Technische Universität Wien
Institut für Nachrichtentechnik und Hochfrequenztechnik
Universitat Politècnica de Catalunya
Escola Politècnica Superior de Castelldefels

Master Thesis

Performance of an Error Detection Mechanism for Damaged H. 264/AVC Sequences

by

Josep Colom Ikuno

Supervisor: Univ.Prof. Dr. Markus Rupp, Luca Superiori

Vienna, 16th December 2007

Abstract

In mobile video applications, the error-prone wireless connection can cause the stream to be incorrectly received. An occurring error will propagate both spatially (in the current frame) and temporally (to the following frames).

This work presents the implementation of an error detection and concealment mechanism for H.264/AVC encoded video and the design of a quality estimator. The detection is performed by means of two interacting strategies. At bit level, the syntax of the received bitstream will be analyzed in order to detect inconsistent or illegal codewords. At the pixel level, the remaining visual impairments in the decoded frame will be detected.

The quality estimator is capable of, given the information output by the decoder, to estimate the subjective quality of the decoded H.264 video.

This detection and concealment is implemented in the H.264/AVC decoder, without causing transmission overhead. Simulations show improvements both in objective (luminance peak-signal-to-noise ratio) and subjective (mean opinion score) tests with respect to the common slice rejection mechanism. The quality estimator is only a Matlab design and is not implemented in the decoder.

Contents

Chapter 1

Introduction

Due to the rapid growth of wireless communications, video over wireless networks has gained a lot of attention. This project deals with transmission of QCIF (176x144 pixels resolution) videos encoded with the H.264/AVC codec and streamed in real time over mobile networks.

In this type of applications, because of the use of the wireless channel which is an error-prone channel (Figure 1.1), streams can be received erroneously. An occurring error could then propagate both spatially (in the current frame) and temporally (to the following frames).

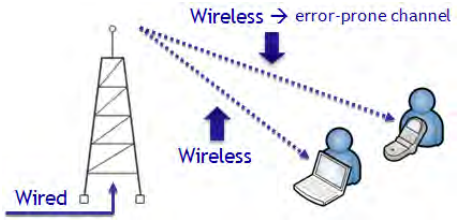


Figure 1.1: Video streaming in mobile applications

This work presents the implementation of an error detection and concealment mechanism for H.264/AVC encoded video sequences and the design of a quality estimator. The quality estimator uses the data output by the newly implemented decoder and is capable of estimating the subjective quality of the decoded sequence without any other reference. The implementation has been done using the JM v.10.2 reference H.264/AVC decoder [1] as a base. The detection is performed by means of two interacting strategies.

At bit level, the syntax of the received bitstream is analyzed in order to detect inconsistent or illegal codewords caused by the desynchronization of the bitstream due to a transmission error. This method was already implemented in a modified version of the JM software, but needed to be modified in order to interact with the other method, which needed to be implemented in the decoder.

The other detection and concealment method, which works at pixel level, has been implemented and works in conjunction with syntax analysis. With this additional algorithm, remaining visual impairments in the decoded frame can be detected by means of simple image processing.

These strategies are implemented in the H.264/AVC decoder without causing transmission

overhead and without needing any modification in the H.264/AVC encoder. Simulations show improvements both using objective (Luminance Peak Signal-to-Noise-Ratio or Y-PSNR) and subjective (Mean Opinion Score or MOS) metrics with respect to the common slice rejection mechanism, in which erroneous slices are discarded and replaced by the one in the previous frame.

Chapter 2

H.264/AVC

H.264/AVC (Advanced Video Coding) [2] [3] is the newest video coding standard, written by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC Moving Picture Experts Group (MPEG) as the product of a collective partnership effort known as the Joint Video Team (JVT). This standard is especially suitable for low data rate applications as it provides substantially better video quality at the same data rates compared to previous standards (MPEG-2, MPEG-4, H.263), with only a moderate increase of the complexity. Moreover, H.264/AVC was designed to support a wide variety of applications and to operate over several types of networks and systems.

It increases video quality, both objectively and subjectively. Figure 2.1a [4] depicts the rate-PSNR curves of QCIF *foreman* sequence encoded using H.264, H.263 Baseline and H.263 CHC encoders, while Figure 2.1b the subjective quality improvement with H.264 (right) against MPEG-2 (left) of a QCIF *foreman* sequence encoded at 100 Kbps.

2.1 Overview

H.264 compresses the data by exploiting spatial and temporal redundancies in the video stream.

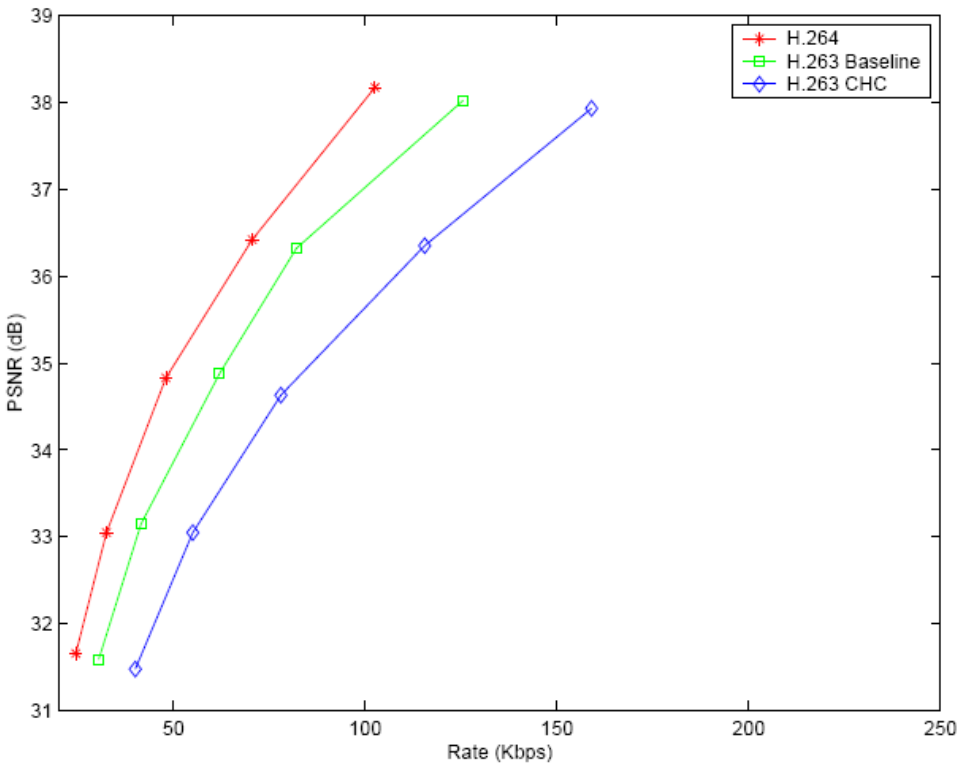
As in all prior ITU-T and ISO/IEC JTC1 video standards since H.261, the VCL design follows the so-called block-based hybrid video coding approach, in which each coded picture is represented in block-shaped units of associated luma and chroma samples called macroblocks.

The basic source-coding algorithm is a hybrid of inter-picture prediction to exploit temporal statistical dependencies and transform coding of the prediction residual to exploit spatial statistical dependencies (intra-picture prediction) [5](Figure 2.2).

The standard defines different sets of capabilities, defined as *profiles*, adapted to the needs of different applications:

- **Baseline Profile:** Primarily defined for lower-cost applications with limited computing resources, this profile is used widely in video conferencing and mobile applications.

- **Main Profile:** Intended as the mainstream consumer profile for broadcast and storage applications.



(a) Comparison of H.264 using objective metrics

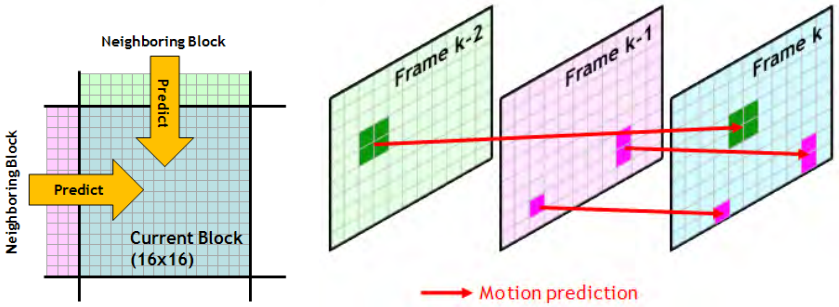
(b) Subjective comparison of H.264

Figure 2.1: Comparison of H.264 vs previous codecs

- **Extended Profile:** Intended as the streaming video profile, this profile has relatively high compression capability and some extra tricks for robustness against data losses and server stream switching.

