional\ Encoding$), and the DI files can be channel coded with $K$ parameter equal to a low value ($Weak\ Convolutional\ Encoding$), or even they can stay channel uncoded. These transmitted files can be decoded using a Viterbi decoder, and we assumed that the CSI and CAI files could be recovered with no errors, in order to expand them and get the original SI and AI files. We tolerated errors in the DI files, and possible
problems to reconstruct the original bitstream (or even a corrupted version of it) were shown. Two different solutions to these reconstructing problems were described.
In this Chapter, several improvements and suggestions for the future are presented, in order to complete the work of this thesis.

- MPEG-2 bitstream with very high bit rate and number of frames can be generated, to check whether it is worth to losslessly compress associated signalling information in real world applications, such as High Definition TV (HDTV), which uses bit rates around 18 Mbit/s [1].

- Different sizes for the group of pictures (GOP) layer, that is, different number of pictures per GOP can be introduced, to see how the bitstream structure and the compression ratio change. Generation of bitstreams without B pictures can also be interesting.

- A third type of experiment can be done as a combination of those two types presented in this thesis. The number of frames and the bit rate should then be varied to obtain a three dimensional plot, where we represent the bit rate on the x axis, the number of frames on the y axis, and the compression ratio (CR) on the z axis, as it is shown in Figure 9.1

![Figure 9.1 3D plot to represent the CR 'surface', depending on the bit rate and the number of frames in the bitstream.](image)

- Another improvement is related to the way in which we split a bitstream. We have seen that the most critical file when trying to get a high compression ratio is the one related to the signalling information of the macroblock layer (MB SI). It should be interesting to find out new ways of splitting the bitstream, in order to efficiently exploit the redundancy contained in the signalling information found in the bitstreams.
A final suggestion is to modify directly the MPEG-2 standard [2], in order to generate directly compressed bitstreams. We do not leave the idea of bitstream, but we consider a new structure for it, with three clearly different parts: the compressed signalling information (CSI), the compressed auxiliary information (CAI), and the data information (DI).


Appendix A: Tables

References


[2] Information Technology. Generic Coding of Moving Pictures and Associated Audio
    Information: Video
    ITU-T Recommendation H.262, July 1995

    MPEG-2 Software Simulation Group, June 1994

    Wolfram Keck, IEEE Transactions on Consumer Electronics, August 1996

www references

    Chip Fleming, Spectrum Applications, July 1999
    http://pweb.netcom.com/~chip.f/viterbi/tutorial.html

[6] Huffman Compression
    John Koss
    http://www.geocities.com/SiliconValley/2151/huffman.html

[7] LZW Data Compression
    Mark Nelson, Dr. Dobb's Journal, October 1989
    http://www.dogma.net/markn/articles/lzw/lzw.htm
Appendix A: Tables

Appendix A: Tables

This Appendix contains the tables from which the plots shown in Chapters 4 and 5 were made. These tables also include additional information that we found when the experiments were done.

A.1 MPEG-2 Bitstream Structure

In this Section, the percentages (%) shown in the different tables are related to the whole size of the bitstream file (e.g. SLC SI (%) is the ratio between the size of the Slice Signalling Information and the size of the bitstream). AI/Bitstr. (%) is the ratio between the Auxiliary Information file and the bitstream file.

Fixed number of frames, variable bit rate

Number of frames: 30     Bit Rate: 0.1-1.0 Mbit/s

File Name: mad.yuv.

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Bitstream Size (Bytes)</th>
<th>PSNR (dB)</th>
<th>HL SI (%)</th>
<th>SLC SI (%)</th>
<th>MB SI (%)</th>
<th>MV SI (%)</th>
<th>TOTAL SI (%)</th>
<th>DCT DI (%)</th>
<th>MV DI (%)</th>
<th>AI/Bitstr. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>42781</td>
<td>31.7</td>
<td>2.02</td>
<td>6.00</td>
<td>17.57</td>
<td>5.41</td>
<td>31.00</td>
<td>24.66</td>
<td>44.35</td>
<td>2.84</td>
</tr>
<tr>
<td>0.2</td>
<td>44040</td>
<td>31.7</td>
<td>2.00</td>
<td>5.82</td>
<td>17.93</td>
<td>5.17</td>
<td>30.93</td>
<td>26.69</td>
<td>42.39</td>
<td>2.76</td>
</tr>
<tr>
<td>0.3</td>
<td>45623</td>
<td>31.9</td>
<td>1.91</td>
<td>5.62</td>
<td>18.10</td>
<td>4.87</td>
<td>30.51</td>
<td>29.43</td>
<td>40.07</td>
<td>2.67</td>
</tr>
<tr>
<td>0.4</td>
<td>49149</td>
<td>33.3</td>
<td>1.79</td>
<td>5.22</td>
<td>18.10</td>
<td>4.49</td>
<td>29.60</td>
<td>33.45</td>
<td>36.95</td>
<td>2.48</td>
</tr>
<tr>
<td>0.5</td>
<td>62629</td>
<td>37.5</td>
<td>1.40</td>
<td>4.10</td>
<td>17.66</td>
<td>3.38</td>
<td>26.53</td>
<td>45.97</td>
<td>27.51</td>
<td>1.94</td>
</tr>
<tr>
<td>0.6</td>
<td>74926</td>
<td>38.9</td>
<td>1.15</td>
<td>3.42</td>
<td>16.08</td>
<td>2.77</td>
<td>23.42</td>
<td>53.98</td>
<td>22.60</td>
<td>1.62</td>
</tr>
<tr>
<td>0.7</td>
<td>87473</td>
<td>39.7</td>
<td>1.01</td>
<td>2.93</td>
<td>15.00</td>
<td>2.36</td>
<td>21.31</td>
<td>59.52</td>
<td>19.18</td>
<td>1.39</td>
</tr>
<tr>
<td>0.8</td>
<td>99964</td>
<td>40.5</td>
<td>0.88</td>
<td>2.57</td>
<td>14.25</td>
<td>2.03</td>
<td>19.72</td>
<td>63.97</td>
<td>16.31</td>
<td>1.22</td>
</tr>
<tr>
<td>0.9</td>
<td>112514</td>
<td>41.1</td>
<td>0.77</td>
<td>2.28</td>
<td>13.64</td>
<td>1.81</td>
<td>18.51</td>
<td>66.98</td>
<td>14.52</td>
<td>1.08</td>
</tr>
<tr>
<td>1.0</td>
<td>124980</td>
<td>41.5</td>
<td>0.70</td>
<td>2.05</td>
<td>13.15</td>
<td>1.63</td>
<td>17.53</td>
<td>69.64</td>
<td>12.83</td>
<td>0.97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>SI (%)</th>
<th>DI (%)</th>
<th>SI (%)</th>
<th>DI (%)</th>
<th>SI (%)</th>
<th>DI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2.01</td>
<td>21.82</td>
<td>5.77</td>
<td>7.42</td>
<td>22.37</td>
<td>39.77</td>
</tr>
<tr>
<td>0.2</td>
<td>2.16</td>
<td>23.15</td>
<td>6.24</td>
<td>8.00</td>
<td>21.66</td>
<td>37.93</td>
</tr>
<tr>
<td>0.3</td>
<td>2.44</td>
<td>24.76</td>
<td>6.43</td>
<td>8.83</td>
<td>20.83</td>
<td>35.90</td>
</tr>
<tr>
<td>0.4</td>
<td>2.50</td>
<td>26.53</td>
<td>7.02</td>
<td>10.81</td>
<td>19.31</td>
<td>33.06</td>
</tr>
<tr>
<td>0.5</td>
<td>1.91</td>
<td>28.51</td>
<td>7.67</td>
<td>17.36</td>
<td>16.35</td>
<td>27.60</td>
</tr>
<tr>
<td>0.6</td>
<td>1.56</td>
<td>29.01</td>
<td>7.10</td>
<td>22.30</td>
<td>14.28</td>
<td>25.27</td>
</tr>
<tr>
<td>0.7</td>
<td>1.30</td>
<td>28.66</td>
<td>6.53</td>
<td>25.49</td>
<td>13.04</td>
<td>24.54</td>
</tr>
<tr>
<td>0.8</td>
<td>1.15</td>
<td>28.05</td>
<td>6.01</td>
<td>28.09</td>
<td>12.18</td>
<td>24.14</td>
</tr>
<tr>
<td>0.9</td>
<td>1.03</td>
<td>27.29</td>
<td>5.50</td>
<td>29.55</td>
<td>11.65</td>
<td>24.66</td>
</tr>
<tr>
<td>1.0</td>
<td>0.94</td>
<td>26.74</td>
<td>5.11</td>
<td>30.87</td>
<td>11.17</td>
<td>24.87</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>SI (%)</th>
<th>DI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2.01</td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td>2.16</td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td>2.44</td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td>2.50</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>1.91</td>
<td></td>
</tr>
<tr>
<td>0.6</td>
<td>1.56</td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>1.30</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.15</td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td>1.03</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>0.94</td>
<td></td>
</tr>
</tbody>
</table>