Additionally, there exist the *high profiles*, designed for use with high definition (HD) video.

The profile used in this work is the baseline profile, as is the only one recommended for use in UMTS [6] (it is the one most suitable for mobile applications due to its low computational complexity, compared to the other profiles). Figure 2.3 shows the set of capabilities for the H.264/AVC profiles, highlighting the ones of the baseline profiles, the one used in this work.



(a) Spatial prediction (b) Temporal prediction

Figure 2.2: Prediction in H.264/AVC

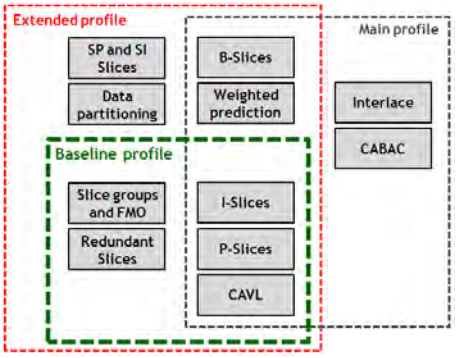


Figure 2.3: Capabilities for H.264/AVC profiles

2.2 Structure

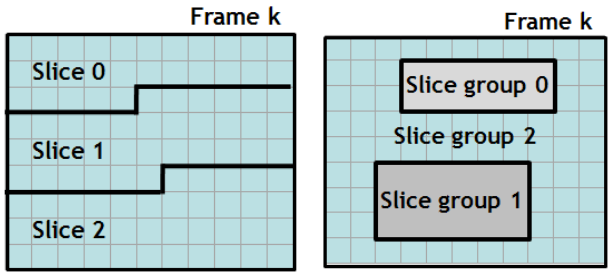
The video color space used by H.264/AVC separates a color representation into three components called Y, Cb, and Cr. Component Y is called *luma*, and represents brightness. The two *chroma* components Cb and Cr represent the extent to which the color deviates from gray toward blue and red, respectively.

Because the human visual system is more sensitive to luma than chroma, H.264/AVC uses a sampling structure in which the chroma component has one fourth of the number of samples than the luma component (half the number of samples in both the horizontal and vertical dimensions). This is called 4:2:0 sampling with 8 bits of precision per sample. The sampling structure used is the same as in MPEG-2 Main-profile video.

Although some of the *high profile* modes also support higher-resolution chroma and a larger number of bits per sample, the 3 main profiles and specifically baseline profile, the one used in this work, use 4:2:0 sampling with 8 bits of precision per sample.

Each frame to be encoded is partitioned into fixed-size macroblocks, where each macroblock covers a rectangular frame area of 16x16 samples of the luma component and 8x8 samples of each of the two chroma components. This means that in the case of QCIF videos, which have a resolution of 176x144 pixels, one frame consists of 11x9 MBs (99 MBs). This is because the luma component of the frame is 176x144 pixels while each chroma component is 88x72 pixels.

The macroblocks (MBs) are the basic building blocks of the standard for which the coding and decoding process is specified. They are organized in slices, which represent a subset of a given picture that can be decoded independently from the other slices in the same picture. In this work, one slice corresponds to one packet sent through the network. The transmission order of macroblocks in the bit stream depends on the so-called Macroblock Allocation Map (MAM), and although in this work the macroblocks are allocated in the slices in raster order (Figure 2.4a), that is not necessarily the case, as Flexible Macroblock Ordering (FMO) could be used (Figure 2.4b).



(a) Without FMO (b) With FMO

Figure 2.4: Subdivision of a picture into slices.

The H.264/AVC design covers a Video Coding Layer (VCL), which efficiently represents the video content, performing the core block-based hybrid coding functions, and a Network Abstraction Layer (NAL), which formats the VCL representation of the video and provides header information in a manner appropriate for conveyance by particular transport layers or storage media [7].

H.264/AVC supports five different slice-coding types. The simplest one is the I slice (where I stands for intra). In I slices, all macroblocks are coded without referring to other pictures within the video sequence. On the other hand, prior-coded images can be used to form a prediction signal for macroblocks of the predictive-coded P and B slices (where P stands for predictive and B stands for bi-predictive). The remaining two slice types are SP (switching P) and SI (switching I), which are specified for efficient switching between bitstreams coded at various bit-rates.

In baseline profile, only I and P slices are available. Although the codec defines slice types, it is not common to have different types of slices in the same frame, thus the terms I frame and P frame are more commonly used.

- I frames are coded without reference to any MB outside of the current slice, they exploit spatial redundancy with the neighboring MBs (Figure 2.2a). Here, a MB is decoded using previously decoded MBs from the same slice as reference.

- P frames are coded referencing MBs in previous frames through a process known as *motion estimation* (Figure 2.2b), although the standard allows for intra-predicted MBs to be present in a P-slice. H.264 allows multiple reference picture motion compensation, so not only the previous frame can be referenced. Nonetheless, in this work a buffer size of one frame has been used, so only the last frame will be used for reference.

Each P-type macroblock corresponds to a specific partitioning of the macroblock into fixed-size blocks used for motion description. Typical luma bock sizes are 16x16, 16x8, 8x16 and 8x8.

The prediction signal for each predictive-coded $m \times n$ luma block is obtained by displacing an area of the corresponding reference picture, which is specified by a translational motion vector and a picture reference index.

The set of frames from an I frame up to the P frame preceding the next I frame is defined as Group Of Pictures (GOP). Since a picture buffer of one frame has been used, at the end of each GOP the picture buffer is effectively cleared.

The implications of having a set a buffer size is that, since the decoded frames are put in the *decoded picture buffer*, its size determines how many frames can the codec use for inter-prediction. Setting its size to one means that the temporal referencing can only be of the previous frame. Thus, sending one I frame (beginning of the GOP) cuts the temporal referencing and is equivalent to clearing the picture buffer.

Advancing some contents from Chapter 3, this GOP structure means that an error occurring in a certain frame could propagate until the end of that specific GOP, as subsequent frames could directly or indirectly use an erroneous frame as reference.

Figure 2.5 shows the series of I and P frames that form a GOP. The arrows represent the temporal referencing.

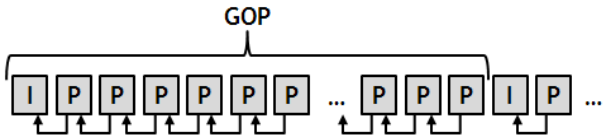


Figure 2.5: Group of Pictures (GOP)

The coded video data is organized into NAL units (NALUs), each of which is effectively a packet that contains an integer number of bytes. The first byte of each NAL unit is a header byte that contains an indication of the type of data in the NAL unit, and the remaining bytes contain payload data of the type indicated by the header.

Each NAL unit regardless of its type is encapsulated in the RTP/UDP/IP packet by adding header information of each protocol to the NAL unit (Figure 2.6). In this work, each NALU is limited to 700 bytes, each of them containing in our application exactly one slice.

Since temporal prediction is much more effective than spatial prediction, the size of I frames is in average 3-4 times bigger than the size of the P frames. A P frame will typically fit in a single slice, while an I frame usually needs the aforementioned 3-4 slices.

Figure 2.6: Encapsulation of NAL units in RTP/UDP/IP

2.3 Data in the NALU

As mentioned in Section 2.1 and 2.2, the encoder processes a frame of video in units of a macroblock (16x16 displayed pixels). It forms a prediction of the macroblock based on previously-coded data, either from the current frame (intra prediction) or from other frames that have already been coded and transmitted (inter prediction). The encoder then subtracts the prediction from the current macroblock to form a residual.

A block of residual samples is transformed using a 4x4 or 8x8 integer transform, an approximate form of the Discrete Cosine Transform (DCT).

The output of the transform, a block of transform coefficients, is then quantized (i.e. each coefficient is divided by an integer value). Quantization reduces the precision of the transform coefficients according to a quantization parameter (QP). Typically, the result is a block in which most or all of the coefficients are zero, with a few non-zero coefficients. This process keeps the information from the lower frequencies, whereas the high frequencies are set to 0. Setting QP to a high value means that more coefficients are set to zero, resulting in high compression at the expense of poor decoded image quality. Setting QP to a low value means that more non-zero coefficients remain after quantization, resulting in better decoded image quality but lower compression.

The video coding process produces a number of values that must be encoded to form the compressed bitstream. These values include:

- quantized transform coefficients
- information to enable the decoder to re-create the prediction
- information about the structure of the compressed data and the compression tools used during encoding and about the complete video sequence.

These values and parameters are converted into binary codes using variable length coding and/or arithmetic coding. Each of these encoding methods produces an efficient, compact binary representation of the information. The encoded bitstream can then be stored and/or transmitted.