89
### Separating the MPEG-2 Bitstream for Lossless Compression and Transmission

**File Name:** `akiyo.yuv`  
**File Name:** `contain.yuv`  

#### Table 1: I Pictures, P Pictures, B Pictures for akiyo.yuv

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Bitstream Size (Bytes)</th>
<th>PSNR (dB)</th>
<th>HL SI (%)</th>
<th>SLC SI (%)</th>
<th>MB SI (%)</th>
<th>MV SI (%)</th>
<th>TOTAL SI (%)</th>
<th>DCT DI (%)</th>
<th>MV DI (%)</th>
<th>AI/Bitstr (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>37513</td>
<td>30.4</td>
<td>2.28</td>
<td>6.84</td>
<td>19.65</td>
<td>5.96</td>
<td>34.73</td>
<td>28.81</td>
<td>36.47</td>
<td>3.24</td>
</tr>
<tr>
<td>0.2</td>
<td>38924</td>
<td>30.5</td>
<td>2.25</td>
<td>6.59</td>
<td>19.95</td>
<td>5.61</td>
<td>34.40</td>
<td>31.71</td>
<td>33.89</td>
<td>3.13</td>
</tr>
<tr>
<td>0.3</td>
<td>41074</td>
<td>31.1</td>
<td>2.10</td>
<td>6.24</td>
<td>20.02</td>
<td>5.24</td>
<td>33.61</td>
<td>35.62</td>
<td>30.78</td>
<td>2.96</td>
</tr>
<tr>
<td>0.4</td>
<td>50417</td>
<td>35.7</td>
<td>1.71</td>
<td>5.09</td>
<td>18.70</td>
<td>4.25</td>
<td>29.75</td>
<td>46.92</td>
<td>23.32</td>
<td>2.41</td>
</tr>
<tr>
<td>0.5</td>
<td>62646</td>
<td>38.1</td>
<td>1.40</td>
<td>4.09</td>
<td>16.32</td>
<td>3.40</td>
<td>25.21</td>
<td>57.20</td>
<td>17.60</td>
<td>1.94</td>
</tr>
<tr>
<td>0.6</td>
<td>75080</td>
<td>39.8</td>
<td>1.16</td>
<td>3.42</td>
<td>14.91</td>
<td>2.86</td>
<td>22.34</td>
<td>63.71</td>
<td>13.95</td>
<td>1.62</td>
</tr>
<tr>
<td>0.7</td>
<td>87447</td>
<td>41.1</td>
<td>1.00</td>
<td>2.93</td>
<td>13.65</td>
<td>2.40</td>
<td>19.98</td>
<td>68.58</td>
<td>11.44</td>
<td>1.39</td>
</tr>
<tr>
<td>0.8</td>
<td>99914</td>
<td>41.9</td>
<td>0.87</td>
<td>2.57</td>
<td>12.91</td>
<td>2.13</td>
<td>18.48</td>
<td>71.57</td>
<td>9.95</td>
<td>1.22</td>
</tr>
<tr>
<td>0.9</td>
<td>112461</td>
<td>42.5</td>
<td>0.77</td>
<td>2.28</td>
<td>12.20</td>
<td>1.90</td>
<td>17.16</td>
<td>74.27</td>
<td>8.58</td>
<td>1.08</td>
</tr>
<tr>
<td>1.0</td>
<td>124923</td>
<td>43.2</td>
<td>0.70</td>
<td>2.05</td>
<td>11.48</td>
<td>1.72</td>
<td>15.96</td>
<td>76.42</td>
<td>7.62</td>
<td>0.97</td>
</tr>
</tbody>
</table>

#### Table 2: I Pictures, P Pictures, B Pictures for contain.yuv

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Bitstream Size (Bytes)</th>
<th>PSNR (dB)</th>
<th>HL SI (%)</th>
<th>SLC SI (%)</th>
<th>MB SI (%)</th>
<th>MV SI (%)</th>
<th>TOTAL SI (%)</th>
<th>DCT DI (%)</th>
<th>MV DI (%)</th>
<th>AI/Bitstr (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>41363</td>
<td>26.1</td>
<td>2.11</td>
<td>6.20</td>
<td>17.90</td>
<td>5.12</td>
<td>31.33</td>
<td>33.99</td>
<td>34.68</td>
<td>2.94</td>
</tr>
<tr>
<td>0.2</td>
<td>42902</td>
<td>26.2</td>
<td>2.03</td>
<td>5.98</td>
<td>18.00</td>
<td>4.98</td>
<td>30.99</td>
<td>36.13</td>
<td>32.89</td>
<td>2.84</td>
</tr>
<tr>
<td>0.3</td>
<td>45016</td>
<td>26.4</td>
<td>1.92</td>
<td>5.70</td>
<td>17.90</td>
<td>4.80</td>
<td>30.31</td>
<td>39.13</td>
<td>30.56</td>
<td>2.70</td>
</tr>
<tr>
<td>0.4</td>
<td>49385</td>
<td>27.7</td>
<td>1.78</td>
<td>5.19</td>
<td>17.32</td>
<td>4.47</td>
<td>28.76</td>
<td>44.47</td>
<td>26.77</td>
<td>2.46</td>
</tr>
<tr>
<td>0.5</td>
<td>62781</td>
<td>30.3</td>
<td>1.39</td>
<td>4.09</td>
<td>15.89</td>
<td>3.78</td>
<td>25.13</td>
<td>54.40</td>
<td>20.47</td>
<td>1.94</td>
</tr>
<tr>
<td>0.6</td>
<td>75240</td>
<td>31.9</td>
<td>1.16</td>
<td>3.41</td>
<td>13.65</td>
<td>3.29</td>
<td>21.52</td>
<td>62.59</td>
<td>15.89</td>
<td>1.62</td>
</tr>
<tr>
<td>0.7</td>
<td>87511</td>
<td>32.9</td>
<td>1.00</td>
<td>2.93</td>
<td>12.39</td>
<td>2.92</td>
<td>19.24</td>
<td>67.41</td>
<td>13.35</td>
<td>1.39</td>
</tr>
<tr>
<td>0.8</td>
<td>99976</td>
<td>34.0</td>
<td>0.88</td>
<td>2.57</td>
<td>11.36</td>
<td>2.62</td>
<td>17.43</td>
<td>71.11</td>
<td>11.47</td>
<td>1.22</td>
</tr>
<tr>
<td>0.9</td>
<td>112312</td>
<td>34.7</td>
<td>0.78</td>
<td>2.28</td>
<td>10.44</td>
<td>2.36</td>
<td>15.86</td>
<td>74.10</td>
<td>10.04</td>
<td>1.08</td>
</tr>
<tr>
<td>1.0</td>
<td>124852</td>
<td>35.6</td>
<td>0.70</td>
<td>2.05</td>
<td>9.76</td>
<td>2.16</td>
<td>14.67</td>
<td>76.41</td>
<td>8.92</td>
<td>0.97</td>
</tr>
</tbody>
</table>

#### Table 3: I Pictures, P Pictures, B Pictures for akiyo.yuv

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>SI (%)</th>
<th>DI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2.01</td>
<td>26.26</td>
</tr>
<tr>
<td>0.2</td>
<td>2.17</td>
<td>27.35</td>
</tr>
<tr>
<td>0.3</td>
<td>2.35</td>
<td>29.14</td>
</tr>
<tr>
<td>0.4</td>
<td>2.28</td>
<td>31.99</td>
</tr>
<tr>
<td>0.5</td>
<td>1.83</td>
<td>32.49</td>
</tr>
<tr>
<td>0.6</td>
<td>1.47</td>
<td>34.99</td>
</tr>
<tr>
<td>0.7</td>
<td>1.26</td>
<td>34.77</td>
</tr>
<tr>
<td>0.8</td>
<td>1.10</td>
<td>34.25</td>
</tr>
<tr>
<td>0.9</td>
<td>0.97</td>
<td>33.32</td>
</tr>
<tr>
<td>1.0</td>
<td>0.87</td>
<td>32.46</td>
</tr>
</tbody>
</table>

#### Table 4: P Pictures, B Pictures for contain.yuv

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>SI (%)</th>
<th>DI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>4.76</td>
<td>5.37</td>
</tr>
<tr>
<td>0.2</td>
<td>5.12</td>
<td>6.99</td>
</tr>
<tr>
<td>0.3</td>
<td>5.41</td>
<td>8.41</td>
</tr>
<tr>
<td>0.4</td>
<td>5.86</td>
<td>10.86</td>
</tr>
<tr>
<td>0.5</td>
<td>6.44</td>
<td>18.07</td>
</tr>
<tr>
<td>0.6</td>
<td>5.99</td>
<td>23.72</td>
</tr>
<tr>
<td>0.7</td>
<td>5.50</td>
<td>27.54</td>
</tr>
<tr>
<td>0.8</td>
<td>4.94</td>
<td>30.29</td>
</tr>
<tr>
<td>0.9</td>
<td>4.47</td>
<td>32.77</td>
</tr>
<tr>
<td>1.0</td>
<td>4.01</td>
<td>34.28</td>
</tr>
</tbody>
</table>

#### Table 5: B Pictures for contain.yuv

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>SI (%)</th>
<th>DI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>23.67</td>
<td>37.04</td>
</tr>
<tr>
<td>0.2</td>
<td>22.83</td>
<td>34.67</td>
</tr>
<tr>
<td>0.3</td>
<td>21.75</td>
<td>32.14</td>
</tr>
<tr>
<td>0.4</td>
<td>19.87</td>
<td>28.39</td>
</tr>
<tr>
<td>0.5</td>
<td>16.27</td>
<td>24.31</td>
</tr>
<tr>
<td>0.6</td>
<td>13.57</td>
<td>19.77</td>
</tr>
<tr>
<td>0.7</td>
<td>12.04</td>
<td>18.45</td>
</tr>
<tr>
<td>0.8</td>
<td>11.01</td>
<td>18.03</td>
</tr>
<tr>
<td>0.9</td>
<td>10.08</td>
<td>18.05</td>
</tr>
<tr>
<td>1.0</td>
<td>9.50</td>
<td>18.59</td>
</tr>
</tbody>
</table>
### Appendix A: Tables

#### Fixed bit rate, variable number of frames

**Number of frames: 60-300**  
**Bit Rate: 0.5 Mbit/s**

**File Name:** mad.yuv.

<table>
<thead>
<tr>
<th># Frames</th>
<th>Bitstream Size (Bytes)</th>
<th>PSNR (dB)</th>
<th>HL SI (%)</th>
<th>SLC SI (%)</th>
<th>MB SI (%)</th>
<th>MV SI (%)</th>
<th>TOTAL SI (%)</th>
<th>DCT DI (%)</th>
<th>MV DI (%)</th>
<th>AI/Bitstr. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>127526</td>
<td>37.9</td>
<td>1.32</td>
<td>4.02</td>
<td>17.18</td>
<td>3.39</td>
<td>23.91</td>
<td>49.30</td>
<td>24.79</td>
<td>1.90</td>
</tr>
<tr>
<td>120</td>
<td>252349</td>
<td>37.5</td>
<td>1.30</td>
<td>4.07</td>
<td>17.35</td>
<td>3.43</td>
<td>26.13</td>
<td>47.51</td>
<td>26.36</td>
<td>1.91</td>
</tr>
<tr>
<td>180</td>
<td>377219</td>
<td>38.0</td>
<td>1.29</td>
<td>4.08</td>
<td>17.54</td>
<td>3.45</td>
<td>26.35</td>
<td>46.60</td>
<td>27.05</td>
<td>1.92</td>
</tr>
<tr>
<td>240</td>
<td>502562</td>
<td>38.7</td>
<td>1.28</td>
<td>4.08</td>
<td>17.57</td>
<td>3.45</td>
<td>26.39</td>
<td>46.58</td>
<td>27.03</td>
<td>1.92</td>
</tr>
<tr>
<td>300</td>
<td>627569</td>
<td>39.0</td>
<td>1.28</td>
<td>4.08</td>
<td>17.57</td>
<td>3.48</td>
<td>26.42</td>
<td>46.91</td>
<td>26.67</td>
<td>1.92</td>
</tr>
</tbody>
</table>