NALUs are subdivided into non-VCL and VCL NAL units. This will be explained in more detail in Chapter 3, while explaining the effects of transmission errors in the video stream. Basically, Non-VCL NALUs transport a set of parameters informing the decoder how to perform the decoding, whilst VCL NALUs transport the actual data. Typically Non-VCL NALUs are transported over a reliable protocol, while VCL NALUs are not.

Chapter 3

Effects of errors

Video telephony and video streaming over IP packet networks are quite challenging application due to their requirement on delay and data rates.

A video stream is encoded and packetized in Real Time Protocol (RTP) packets. These packets are then transported end-to-end within User Datagram Protocol (UDP). Unlike Transmission Control Protocol (TCP), UDP does not provide any retransmissions control mechanism. Nevertheless, it has been widely adopted for video streaming and video telephony, since the end-to-end retransmissions would cause unacceptable delays. Thus, in such real-time applications, transmission errors cannot be completely avoided.

To allow for applications to be able to use the standard even in error-prone environments such as mobile networks, apart from the improved compression performance, H.264/AVC provides several error resilience features.

Therefore, the 3rd Generation Partnership Project (3GPP), the body standardizing the Universal Mobile Telecommunications Network (UMTS), has approved the inclusion of H.264/AVC as an optional feature in release 6 of its mobile multimedia telephony and streaming services specifications [8] [6].

To facilitate error detection at the receiving entity, each UDP datagram is provided with a simple 16 bit long checksum [9]. The packets with detected errors are typically discarded and missing parts of video are subsequently concealed. Since one packet contains one slice (Section 2.2), this method is called Slice Level Concealment. From now on, we will refer to this method as "SLC".

Transmission errors occur at bit level, but afterwards manifest themselves in the decoded image (at pixel level).

NALUs are subdivided into non-VCL and VCL NAL units. Non-VCL NALUs contain sets of video parameters. To this category belong the Sequence Parameter Set (SPS), defining profile, resolution and other properties of the sequence, and the Picture Parameter Set (PPS), containing the type of entropy coding, slice group and quantization properties.

VCL NALUs contain the data associated to the video slice. Each VCL NALU refers to a non-VCL NALU (Figure 3.1). In this work, the focus is on the detection of transmission errors in VCL NALUs, as non-VCL NALUs would not be transmitted within the RTP payload, but provided in the SDP (Session Description Protocol), which is commonly transmitted over a reliable connection (TCP).

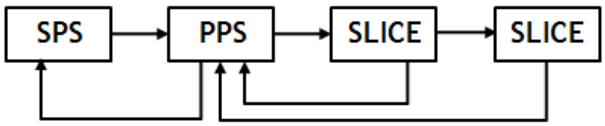


Figure 3.1: NALU sequence

3.1 Bitstream desynchronization

The standard [2] [3] defines several binarization ways of the data to be transmitted. Besides fixed length codes, several variable length coding strategies are used. The data to be sent is encoded in baseline profile by means of Contex Adaptive Variable Length Coding (CAVL) and exp-Golomb coding (also a variable length code).

As VLC maps source symbols to a variable number of bits codeword depending of the probability of their occurrence, a change in one bit could change the boundaries of the codewords, thus effectively desynchronizing the bitstream.
The data transmitted in VCL NALUs is comprised of a Slice Header (SH), followed by several MacroBlocks (MBs) pertaining to a slice (Figure 3.2).

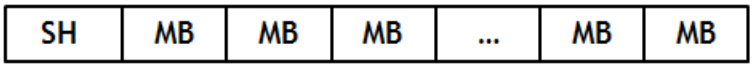


Figure 3.2: Structure of a VCL NALU

Although an error in the SH could make the entire slice undecodable (determines the decoding of the contained macroblocks), this work is focused on errors in the data part of the NALU, where the residuals are stored. Since there is no resynchronization word between SH and slice payload neither between macroblocks, a transmission error could easily alter the boundaries of the VLC words from the point of the error occurrence up to the end of the slice.

In baseline profile, residual block data is coded using CAVLC and other variable-length coded units are coded using exp-Golomb. To illustrate how desynchronization occurs, an example using exp-Golomb will be used.

Exp-Golomb codewords are constructed in the following way (Figure 3.3, Table 3.1 for examples):

1. Take the number in binary and add 1 to it (arithmetically). Write this down.
2. Count the bits written, subtract one, and write that number of starting zero bits preceding the previous bit string.

To decode, the process is as following:

1. Read in M leading zeros followed by 1.
2. Read M-bit $INFO$ field.
3. $coded_word = 2^M + INFO1$

In Figure 3.4, the different colours represent the boundaries of the codewords. Because of the transmission errors, these boundaries are changed, leading to desynchronization.

$$\underbrace{0_1 \cdots 0_m}_{M}\, 1\, \underbrace{b_1 \cdots b_m}_{M}$$

Figure 3.3: Exp-Golomb word structure

Codeword	Binary coded	Exp-Golomb coded
0	1	1
1	10	010
2	11	011
3	100	00100
4	101	00101
5	110	00110
6	111	00111
7	1000	0001000
8	1001	0001001
...	...	...

Table 3.1: Exp-Golomb codewords

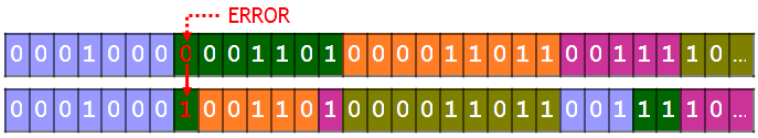


Figure 3.4: Bitstream desynchronization

3.2 Visual artifacts caused by bitstream desynchronization

In the previous Section, it was seen that an error in the bitstream could cause desynchronization of the encoded bitstream because of the use of VLC. Such errors would typically propagate until the end of the slice, as the decoder will continue decoding until the end of the slice.

It should be noted, though, that the standard decoder would typically discard the whole slice (SLC) and lose the correct information preceding the error. For this investigation, a modified decoder has been used, as explained in Chapter 5.

The errors in the bitstream can cause visual artifacts in the image. As each NALU contains one slice, propagation from the position of the error occurrence until the end of the NALU that the slice could be corrupt from the MB in which the error occurred up to the end of the slice (Spatial propagation, Figure 3.5a). Spatial propagation will specially manifest itself in I frames, as they are intra-predicted, meaning that following MBs could be predicted using erroneous MBs as reference.

Since the H.264/AVC codec uses temporal prediction, the P frames following the one where the error occurred will directly or indirectly use an erroneous frame as reference. This causes the visual artifacts to propagate to subsequent frames. This can be seen in Figure 3.5b.

The error will propagate until the end of the GOP, at which point the picture buffer is cleared (Figure 3.5c). The latter is true because in this project a buffer size of size 1 is used.

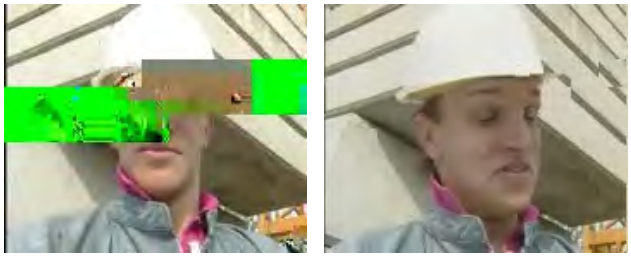
Although judging from this data one could reach the conclusion that the effect of transmission errors is the same (propagation until the end of the slice) independently of the type

(a) Spatial propagation (b) Temporal propagation (c) Resynchronization

Figure 3.5: Error propagation and resynchronization

of frame, this is not the case.

Errors in I frames (Figure 3.6a) tend to produce much more visible artifacts than for P frames (Figure 3.6b). This is caused by the different type of information that the NALU transports for each type of prediction.



(a) I frame artifact (b) P frame artifact

Figure 3.6: Visual artifacts

The reason for this is that inter-predicted macroblocks code consist of the information about the reference (index of the frame in the buffer and the motion vector) and residual levels. Since the residuals are typically small or even skipped, their errors have often a negligible influence on the quality and on error propagation. The errors in motion vectors and/or reference frames cause artifacts similar to those of temporal error concealment, i.e. spatial shifting of affected macroblocks. This means that the VLC desynchronizes rarely.

Also, since encoding a motion vector for each partition can take a significant number of bits, especially if small partition sizes are chosen, the standard states for motion vector prediction to be used. Motion vectors for neighboring partitions are often highly correlated and so each motion vector is predicted from vectors of nearby, previously coded partitions.

In addition, the absence of spatial prediction results in lack of direct artifact propagation within the same frame (although not in the same degree as in I frames, in P frames spatial error propagation can also be present, and is caused by decoding desynchronization and by wrong motion prediction).

As a result, in P frames, MBs following the error could be still consistent (the user does not perceive them as an artifact). The main cause of this is that usually, the error lies in the `mb_skip` parameter, so theoretically erroneous MBs are skipped. Because of the motion compensation in used in H.264, those MBs will not be perceived as erroneous.

As a summary, the following can be said about the effects of errors in the bitstream:

- Even a simple bit inversion can cause desynchronization.
- I frames are much more sensitive to desynchronization than P frames.
- The artifacts that such errors will produce will be much worse for I frames than for P frames.
- In I frames, the error will propagate spatially and then to the subsequent P frames.
- In P frames, the error will rarely propagate spatially, but will still propagate to subsequent P frames.

Chapter 4

Error detection and concealment

As this work consists of the implementation of the VIDC algorithm into the JM reference H.264/AVC decoder, its optimization and its performance evaluation, this Chapter is part of the core of this work. Here it will be explained how the two algorithms that are used to detect and conceal visual artifacts generated by transmission errors in the modified JM 10.2 modified decoder work and how VIDC has been embedded into the decoder and the performance results of the resulting error concealment.

The algorithms used for error detection and concealment are Syntax Check analysis (SC)[10] and Visual Impairments Detection and Concealment (VIDC)[11].

While the first works by monitoring when the decoder, due to the desynchronization of the bitstream performs an illegal operation, the second works by analyzing the decoded pictures in search of visually perceivable artifacts.

Although in this section it will be mentioned several times that the errors will be concealed, the main focus of the algorithms is the error detection. This is why a very simple concealment method (copy-paste) has been used.

4.1 Syntax check analysis

The algorithm is a syntax check mechanism capable of spotting errors during the decoding of the stream. It monitors the decoding process and logs when:

- illegal operations occur due to the decoder being incapable of decoding the bitstream.
- the decoder is fed erroneous instruction.

In order to suite the structure of the JM reference software, the macroblock decoding process has been subdivided into two main blocks (Figure 4.1. During the *READ* phase, the raw bitstream is read and partitioned in codewords. During the *DECODE* phase these codewords are interpreted as information elements (IE) and used to reconstruct the slice.

The algorithm is explained in [10].

The algorithm categorizes the errors in three subtypes, depending of their characteristics:

- **Illegal Codeword IC:** Arises when the codeword does not find correspondence in the appropriate look-up table. IC occurs during the *READ* process for tabled codewords.

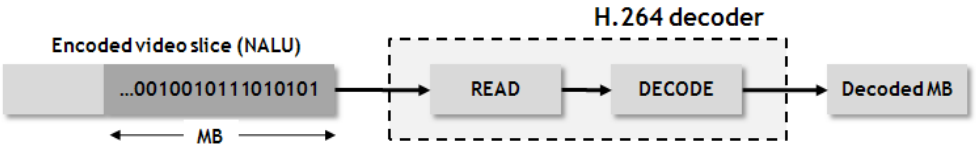


Figure 4.1: Conceptual scheme the decoder

- **Out of Range Codeword OR:** Results when the decoded value lies outside the legal range. It appears during *READ* process for all types of codewords. If the decoded parameter can only take values between $[-K, K]$, an error is produced if the absolute value of the read parameter is greater than K.
- **Contextual Error CE:** Occurs when the decoded word leads the decoder to illegal actions. It arises during the *DECODE* phase for all types of encoded parameters.

The presented errors are not strictly related to current bitstream failures. They are rather referred to the detectable anomalies, possibly caused by propagation of previously undetected errors.

4.1.1 Error detection

The algorithm works by logging when an error occurs in either the *READ* or *DECODE* phase and then trying to conceal those errors. Figure 4.2 illustrates how the algorithm works.

The encoded video slice, contained in a NALU is fed to the decoder. At this point, the decoder will have to (1) read the encoded bitstream and partition it into codewords and (2) process these codewords in order to reconstruct the slice.

In case an IC or OR error occurs during the *READ* phase or a CE error occurs during the *DECODE* phase, the algorithm will log this error and conceal the error. As it will be seen in Section 4.1.2, this means that the MBs from the detected error up the end of the slice are concealed. Figures 4.2 and 4.3 illustrate how the algorithm works (in conjunction with the H.264 decoder).

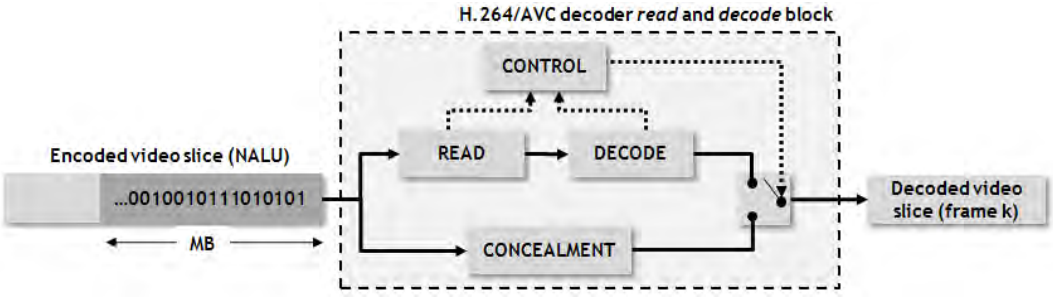


Figure 4.2: Block diagram of the SC algorithm

The SC analysis detects when an error occurs leads the decoder to perform illegal (not allowed) actions, which is not necessarily the moment when the error occurred. Also, some inserted errors may not trigger any consequence while decoding but nevertheless produce visible artifacts in the decoded image.

Input: NALU
Output: Decoded slice, detected errors location

try to read NALU;
if *error occurred* **then**
| log position of error;
end
try to decode slice;
if *error occurred* **then**
| log position of error;
end
if *error occurred* **then**
| conceal;
end

Figure 4.3: SC algorithm

While SC does not assure detection of all errors or location of their exact position, it assures that whenever it detects an error, there is an error at the point of detection or earlier in the current slice. While using SC, we define a *detection distance*, as the distance in MBs between the point an error occurs to the point it is detected.

This is illustrated in Figure 4.4, where the horizontal line represents the distance (expressed in MBs) between the error occurrence (1) and (2) the error detection.

Figure 4.4: Detection distance

It is critical for the usefulness of the algorithm that the detection distance is as small as possible. Simulations show that the detection distance is different for I frames (Figure 4.5 than for P frames (Figure 4.6. In I frames there is much more information encoded in the bitstream, so a desynchronization can be more easily detected.

For both frame types, [10] calculates the detection probability. More than 60% of the errors inserted in I frame are detected, for P frames the percentage is 47%.

For I frames, the average detection delay is 1.39 MB and over 85% of the errors are detected within 2 MBs. The interval between (1) and (2) in Figure 4.4, where the macroblocks are incorrectly decoded, is therefore extremely narrow. For P frames, the average detection delay is bigger: 15 MBs.

As seen in Section 3.2, the effect of an error is different for I and P frames. While in I frames the artifact tends to propagate until the end of the slice, in P frames it normally consists of isolated artifacts with normally a block-like shape.

While the detection probability is an important factor, it is also important to see how

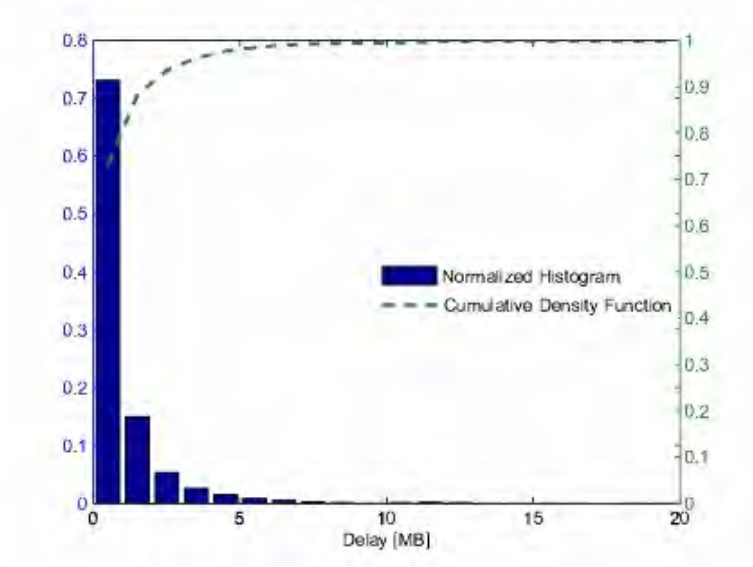


Figure 4.5: Detection distance in MBs for all the errors that SC detected (I frames)