### I Pictures | P Pictures | B Pictures

<table>
<thead>
<tr>
<th># Frames</th>
<th>SI (%)</th>
<th>DI (%)</th>
<th>SI (%)</th>
<th>DI (%)</th>
<th>SI (%)</th>
<th>DI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>1.94</td>
<td>32.64</td>
<td>7.23</td>
<td>16.19</td>
<td>16.21</td>
<td>25.25</td>
</tr>
<tr>
<td>120</td>
<td>1.81</td>
<td>30.37</td>
<td>7.15</td>
<td>16.50</td>
<td>16.67</td>
<td>26.99</td>
</tr>
<tr>
<td>180</td>
<td>1.77</td>
<td>29.13</td>
<td>7.22</td>
<td>16.86</td>
<td>16.87</td>
<td>27.66</td>
</tr>
<tr>
<td>240</td>
<td>1.75</td>
<td>29.19</td>
<td>7.35</td>
<td>17.00</td>
<td>16.80</td>
<td>27.42</td>
</tr>
<tr>
<td>300</td>
<td>1.74</td>
<td>29.49</td>
<td>7.43</td>
<td>17.07</td>
<td>16.77</td>
<td>27.02</td>
</tr>
</tbody>
</table>

**File Name:** akiyo.yuv.

<table>
<thead>
<tr>
<th># Frames</th>
<th>Bitstream Size (Bytes)</th>
<th>PSNR (dB)</th>
<th>HL SI (%)</th>
<th>SLC SI (%)</th>
<th>MB SI (%)</th>
<th>MV SI (%)</th>
<th>TOTAL SI (%)</th>
<th>DCT DI (%)</th>
<th>MV DI (%)</th>
<th>AI/Bitstr. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>127948</td>
<td>39.2</td>
<td>1.31</td>
<td>4.01</td>
<td>16.24</td>
<td>3.39</td>
<td>24.95</td>
<td>58.34</td>
<td>16.72</td>
<td>1.89</td>
</tr>
<tr>
<td>120</td>
<td>252846</td>
<td>39.1</td>
<td>1.29</td>
<td>4.06</td>
<td>16.45</td>
<td>3.46</td>
<td>25.26</td>
<td>57.54</td>
<td>17.20</td>
<td>1.91</td>
</tr>
<tr>
<td>180</td>
<td>377982</td>
<td>39.4</td>
<td>1.28</td>
<td>4.07</td>
<td>16.47</td>
<td>3.49</td>
<td>25.31</td>
<td>57.66</td>
<td>17.02</td>
<td>1.91</td>
</tr>
<tr>
<td>240</td>
<td>503091</td>
<td>38.1</td>
<td>1.28</td>
<td>4.08</td>
<td>16.45</td>
<td>3.49</td>
<td>25.30</td>
<td>57.78</td>
<td>16.92</td>
<td>1.92</td>
</tr>
<tr>
<td>300</td>
<td>627907</td>
<td>38.8</td>
<td>1.28</td>
<td>4.09</td>
<td>16.48</td>
<td>3.50</td>
<td>25.34</td>
<td>57.72</td>
<td>16.94</td>
<td>1.92</td>
</tr>
</tbody>
</table>

### I Pictures | P Pictures | B Pictures

<table>
<thead>
<tr>
<th># Frames</th>
<th>SI (%)</th>
<th>DI (%)</th>
<th>SI (%)</th>
<th>DI (%)</th>
<th>SI (%)</th>
<th>DI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>1.89</td>
<td>33.36</td>
<td>5.33</td>
<td>15.66</td>
<td>17.21</td>
<td>24.03</td>
</tr>
<tr>
<td>120</td>
<td>1.78</td>
<td>33.97</td>
<td>5.34</td>
<td>16.12</td>
<td>17.65</td>
<td>24.65</td>
</tr>
<tr>
<td>180</td>
<td>1.74</td>
<td>34.04</td>
<td>5.30</td>
<td>15.93</td>
<td>17.79</td>
<td>24.73</td>
</tr>
<tr>
<td>240</td>
<td>1.73</td>
<td>34.33</td>
<td>5.25</td>
<td>15.71</td>
<td>17.84</td>
<td>24.66</td>
</tr>
<tr>
<td>300</td>
<td>1.71</td>
<td>34.11</td>
<td>5.27</td>
<td>15.86</td>
<td>17.89</td>
<td>24.68</td>
</tr>
</tbody>
</table>

**File Name:** containe.yuv.

<table>
<thead>
<tr>
<th># Frames</th>
<th>Bitstream Size (Bytes)</th>
<th>PSNR (dB)</th>
<th>HL SI (%)</th>
<th>SLC SI (%)</th>
<th>MB SI (%)</th>
<th>MV SI (%)</th>
<th>TOTAL SI (%)</th>
<th>DCT DI (%)</th>
<th>MV DI (%)</th>
<th>AI/Bitstr. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>127942</td>
<td>31.2</td>
<td>1.30</td>
<td>4.01</td>
<td>15.39</td>
<td>3.96</td>
<td>24.66</td>
<td>55.24</td>
<td>20.10</td>
<td>1.89</td>
</tr>
<tr>
<td>120</td>
<td>252916</td>
<td>31.1</td>
<td>1.28</td>
<td>4.06</td>
<td>15.48</td>
<td>4.07</td>
<td>24.89</td>
<td>54.63</td>
<td>20.49</td>
<td>1.91</td>
</tr>
<tr>
<td>180</td>
<td>377817</td>
<td>31.2</td>
<td>1.28</td>
<td>4.07</td>
<td>15.50</td>
<td>4.11</td>
<td>24.96</td>
<td>54.60</td>
<td>20.43</td>
<td>1.91</td>
</tr>
<tr>
<td>240</td>
<td>503118</td>
<td>31.4</td>
<td>1.28</td>
<td>4.08</td>
<td>15.51</td>
<td>4.12</td>
<td>24.98</td>
<td>54.66</td>
<td>20.36</td>
<td>1.92</td>
</tr>
<tr>
<td>300</td>
<td>628112</td>
<td>31.5</td>
<td>1.27</td>
<td>4.08</td>
<td>15.50</td>
<td>4.13</td>
<td>24.99</td>
<td>54.60</td>
<td>20.41</td>
<td>1.92</td>
</tr>
</tbody>
</table>

### I Pictures | P Pictures | B Pictures

<table>
<thead>
<tr>
<th># Frames</th>
<th>SI (%)</th>
<th>DI (%)</th>
<th>SI (%)</th>
<th>DI (%)</th>
<th>SI (%)</th>
<th>DI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>1.80</td>
<td>35.43</td>
<td>6.20</td>
<td>18.50</td>
<td>16.15</td>
<td>21.41</td>
</tr>
<tr>
<td>120</td>
<td>1.65</td>
<td>34.92</td>
<td>6.36</td>
<td>18.76</td>
<td>16.39</td>
<td>21.44</td>
</tr>
<tr>
<td>180</td>
<td>1.61</td>
<td>34.82</td>
<td>6.41</td>
<td>18.93</td>
<td>16.47</td>
<td>21.28</td>
</tr>
<tr>
<td>240</td>
<td>1.58</td>
<td>34.82</td>
<td>6.45</td>
<td>19.07</td>
<td>16.48</td>
<td>21.13</td>
</tr>
<tr>
<td>300</td>
<td>1.57</td>
<td>34.51</td>
<td>6.43</td>
<td>19.12</td>
<td>16.51</td>
<td>21.39</td>
</tr>
</tbody>
</table>
A.2 Lossless Compression Ratios

In this Section, the different Compression Ratios (CR:s) that we obtained with Huffman coding and Lempel Ziv Welch coding are shown. The percentages correspond to this formula:

\[
CR (\%) = \left(1 - \frac{CS}{OS}\right) \times 100,
\]

where \(OS\) is the Original Size of the file, and \(CS\) is the Compressed Size of the file (or files).