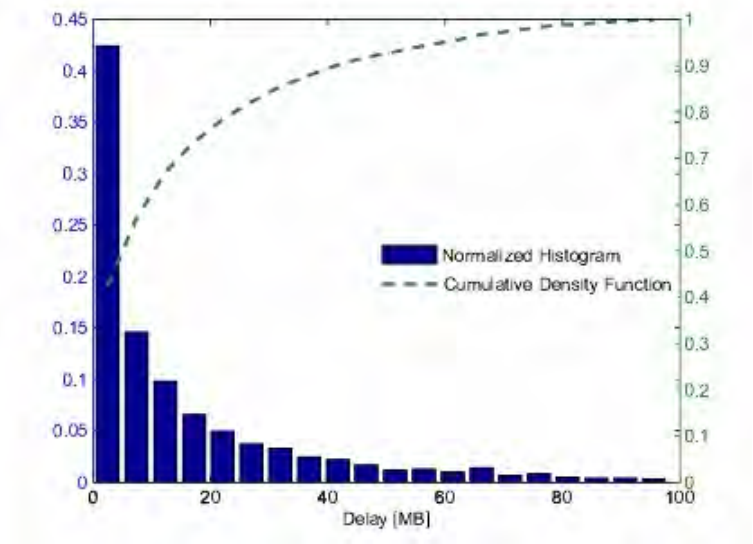


Figure 4.6: Detection distance in MBs for all the errors that SC detected (P frames)

much degradation the undetected errors will introduce. In a simple way, it is possible to quantify this by measuring the distance of the undetected error occurrences to the end of the slice.

The results show that for I frames (Figure 4.7), which are the most important case, since I frame artifacts are the most visible ones, most of the undetected errors are at the end of a slice, while for P frames (Figure 4.8) the distribution is more even, although it still presents a strong peak in errors at the end of the slice.

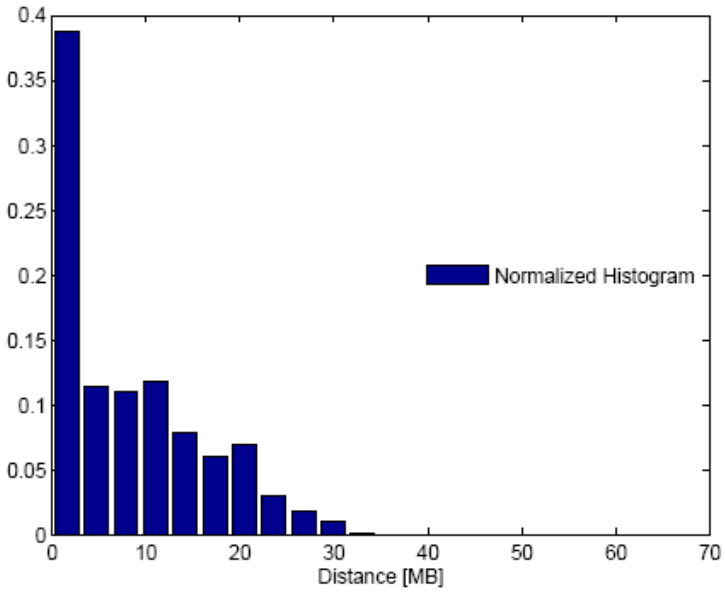


Figure 4.7: Undetected errors: distance between error appearance and end of slice (I frames)

4.1.2 Error concealment

Section 4.1.1 discussed how the SC algorithm detects errors and its detection performance. It was only briefly mentioned that once an error was detected, it would be concealed. This section explains the concealment strategy used in the SC algorithm.

The concealment used in the algorithm is, for simplicity, a zero motion temporal error concealment. It simply replaces each corrupted macroblock in the current frame $MB_f(i,j)$ with the spatially corresponding one in the previous frame $MB_{f-1}(i,j)$. Although more complex concealment strategies could have been used, probably with better results [12], the focus of the algorithms is on rather the detection than the concealment.

This algorithm's task is just to label the incorrectly decoded MBs. Of course afterwards one can use a concealment method more appropriate for a given sequence.

As for the error handling, when deciding to which MBs should the concealment be applied to, there are 3 options, which represent two opposite approaches and one in the middle.

4.1.2.1 Straight decoding - SD

The straight decoding represents the plain decoding strategy where although the errors might have been detected, they are not concealed. It works by assigning the closest valid value to each erroneous parameter. In case this is not possible, it will just perform the decoding process as far as it is possible.

It should be noted that to be able to straight decode, the decoder has to be modified, as in the case of an error it almost invariably crashes. A modified version of the JM, capable of decoding incorrect streams was produced prior to this work (see Chapter 5).

Figure 4.9 shows the output of an I frame decoded by means of SD. The red square marks the error occurrence, and it can be seen how the error propagates spatially until the end of the slice. While the MBs up to the error are correctly decoded, the desynchronization causes

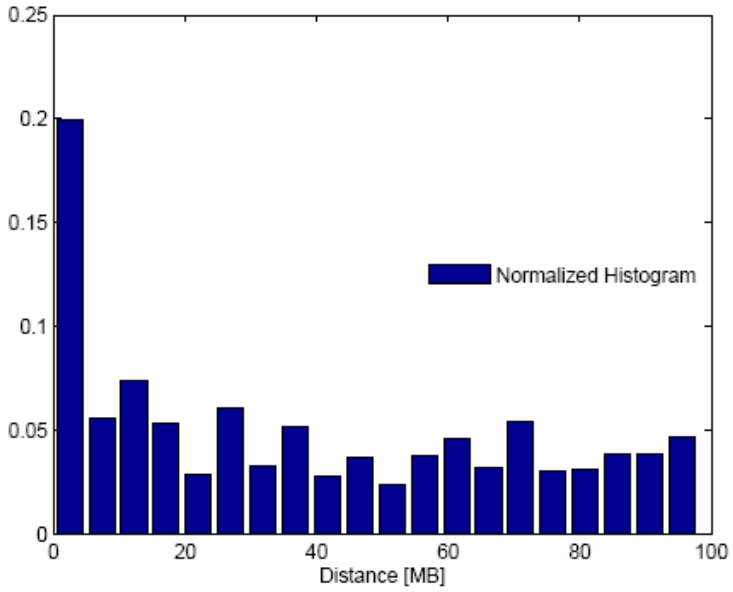


Figure 4.8: Undetected errors: distance between error appearance and end of slice (P frames)

the rest to be decoded incorrectly.

Figure 4.9: Straight Decoded (SD) frame

4.1.2.2 Slice Level Concealment - SLC

This strategy relies on the checksum information provided by the lower layer protocols. For wireless transmission the UDP protocol is used. The checksum information is calculated over the entire NALU and its RTP header. Each error, regardless of its position and effect on the decoding, results in the slice rejection and concealment, from the first MB in the slice to the last, including MBs that are preceding the error.

Figure 4.10 shows a frame concealed using SLC. The error occurs at the same point as in figure 4.9, but as it can be seen, all the MBs in the slice are concealed, including those that were correctly decoded. The green area marks the concealed MBs.

The problem with slice rejection is that only in the case that the erroneous macroblock is the first one, the concealment will replace MBs that are perfectly correct with concealed ones, thus worsening the quality of the picture.

(a) Erroneous frame (b) Concealed part

Figure 4.10: Concealed frame using Slice Level Concealment (SLC)

4.1.2.3 Macroblock Level Concealment

Macroblock Level Concealment (from now on MBLC) works by exploiting the information obtained with the SC analysis. As said, the algorithm is capable of detecting where the bitstream desynchronization has lead the decoder to some illegal action, both in the *READ* or *DECODE* phase. It is capable of detecting the error after a *detection distance*, as shown in figure 4.4.

A more optimum approach than SLC would be to conceal only those MBs that are possibly erroneous. Those MBs are the ones after the error detection.

Figure 4.11 shows the results of using MBLC in the same example as with SD and SLC. The red square shows where the error has occurred, while the green one where SC detected it. So, in this case the detection distance would be 1. While the green area represents erroneous MBs that were concealed, the red area represents an erroneous MB that was not concealed. This strategy allows the use of MBs that can correctly decodes, although because of the detection distance, it leaves a degraded area between the correct MBs and the beginning of the concealment.

(a) Erroneous frame (b) Concealed part

Figure 4.11: Concealed frame using MacroBlock Level Concealment (MBLC)

4.1.2.4 MBLC Performance

Whether using MBLC performs better than SD or SLC entirely depends on the detection distance. Concealing correct MBs introduces a mild quality degradation (proportional to the amount of movement in the sequence, since we use just a copy-paste concealing method), while incorrect unconcealed MBs introduce a strong degradation in the image.

As figure 4.12 shows, MBLC will have a better performance than SD or SLC if the zone of unconcealed erroneous MBs is smaller than the correct but concealed MBs (in terms of quality degradation).

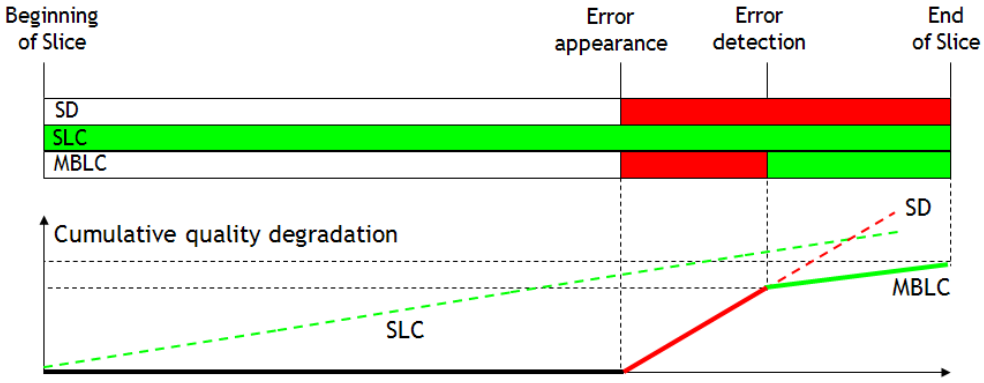


Figure 4.12: Cumulative quality degradation of damaged frames

As shown in figure 4.13 [10], MBLC outperforms SD and SLC, tested for several QP (20, 24 and 28).

4.1.3 SC + MBLC inefficiencies

Although the results for MBLC show that it outperforms SD and SLC, it applies the same policy for both I frames and P frames. The figures showing the concealment of a big artifact in this section, showed an error in an I frame, but as it has been shown in Section 3.2, specially in figure 3.6, an error in an I frame has a different effect than in a P frame.

For I frames the detection distance only means that the artifact will be partially visible. On the other hand, for P frames it means that probably all of the concealed MBs will be consistent. This is because the artifact probably consists of a single MB that was detected too late. So, if MB_k would be erroneous, maybe MB_{k+1} up the end of the slice would be concealed, leaving the single erroneous MB unconcealed.

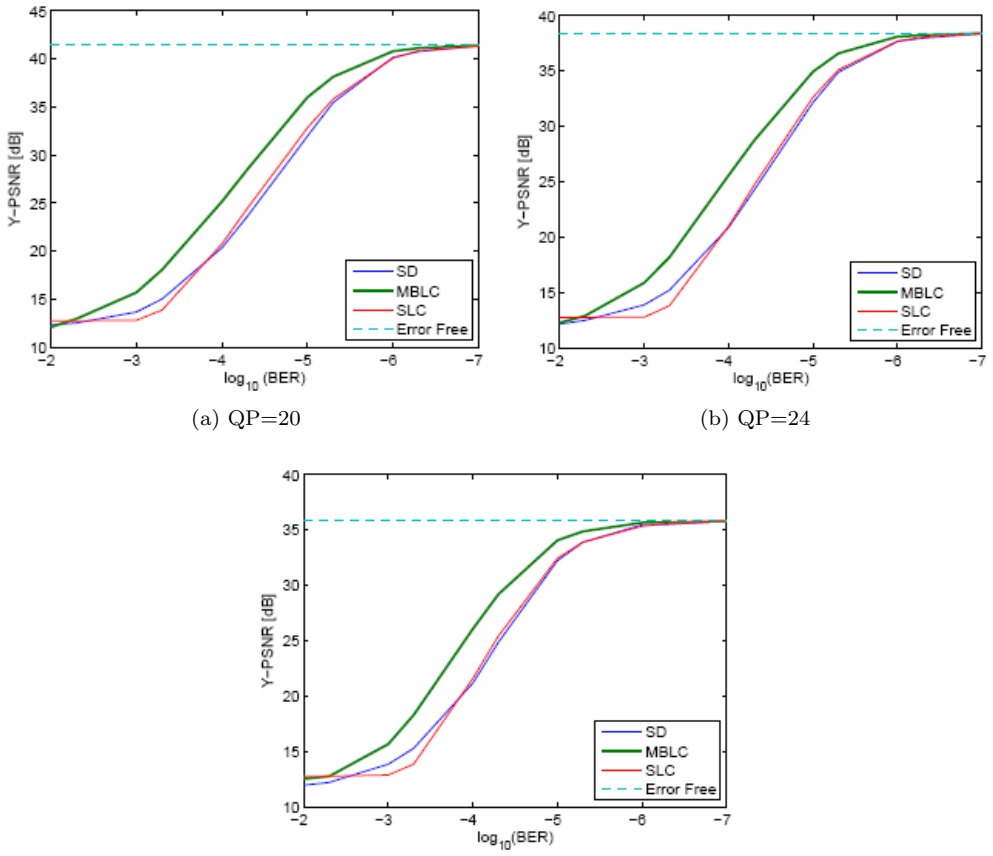
Although the method works well for I frames, the detection distance makes SC not suitable for detecting errors in P frames, as they tend to appear as isolated artifacts instead of propagating until the end of the slice.

This inefficiency means that, for P frames, concealing MBs with MBLC could actually worsen the quality of the image, as correct MBs are being concealed.

Figures 4.14 and 4.15 illustrate why MBLC is not adequate for P frames using a best-case example. In this ideal example, it is assumed that the detection distance is zero, so the error would be immediately detected by the SC algorithm. In Figure 4.14, the error in the I frame is correctly detected and all the MBs after it effectively concealed (green area in 4.14b). Thus, the artifact is effectively concealed.

However, in the P frame, although the error is correctly detected, the algorithm will not only conceal what causes the visual artifact (two MBs), but up to the end of the slice, which because P frames are smaller in size, may mean up to the end of the frame (a whole P frame usually fits in one NALU).

Visual Impairments Detection and Concealment (VIDC), the other algorithm used in this work, was designed to be used instead of SC in P frames and to decrease the detection



(a) QP=20

(b) QP=24

(c) QP=28

Figure 4.13: Performance of SD, SLC and MBLC (Y-PSNR)

distance in I frames.

4.2 Visual Impairments Detection and Concealment

As seen in Section 4.1.3, the use of SC and MBLC has some limitations. Although for I frames the method works well, since errors usually propagate until the end of the slice, the nature of the errors generated in P frames makes using SC less useful for their case.

The objective of the VIDC algorithm is to improve the performance of the error detection and therefore concealment. It aims to improve the detection distance for I frames and to detect only errors that cause visual impairments.

The algorithm [11] is built in the following way (Figure 4.16):

- **I frames:** the algorithms is built on top of SC, only improving its performance. It works by performing an additional analysis in search for visual artifacts, using the detected errors by the SC algorithm in order to make the error detection more precise.

- **P frames:** the algorithm works completely independently.

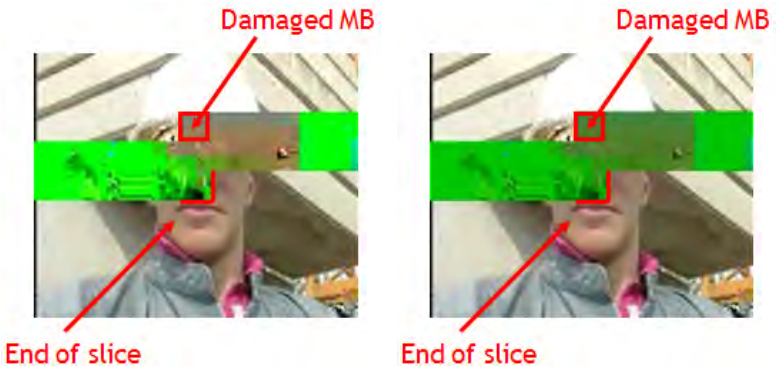


Figure 4.14: MBLC (I frames)

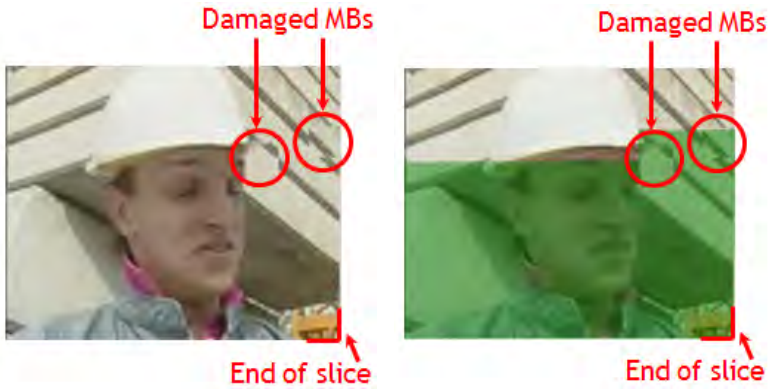


Figure 4.15: MBLC (P frames)

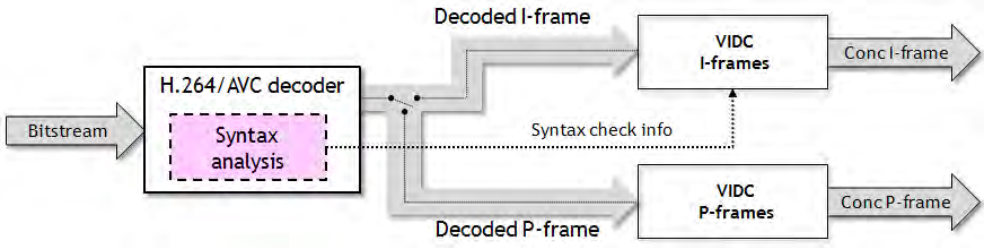


Figure 4.16: Block diagram of the VIDC algorithm

4.2.1 Error detection

While SC worked by analyzing the encoded video stream (actually its approach could be used in any type of decoding in which bitstream desynchronization could occur, not only video), VIDC works by analyzing the SD frame in search for visual artifacts by means of image processing.