**Fixed number of frames, variable bit rate**

Number of frames: 30  \hspace{1cm} Bit Rate: 0.1-1.0 Mbit/s

File Name: \textit{mad.yuv}.

**Lossless Compression Scheme: Huffman Coding**

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>40663</td>
<td>5.0</td>
<td>0</td>
<td>48.8</td>
<td>19.3</td>
<td>0</td>
<td>52.4</td>
</tr>
<tr>
<td>0.2</td>
<td>42178</td>
<td>4.2</td>
<td>0</td>
<td>46.2</td>
<td>16.1</td>
<td>0</td>
<td>51.4</td>
</tr>
<tr>
<td>0.3</td>
<td>43943</td>
<td>3.7</td>
<td>0</td>
<td>43.5</td>
<td>14.5</td>
<td>0</td>
<td>51.2</td>
</tr>
<tr>
<td>0.4</td>
<td>47668</td>
<td>3.0</td>
<td>0</td>
<td>42.3</td>
<td>11.2</td>
<td>0</td>
<td>50.6</td>
</tr>
<tr>
<td>0.5</td>
<td>62083</td>
<td>0.9</td>
<td>0</td>
<td>45.0</td>
<td>0</td>
<td>0</td>
<td>50.3</td>
</tr>
<tr>
<td>0.6</td>
<td>74349</td>
<td>0.8</td>
<td>0</td>
<td>46.0</td>
<td>0</td>
<td>0</td>
<td>50.8</td>
</tr>
<tr>
<td>0.7</td>
<td>86851</td>
<td>0.7</td>
<td>0</td>
<td>47.1</td>
<td>0</td>
<td>0</td>
<td>52.0</td>
</tr>
<tr>
<td>0.8</td>
<td>99308</td>
<td>0.7</td>
<td>0</td>
<td>48.1</td>
<td>0</td>
<td>0</td>
<td>52.7</td>
</tr>
<tr>
<td>0.9</td>
<td>111854</td>
<td>0.6</td>
<td>0</td>
<td>47.3</td>
<td>0</td>
<td>0</td>
<td>54.7</td>
</tr>
<tr>
<td>1.0</td>
<td>123973</td>
<td>0.8</td>
<td>0</td>
<td>47.6</td>
<td>2.0</td>
<td>0</td>
<td>56.0</td>
</tr>
</tbody>
</table>

**Lossless Compression Scheme: Lempel Ziv Welch Coding**

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>38289</td>
<td>10.5</td>
<td>54.3</td>
<td>67.6</td>
<td>31.5</td>
<td>0</td>
<td>62.2</td>
</tr>
<tr>
<td>0.2</td>
<td>39974</td>
<td>9.2</td>
<td>56.8</td>
<td>79.7</td>
<td>27.1</td>
<td>0.1</td>
<td>60.2</td>
</tr>
<tr>
<td>0.3</td>
<td>41937</td>
<td>8.1</td>
<td>56.0</td>
<td>72.3</td>
<td>23.8</td>
<td>0.2</td>
<td>59.4</td>
</tr>
<tr>
<td>0.4</td>
<td>45965</td>
<td>6.5</td>
<td>57.6</td>
<td>60.7</td>
<td>19.5</td>
<td>0.8</td>
<td>59.0</td>
</tr>
<tr>
<td>0.5</td>
<td>60013</td>
<td>4.2</td>
<td>53.9</td>
<td>61.9</td>
<td>10.3</td>
<td>2.5</td>
<td>59.0</td>
</tr>
<tr>
<td>0.6</td>
<td>72395</td>
<td>3.4</td>
<td>55.2</td>
<td>64.9</td>
<td>7.8</td>
<td>4.3</td>
<td>57.8</td>
</tr>
<tr>
<td>0.7</td>
<td>84900</td>
<td>2.9</td>
<td>56.9</td>
<td>68.3</td>
<td>6.2</td>
<td>6.9</td>
<td>58.9</td>
</tr>
<tr>
<td>0.8</td>
<td>97469</td>
<td>2.5</td>
<td>54.7</td>
<td>68.8</td>
<td>5.0</td>
<td>8.5</td>
<td>58.9</td>
</tr>
<tr>
<td>0.9</td>
<td>110013</td>
<td>2.2</td>
<td>53.9</td>
<td>69.8</td>
<td>4.2</td>
<td>9.4</td>
<td>61.5</td>
</tr>
<tr>
<td>1.0</td>
<td>122391</td>
<td>2.1</td>
<td>56.9</td>
<td>71.3</td>
<td>3.9</td>
<td>10.2</td>
<td>62.6</td>
</tr>
</tbody>
</table>
### File Name: akiyo.yuv.

**Lossless Compression Scheme: Huffman Coding**

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>35080</td>
<td>6.5</td>
<td>0</td>
<td>48.0</td>
<td>23.9</td>
<td>0</td>
<td>54.2</td>
</tr>
<tr>
<td>0.2</td>
<td>36703</td>
<td>5.7</td>
<td>0</td>
<td>45.5</td>
<td>20.8</td>
<td>0</td>
<td>54.1</td>
</tr>
<tr>
<td>0.3</td>
<td>39062</td>
<td>4.9</td>
<td>0</td>
<td>43.1</td>
<td>17.9</td>
<td>0</td>
<td>53.7</td>
</tr>
<tr>
<td>0.4</td>
<td>48671</td>
<td>3.5</td>
<td>0</td>
<td>44.1</td>
<td>12.6</td>
<td>0</td>
<td>52.8</td>
</tr>
<tr>
<td>0.5</td>
<td>60953</td>
<td>2.7</td>
<td>0</td>
<td>46.9</td>
<td>10.7</td>
<td>0</td>
<td>50.9</td>
</tr>
<tr>
<td>0.6</td>
<td>74503</td>
<td>0.8</td>
<td>0</td>
<td>46.2</td>
<td>0</td>
<td>0</td>
<td>50.4</td>
</tr>
<tr>
<td>0.7</td>
<td>86104</td>
<td>1.5</td>
<td>0</td>
<td>47.3</td>
<td>6.2</td>
<td>0</td>
<td>50.0</td>
</tr>
<tr>
<td>0.8</td>
<td>98582</td>
<td>1.3</td>
<td>0</td>
<td>47.4</td>
<td>4.7</td>
<td>5.3</td>
<td>50.4</td>
</tr>
<tr>
<td>0.9</td>
<td>111125</td>
<td>1.2</td>
<td>0</td>
<td>48.0</td>
<td>3.3</td>
<td>11.5</td>
<td>51.1</td>
</tr>
<tr>
<td>1.0</td>
<td>123490</td>
<td>0.8</td>
<td>0</td>
<td>48.3</td>
<td>2.9</td>
<td>12.2</td>
<td>52.0</td>
</tr>
</tbody>
</table>