As mentioned several times, transmission error manifest in visual artifacts whose visual appearance varies greatly between I and P frames. Although the VIDC algorithm performs

a different search for I and P frames, both methods share some common elements, which conform the basic working of the VIDC algorithm.

Basically, the VIDC detects errors in the following way:

1. It starts by subtracting the current frame (Figure 4.17a) to the previous one (Figure 4.17b), obtaining a difference frame (Figure 4.18a).

(a) $frame_k$ (b) $frame_{k-1}$

Figure 4.17: Frames used in the VIDC algorithm

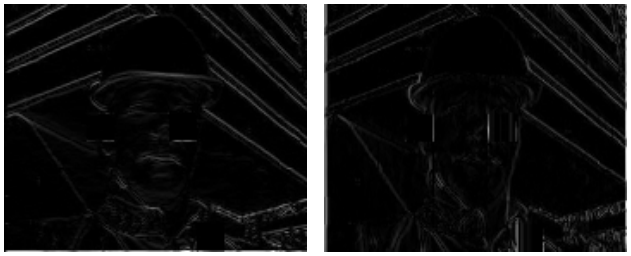
2. It then divides the image in 8x8 blocks (quarter MBs, also referred as sub-MBs, Figure 4.18b) and tries to find blocks in which the difference is high. This highly different blocks are marked as candidates (Figure 4.18c). Since VIDC needs to subtract the current frame to the previous one, the algorithm cannot be applied to the first frame (frame 0), so the first frame is assumed to arrive intact.

(a) Frame difference (b) Frame difference analysis (c) Detected artifacts in the frame difference

Figure 4.18: Difference frame

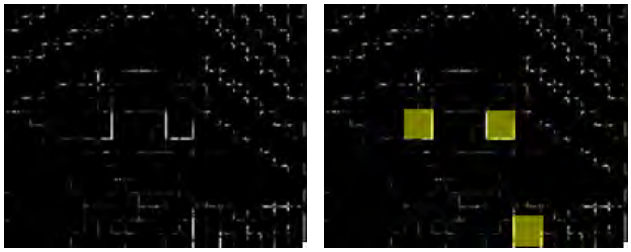
3. The difference of a block with respect to the previous frame (a measure that is called *sub-macroblock power* in the algorithm) is not enough to detect artifacts, so the algorithm also analyzes the edginess of each block in search for the strong edges that characterize the blocky artifacts caused by transmission errors. For this, it filters the current frame using a high-pass filter (Figure 4.19). For this, a simple Haar filter could been used (the implementation uses it).

4. By combining the horizontally filtered (Figure 4.19a) and the vertically filtered image (Figure 4.19b), an image representing the edginess of the 8x8 blocks is formed (Figure 4.19). This image is formed by taking the horizontal edges of the vertically filtered image and the vertical edges of the horizontally filtered image.



(a) Vertically filtered $frame_k$ (b) Horizontally filtered $frame_k$

Figure 4.19: Filtered $frame_k$



(a) Constructed image of the 8x8 block edges (b) Block edges analysis

Figure 4.20: Block edginess

5. The candidates found when analyzing the block power are reevaluated in search of strong edges. When the project started, the VIDC algorithm analyzed only the 8x8 blocks, but this proved to be insufficient for some specific type of artifacts like the one shown in Figure 4.21 left to Foreman's face. Thus, the algorithm was improved in order to consider also the whole 16x16 MB, When the edges of one 8x8 block are not strong enough to be considered as an artifact but the four 8x8 blocks conforming a MB sum up enough edges, the MB will be considered as an artifact. After performing the analysis both at 8x8 block level and 16x16 block level, the algorithm marks the MBs as erroneous (Figure 4.20).

   It outputs a list of 8x8 blocks that thinks that are erroneous. If the 8x8 block was deemed erroneous by itself, the algorithm will list only that exact block (eg. MB 1,2,1,2, meaning the MB in the first row, second column. In that MB, the upper rightmost block).

   As mentioned, the algorithm also performs an edge search at MB (16x16) level. In this case, if the MB was found erroneous, the last two indexes (the ones referring to the sub-block) are not used. So using the last example, that would be MB 1,2.

6. Although for P frames the process stops here, it is not the case in I frames. In P frames, the algorithms searches for isolated artifacts, but since in I frames the artifacts typically propagate until the end of the slice, the extra step consists of a voting system that searches for sequences of erroneous MBs. The decision is made depending on the result of a voting system. The evaluation is performed over a set of characteristics for

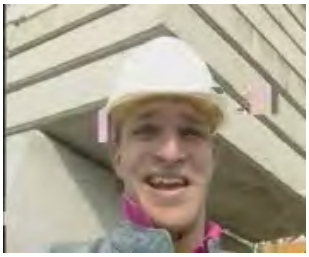


Figure 4.21: Undetected artifact using only 8x8 block analysis in VIDC

sequences of 8x8 pixels block read in raster scan.

As it can be seen in the images,the algorithm uses grayscale images.

Initially, the algorithm analyzed only the luma component of the image. This yielded problems when an error introduced an artifact only in the chrominance components, specially when the generated artifact was a green MB (highly visible), but the luma component of the surrounding MBs was similar to that of the green MB ($Y = 128$). Although the specific problem of the green MBs was tackled in the final implementation (Section 5.5.3.4), it was decided to leave this modification, as it made sense to give some weight to the chroma, as sometimes there are chroma-only artifacts.

In order to resolve this issue, the algorithm was modified to use a mixture of the luma and chroma components. The algorithm detects errors on this "compound". In order to perform the conversion, the following formula has been used:

$$frame_{GS} = 0.6 \cdot Luma + 0.2 \cdot C_b + 0.2C_r$$

4:2:0 sampling is used, so the luma component L is twice the size as the two chroma components C_b and C_r. It is for this reason that we have given the luma component more importance.

Because of the different size of the chroma components, before being able to perform the calculation, both chroma components have to be resized by a factor of 2. The algorithm used for this in the implementation has been the simplest one, nearest neighbor interpolation.

Since we are interested in calculating the average difference and edginess of an 8x8 block, it is not expected that more complex methods such as bilinear or bicubic interpolation would yield better results. Since it may be that they blur the edges of the blocks, there is also the possibility that they decrease the performance of the method, although this remains untested.

The algorithms used for both I and P frames are explained with more detail in Sections 4.2.1.2 and 4.2.1.1.

4.2.1.1 P frame artifact detection

The VIDC algorithm for P frames is depicted in detail in Figure 4.22.

For each MB k in the difference frame, the algorithm divides it in four 8x8 blocks and then calculates its average difference (referred as power in the implementation of the algorithm).