### File Name: containe.yuv.

**Lossless Compression Scheme: Lempel Ziv Welch Coding**

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>32255</td>
<td>14.0</td>
<td>54.2</td>
<td>85.1</td>
<td>41.5</td>
<td>5.9</td>
<td>63.1</td>
</tr>
<tr>
<td>0.2</td>
<td>34063</td>
<td>12.5</td>
<td>56.4</td>
<td>76.5</td>
<td>37.4</td>
<td>4.5</td>
<td>62.2</td>
</tr>
<tr>
<td>0.3</td>
<td>36666</td>
<td>10.7</td>
<td>54.8</td>
<td>67.9</td>
<td>32.7</td>
<td>4.3</td>
<td>62.6</td>
</tr>
<tr>
<td>0.4</td>
<td>46513</td>
<td>7.7</td>
<td>54.9</td>
<td>62.5</td>
<td>24.8</td>
<td>4.5</td>
<td>60.6</td>
</tr>
<tr>
<td>0.5</td>
<td>58719</td>
<td>6.3</td>
<td>56.7</td>
<td>66.6</td>
<td>21.6</td>
<td>6.7</td>
<td>59.4</td>
</tr>
<tr>
<td>0.6</td>
<td>71556</td>
<td>4.7</td>
<td>54.8</td>
<td>67.7</td>
<td>15.8</td>
<td>8.1</td>
<td>58.9</td>
</tr>
<tr>
<td>0.7</td>
<td>84030</td>
<td>3.9</td>
<td>55.5</td>
<td>68.7</td>
<td>13.4</td>
<td>10.5</td>
<td>57.4</td>
</tr>
<tr>
<td>0.8</td>
<td>96646</td>
<td>3.3</td>
<td>57.1</td>
<td>67.9</td>
<td>10.8</td>
<td>12.8</td>
<td>58.2</td>
</tr>
<tr>
<td>0.9</td>
<td>109259</td>
<td>2.8</td>
<td>54.9</td>
<td>73.1</td>
<td>8.1</td>
<td>17.5</td>
<td>58.2</td>
</tr>
<tr>
<td>1.0</td>
<td>121788</td>
<td>2.5</td>
<td>54.2</td>
<td>75.1</td>
<td>6.4</td>
<td>20.4</td>
<td>59.4</td>
</tr>
</tbody>
</table>

**Lossless Compression Scheme: Huffman Coding**

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>39132</td>
<td>4.9</td>
<td>0</td>
<td>48.7</td>
<td>18.0</td>
<td>0</td>
<td>54.6</td>
</tr>
<tr>
<td>0.2</td>
<td>40939</td>
<td>4.6</td>
<td>0</td>
<td>45.5</td>
<td>16.5</td>
<td>4.0</td>
<td>54.2</td>
</tr>
<tr>
<td>0.3</td>
<td>43089</td>
<td>4.3</td>
<td>0</td>
<td>45.0</td>
<td>14.2</td>
<td>8.9</td>
<td>53.9</td>
</tr>
<tr>
<td>0.4</td>
<td>47424</td>
<td>4.0</td>
<td>0</td>
<td>42.8</td>
<td>12.4</td>
<td>16.8</td>
<td>53.5</td>
</tr>
<tr>
<td>0.5</td>
<td>61489</td>
<td>2.1</td>
<td>0</td>
<td>44.4</td>
<td>0</td>
<td>31.8</td>
<td>50.8</td>
</tr>
<tr>
<td>0.6</td>
<td>72855</td>
<td>3.2</td>
<td>0</td>
<td>44.9</td>
<td>6.7</td>
<td>46.2</td>
<td>51.0</td>
</tr>
<tr>
<td>0.7</td>
<td>84933</td>
<td>2.9</td>
<td>0</td>
<td>46.6</td>
<td>5.7</td>
<td>52.9</td>
<td>52.0</td>
</tr>
<tr>
<td>0.8</td>
<td>97224</td>
<td>2.8</td>
<td>0</td>
<td>47.8</td>
<td>4.8</td>
<td>59.0</td>
<td>53.6</td>
</tr>
<tr>
<td>0.9</td>
<td>109461</td>
<td>2.5</td>
<td>0</td>
<td>48.6</td>
<td>4.4</td>
<td>62.2</td>
<td>54.5</td>
</tr>
<tr>
<td>1.0</td>
<td>122434</td>
<td>1.9</td>
<td>0</td>
<td>48.4</td>
<td>0</td>
<td>64.2</td>
<td>54.3</td>
</tr>
</tbody>
</table>

**Lossless Compression Scheme: Lempel Ziv Welch Coding**

<table>
<thead>
<tr>
<th>Bit Rate (Mbit/s)</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>36750</td>
<td>11.2</td>
<td>55.0</td>
<td>85.6</td>
<td>31.5</td>
<td>8.8</td>
<td>63.1</td>
</tr>
<tr>
<td>0.2</td>
<td>38561</td>
<td>10.1</td>
<td>54.7</td>
<td>75.6</td>
<td>29.4</td>
<td>12.4</td>
<td>61.2</td>
</tr>
<tr>
<td>0.3</td>
<td>40871</td>
<td>9.2</td>
<td>54.2</td>
<td>72.7</td>
<td>25.6</td>
<td>16.3</td>
<td>61.3</td>
</tr>
<tr>
<td>0.4</td>
<td>45320</td>
<td>8.2</td>
<td>55.0</td>
<td>67.1</td>
<td>22.5</td>
<td>24.4</td>
<td>61.3</td>
</tr>
<tr>
<td>0.5</td>
<td>58972</td>
<td>6.1</td>
<td>56.0</td>
<td>62.9</td>
<td>14.3</td>
<td>39.1</td>
<td>57.5</td>
</tr>
<tr>
<td>0.6</td>
<td>71115</td>
<td>5.5</td>
<td>57.2</td>
<td>66.0</td>
<td>13.3</td>
<td>49.0</td>
<td>57.4</td>
</tr>
<tr>
<td>0.7</td>
<td>83204</td>
<td>4.9</td>
<td>55.1</td>
<td>69.3</td>
<td>12.0</td>
<td>54.4</td>
<td>57.7</td>
</tr>
<tr>
<td>0.8</td>
<td>95441</td>
<td>4.5</td>
<td>56.5</td>
<td>72.0</td>
<td>10.8</td>
<td>60.9</td>
<td>58.9</td>
</tr>
<tr>
<td>0.9</td>
<td>107670</td>
<td>4.1</td>
<td>55.6</td>
<td>73.6</td>
<td>10.1</td>
<td>64.1</td>
<td>59.8</td>
</tr>
<tr>
<td>1.0</td>
<td>120108</td>
<td>3.8</td>
<td>55.1</td>
<td>74.2</td>
<td>9.3</td>
<td>67.0</td>
<td>59.8</td>
</tr>
</tbody>
</table>
### Separating the MPEG-2 Bitstream for Lossless Compression and Transmission

**Fixed bit rate, variable number of frames**

Number of frames: 60-300  
Bit Rate: 0.5 Mbit/s

**File Name:** *mad.yuv.*

**Lossless Compression Scheme:** Huffman Coding

<table>
<thead>
<tr>
<th># Frames</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>123933</td>
<td>2.8</td>
<td>45.1</td>
<td>47.4</td>
<td>7.1</td>
<td>0</td>
<td>52.4</td>
</tr>
<tr>
<td>120</td>
<td>244077</td>
<td>3.3</td>
<td>50.0</td>
<td>48.8</td>
<td>7.7</td>
<td>5.8</td>
<td>53.5</td>
</tr>
<tr>
<td>180</td>
<td>364515</td>
<td>3.4</td>
<td>51.8</td>
<td>49.2</td>
<td>7.8</td>
<td>5.9</td>
<td>53.8</td>
</tr>
<tr>
<td>240</td>
<td>485283</td>
<td>3.4</td>
<td>52.7</td>
<td>49.5</td>
<td>8.0</td>
<td>6.5</td>
<td>53.9</td>
</tr>
<tr>
<td>300</td>
<td>605478</td>
<td>3.5</td>
<td>53.2</td>
<td>49.5</td>
<td>8.1</td>
<td>8.2</td>
<td>53.9</td>
</tr>
</tbody>
</table>

**Lossless Compression Scheme:** Lempel Ziv Welch Coding

<table>
<thead>
<tr>
<th># Frames</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>120973</td>
<td>5.1</td>
<td>67.8</td>
<td>70.2</td>
<td>11.8</td>
<td>7.7</td>
<td>59.7</td>
</tr>
<tr>
<td>120</td>
<td>238179</td>
<td>5.6</td>
<td>76.3</td>
<td>74.6</td>
<td>12.4</td>
<td>6.8</td>
<td>60.9</td>
</tr>
<tr>
<td>180</td>
<td>355231</td>
<td>5.8</td>
<td>79.8</td>
<td>76.1</td>
<td>12.8</td>
<td>6.3</td>
<td>61.5</td>
</tr>
<tr>
<td>240</td>
<td>472601</td>
<td>6.0</td>
<td>81.7</td>
<td>77.6</td>
<td>12.9</td>
<td>6.5</td>
<td>61.6</td>
</tr>
<tr>
<td>300</td>
<td>588986</td>
<td>6.1</td>
<td>83.3</td>
<td>79.0</td>
<td>13.3</td>
<td>7.5</td>
<td>61.6</td>
</tr>
</tbody>
</table>

**File Name:** *akiyo.yuv.*

**Lossless Compression Scheme:** Huffman Coding

<table>
<thead>
<tr>
<th># Frames</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>123613</td>
<td>3.4</td>
<td>44.9</td>
<td>47.9</td>
<td>10.9</td>
<td>0</td>
<td>52.6</td>
</tr>
<tr>
<td>120</td>
<td>243507</td>
<td>3.7</td>
<td>49.8</td>
<td>49.4</td>
<td>11.7</td>
<td>0</td>
<td>53.9</td>
</tr>
<tr>
<td>180</td>
<td>362597</td>
<td>4.1</td>
<td>51.7</td>
<td>49.6</td>
<td>12.1</td>
<td>7.7</td>
<td>54.2</td>
</tr>
<tr>
<td>240</td>
<td>482144</td>
<td>4.2</td>
<td>52.8</td>
<td>49.9</td>
<td>12.4</td>
<td>8.3</td>
<td>54.3</td>
</tr>
<tr>
<td>300</td>
<td>601480</td>
<td>4.2</td>
<td>53.4</td>
<td>50.0</td>
<td>12.5</td>
<td>8.5</td>
<td>54.3</td>
</tr>
</tbody>
</table>

**Lossless Compression Scheme:** Lempel Ziv Welch Coding

<table>
<thead>
<tr>
<th># Frames</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>119366</td>
<td>6.7</td>
<td>65.6</td>
<td>71.6</td>
<td>21.3</td>
<td>10.6</td>
<td>61.1</td>
</tr>
<tr>
<td>120</td>
<td>233428</td>
<td>7.7</td>
<td>75.5</td>
<td>77.7</td>
<td>23.5</td>
<td>12.5</td>
<td>63.1</td>
</tr>
<tr>
<td>180</td>
<td>347506</td>
<td>8.1</td>
<td>79.5</td>
<td>79.6</td>
<td>24.6</td>
<td>13.8</td>
<td>63.2</td>
</tr>
<tr>
<td>240</td>
<td>461148</td>
<td>8.3</td>
<td>81.8</td>
<td>80.8</td>
<td>25.5</td>
<td>14.6</td>
<td>63.4</td>
</tr>
<tr>
<td>300</td>
<td>574311</td>
<td>8.5</td>
<td>83.4</td>
<td>81.8</td>
<td>26.2</td>
<td>14.8</td>
<td>63.6</td>
</tr>
</tbody>
</table>
Appendix A: Tables

File Name: containe.yuv.

**Lossless Compression Scheme: Huffman Coding**

<table>
<thead>
<tr>
<th># Frames</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>121935</td>
<td>4.7</td>
<td>44.7</td>
<td>47.3</td>
<td>9.2</td>
<td>42.6</td>
<td>52.9</td>
</tr>
<tr>
<td>120</td>
<td>239766</td>
<td>5.2</td>
<td>49.5</td>
<td>49.0</td>
<td>10.3</td>
<td>45.6</td>
<td>53.8</td>
</tr>
<tr>
<td>180</td>
<td>357434</td>
<td>5.4</td>
<td>51.4</td>
<td>49.3</td>
<td>10.8</td>
<td>47.2</td>
<td>54.0</td>
</tr>
<tr>
<td>240</td>
<td>475438</td>
<td>5.5</td>
<td>52.3</td>
<td>49.4</td>
<td>11.0</td>
<td>48.5</td>
<td>54.1</td>
</tr>
<tr>
<td>300</td>
<td>593172</td>
<td>5.6</td>
<td>52.9</td>
<td>49.7</td>
<td>11.1</td>
<td>48.9</td>
<td>54.2</td>
</tr>
</tbody>
</table>

**Lossless Compression Scheme: Lempel Ziv Welch Coding**

<table>
<thead>
<tr>
<th># Frames</th>
<th>Compressed Bitstream Size (Bytes)</th>
<th>CR (%)</th>
<th>HL CSI (%)</th>
<th>SLC CSI (%)</th>
<th>MB CSI (%)</th>
<th>MV CSI (%)</th>
<th>CAI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>118603</td>
<td>7.3</td>
<td>67.1</td>
<td>72.0</td>
<td>16.9</td>
<td>45.7</td>
<td>59.5</td>
</tr>
<tr>
<td>120</td>
<td>232054</td>
<td>8.2</td>
<td>75.6</td>
<td>79.6</td>
<td>18.9</td>
<td>47.2</td>
<td>60.6</td>
</tr>
<tr>
<td>180</td>
<td>345193</td>
<td>8.6</td>
<td>79.8</td>
<td>82.4</td>
<td>19.7</td>
<td>47.9</td>
<td>61.0</td>
</tr>
<tr>
<td>240</td>
<td>458492</td>
<td>8.9</td>
<td>81.8</td>
<td>83.7</td>
<td>20.3</td>
<td>48.0</td>
<td>61.2</td>
</tr>
<tr>
<td>300</td>
<td>571628</td>
<td>9.0</td>
<td>83.2</td>
<td>84.5</td>
<td>20.7</td>
<td>49.0</td>
<td>61.5</td>
</tr>
</tbody>
</table>
Appendix B: Lossless Compression and Expansion Theory

The theory involved in both Huffman and LZW schemes, is explained in this Section.

B.1 Huffman Coding and Decoding

Huffman Compression, also known as Huffman Encoding, is one of many compression techniques in use today [6]. One of the main benefits of Huffman Compression is how easy it is to understand and implement while still providing a decent compression ratio on average files.

The Huffman compression algorithm assumes that data files consist of some byte values that occur more frequently than other byte values in the same file. This is true for text files and most raw gfx images, as well as EXE and COM file code segments. Huffman coding attempts to create a minimum redundancy code, that minimizes the average number of bits per character. Let $N$ = number of characters (256 for eight bit codes) and $p(i) = \text{probability of the } i^{th} \text{ character (byte) occurring}$. So, $\sum_{i=1}^{N} p(i) = 1$. Then, we let $L(i) = \text{number of bits to represent the } i^{th} \text{ character}$ and we want to minimize the average number of bits needed to code a file as shown in this equation: $L_{\text{average}} = \sum_{i=1}^{N} L(i) \times p(i)$.

For this purpose, the algorithm builds a "Frequency Table" for each byte value within a file. With the frequency table the algorithm can then build the "Huffman Tree". The purpose of the tree is to associate each byte value with a bit string of variable length. The more frequently used characters get shorter bit strings, while the less frequent characters get longer bit strings. Thus the data file may be compressed.

To compress the file, the Huffman algorithm reads the file a second time, converting each byte value into the bit string assigned to it by the Huffman Tree and then writing the bit string to a new file.

The decompression routine reverses the process by reading in the stored frequency table (stored in the compressed file as a header) that was used in compressing the file. With the frequency table the decompressor can then re-build the Huffman Tree, and from that, extrapolate all the bit strings stored in the compressed file to their original byte value form.

The first task of the Huffman algorithm is to convert a data file into a frequency table. As an example, our data file might contain the text (excluding the quotation marks): "this is a test".
The frequency table will tell us the frequency of each character in the file. In this case the frequency table will look like this:

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency (times)</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>3</td>
<td>3/14</td>
</tr>
<tr>
<td>s</td>
<td>3</td>
<td>3/14</td>
</tr>
<tr>
<td>&lt;space&gt;</td>
<td>3</td>
<td>3/14</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>2/14</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>1/14</td>
</tr>
<tr>
<td>h</td>
<td>1</td>
<td>1/14</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1/14</td>
</tr>
</tbody>
</table>

Table B-1 Frequency and probability for each character.

The Huffman algorithm then builds the Huffman Tree using the frequency table. The tree structure contains nodes, each of which contains a character, its frequency, a pointer to a parent node, and pointers to the left and right child nodes. The tree can contain entries for all 256 possible characters and all 255 possible parent nodes. At first there are no parent nodes. The tree grows by making successive passes through the existing nodes. Each pass searches for two nodes “that have not grown a parent node” and that have the two lowest frequency counts. When the algorithm finds those two nodes, it allocates a new node, assigns it as the parent of the two nodes, and gives the new node a frequency count that is the sum of the two children nodes frequency counts. The following iterations ignore those two children nodes but include the new parent node. The passes continue until only one node with no parent remains. That node will be the root node of the tree. The tree for our example text will look like in Figure B.1.

Compression then involves traversing the tree beginning at the leaf node for the character to be compressed and navigating to the root. This navigation iteratively selects the parent of the current node and sees whether the current node is the "right" or "left" child of the parent, thus determining if the next bit is a one (1) or a zero (0). Because we are proceeding from leaf to root, we are collecting bits in the “reverse” order in which we will write them to the compressed file.

The assignment of the 0 bit to the left branch and the 1 bit to the right branch is arbitrary. Figure B.2 shows the tree with 1:s and 0:s assigned to each branch.
Figure B.1 Huffman Tree using the frequency table.

Figure B.2 Tree with 1:s and 0:s assigned to each branch.
The tree in this example would compress "this is a test" into the bit stream:

<table>
<thead>
<tr>
<th>t</th>
<th>h</th>
<th>i</th>
<th>s</th>
<th>i</th>
<th>s</th>
<th>a</th>
<th>t</th>
<th>e</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11110</td>
<td>110</td>
<td>00</td>
<td>01</td>
<td>110</td>
<td>00</td>
<td>01</td>
<td>11111</td>
<td>01</td>
<td>1110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Partial Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>10111101</td>
<td>10000111</td>
<td>00001111</td>
<td>11011011</td>
<td>100010…</td>
</tr>
</tbody>
</table>

Decompression involves re-building the Huffman tree from a stored frequency table (again, in the header of the compressed file), and converting its bit streams into characters. We read the file a bit at a time. Beginning at the root node in the Huffman Tree and depending on the value of the bit, we take the right or left branch of the tree and then return to read another bit. When the node we select is a leaf (it has no right and left child nodes) we write its character value to the decompressed file and go back to the root node for the next bit.

For this example, we obtain the next code word lengths (bits/symbol) table:

<table>
<thead>
<tr>
<th>Character</th>
<th>Length</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>2</td>
<td>3/14</td>
</tr>
<tr>
<td>s</td>
<td>2</td>
<td>3/14</td>
</tr>
<tr>
<td>&lt;space&gt;</td>
<td>2</td>
<td>3/14</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td>2/14</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>1/14</td>
</tr>
<tr>
<td>h</td>
<td>5</td>
<td>1/14</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>1/14</td>
</tr>
</tbody>
</table>

Table B-2 Code word length and probabilities for each character.

Then, the average length is $L_{\text{average}} = 2.71$ bits/symbol. We can compare this result with the entropy of the source (file): $H(i) = \sum_{i=1}^{N} p(i) \times \log_2 \frac{1}{p(i)}$. In this case, $H(i) = 2.66$ bits/symbol.
Appendix B.2 – Lempel Ziv Welch Coding and Decoding

For a detailed description of Huffman and LZW compression, see [7].