The average block difference is calculated then by performing the average difference of each $k \times k$ block, where ($k = 8$), in the following way:

Input: Straight decoded $frame_k$, errors detected by SC in $frame_k$
Output: Concealed $frame_k$, errors detected by VIDC in $frame_k$

$difference\_frame$ =`abs(`$frame_k - frame_{k-1}$`);`
$avg\_diff = frame_{k-1} \rightarrow avg\_diff$;
foreach *macroblock in* $frame_k$ **do**
  foreach *8x8 block in macroblock* **do**
    `calculate_8x8_block_average_difference(`*8x8 block power*`);`
    if *8x8 block power* > *threshold* **then**
      mark 8x8 block as candidate;
    end
  end
end
`find_edges` ($frame_k$);
foreach *candidate* **do**
  if *8x8 block edginess* > *edginess_threshold* **then**
    mark 8x8 block as erroneous;
  end
  else if *16x16 block edginess* > *edginess_threshold2* **then**
    mark 16x16 block as erroneous;
  end
end
conceal erroneous macroblocks;

Figure 4.22: VIDC P frame algorithm

$$\text{block average difference} = \frac{\sum_{i=1}^{k}\sum_{j=1}^{k} block_{i,j}}{k^2}$$

The threshold used to decide if a certain block will be marked as a candidate is not fixed, it uses both a fixed value and the average difference of the previous frame (referred in the implementation as *diff_frame* or average difference frame). The threshold is calculated as follows:

$$threshold = \max(frame\_diff, 20)$$

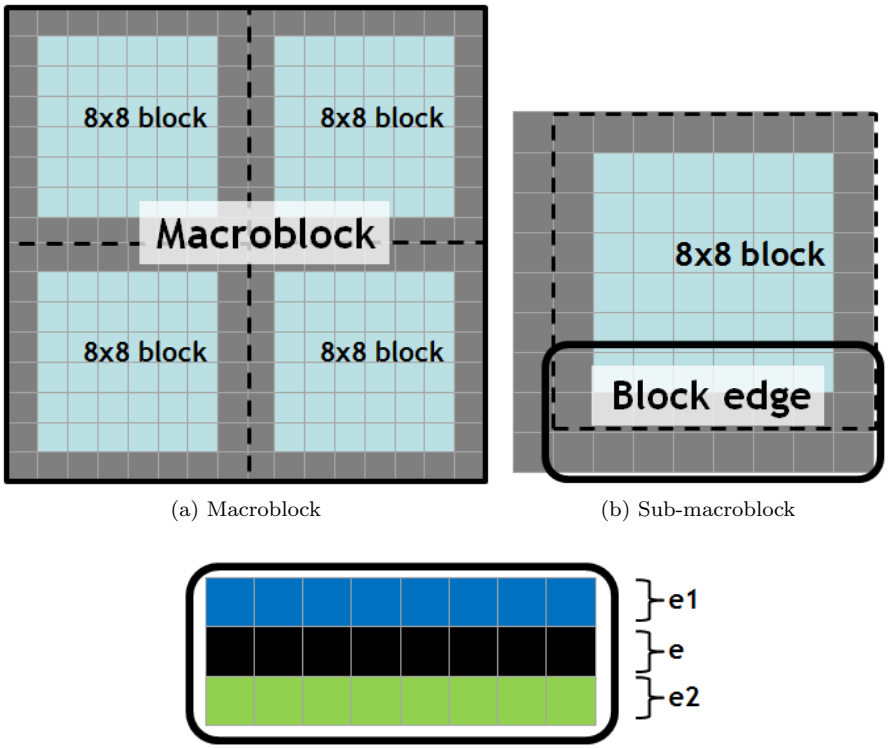
In the case there were errors detected in the previous frame, the average difference frame is the average power of the correct 8x8 blocks. If no errors were detected, it is the average power of all the 8x8 blocks from that frame.

The candidates found only by means of the difference are analyzed in search of edges. As mentioned in Section 4.2.1, the image goes through a 2D high-pass filter.

Before any thresholding occurs, for each block, the algorithm sums the pixels in each edge separately and the line (row or columns) adjacent to each side to the edge. This is done in order to detect how isolated is the edge. The more isolated it is, the more probably it is an error (lines 1 pixel width across an edge are not that common).

Figure 4.23a shows a MB and how it is subdivided in 8x8 pixel blocks. For each sub-MB, each edge is analyzed. In this case, the lowermost edge of the upper-right sub-MB is highlighted. Figure 4.23c depicts the actual frame edge *e*, surrounded by its boundaries *e1* and *e2*. Although in the figure only one MB is drawn, the pixels in e1 and e2 do not need to

be in the same MB as the current MB. In the case there are no such pixels (at the edge of the frame), the edge is not counted.



(a) Macroblock (b) Sub-macroblock

(c) Block edges

Figure 4.23: Macroblock and sub-macroblock edges

Each edge is then assigned a score according the algorithm in Figure 4.24.

if $e/\max(e_1, e_2) > 4$ || $(e/\max(e_1, e_2) > 1.75$ && $e > 400)$ **then**
    mark $= 1.1$;
end
else if $e/\max(e_1, e_2) > 1.75$ **then**
    mark $= 1$;
end
else
    mark $= 0$;
end

Figure 4.24: Edge detection algorithm

For edge blocks (the ones in the border of the frame) the score (sum of the mark of all its edges) is doubled. A sub-macroblock is considered erroneous if its score is over 2 ($score > 2$).

After this, the candidates that have not been considered erroneous undergo a second analysis. The conditions to consider a MB erroneous based on candidates that by themselves would not be considered erroneous requires more conditions. These conditions were written so they would detect previously undetected errors, such as the one shown in figure 4.21 but

strict enough so that they would not include correct macroblocks.

The conditions are as follow:

- The sum of the four biggest edges > 740.
- The average power of the four 8x8 blocks > 11.
- The biggest edge ≥ 260.

In order to avoid the case in which a single very strong edge would skyrocket the value of the sum (one single strong edge in a total of 12 is not considered as an artifact when analyzing the whole MB), when calculating the sum, each edge can count a maximum of 300, so if the edge is greater than 300, it will be counted as a value of 300, not more.

At the end of the edge detection, the result is a list of sub-MBs and MBs that the algorithm believes are artifacts. As the last step, those MBs are concealed using a copy-paste method. This is the same method mentioned in Section 4.1.2, but with the important difference that only the erroneous MBs are concealed, as it has been seen in Chapter 3 that errors in P frames normally do not propagate spatially in a visible way.

4.2.1.2 I frame artifact detection

The reason to explain first the P frame artifact detection is that it shares a lot of common elements with the I frame algorithm, the only big difference being the concealment and that for I frames there is a voting system that searches for damaged sequences of MBs. The I frame detection algorithm can be seen in Figure 4.25.

In detail, the difference between the already explained P frame detection algorithm and the one used for I frames are as follow:

- The candidates outputted by the edge detection are not directly marked as erroneous, but instead are a second type of candidate.
- After the edge detection is finished, all of the 8x8 blocks are analyzed in raster order. Depending on its power and whether they were marked as a candidate in the previous detection step, a vote is assigned to a sequence.
- At the end of the slice, if the vote is high enough or if an error was detected either by SC or the edge detection, the algorithm will determine that there is an artifact in the slice and will try to find its beginning.
- After the voting system, the algorithm also outputs a list of sub-MBs, but its meaning is different than for P frames. Those MBs are the positions in the slice where an an artifact was detected, and mean that subsequent MBs up to the end of the slice need to be concealed, as they will very probably show spatial error propagation.
- In the last step, where the artifacts are concealed, the algorithm conceals the MBs since the error occurrence propagates to the end of the slice, as opposite as in P frames.

The voting system algorithm is shown in Figure 4.26.

Input: Straight decoded $frame_k$, errors detected by SC in $frame_k$
Output: Concealed $frame_k$, errors detected by VIDC in $frame_k$

$difference\_frame$ =`abs(`$frame_k - frame_{k-1}$`)`;
$avg\_diff = frame_{k-1} \rightarrow avg\_diff$;
foreach *macroblock in* $frame_k$ **do**
  foreach *8x8 block in macroblock* **do**
    `calculate_8x8_block_average_difference(`*8x8 block power*`)`;
    if *8x8 block power* > *threshold* **then**
      mark 8x8 block as $difference\_candidate$;
    end
  end
end
`find_edges(`$frame_k$`)`;
foreach *difference_candidate* **do**
  if *8x8 block edginess* > *edginess_threshold* **then**
    mark 8x8 block as $error\_candidate$;
  end
  else if *16x16 block edginess* > *edginess_threshold2* **then**
    mark 16x16 block as $error\_candidate$;
  end
end
evaluate $error_candidate$s (voting system, see figure 4.26);
foreach *slice in* $frame_k$ **do**
  conceal from the first erroneous macroblock up to the end of the slice;
end

Figure 4.25: VIDC I frame algorithm

4.2.2 Error concealment

As mentioned in Section 4.1.3, MBLC works well for I frames but showed the inefficiency of concealing correct MBs in P frames.

Figures 4.14 and 4.14 showed very well how MBLC was not the optimal approach for P frames. Since the VIDC algorithm directly detects visual artifacts in the pixel domain, using this algorithm concealment of only erroneous artifacts can be done.

Since MBLC works partially well, it needs only to be partially replaced with another concealment strategy. After the addition to the VIDC algorithm, the concealment strategy will remain as follows:

- I frames: conceal erroneous MBs and the following ones up to the end of the slice.
- P frames: conceal only the erroneous MBs.

```
foreach slice in frame_k do
    foreach 8x8 block do
        if block has syntax-error then
            syntax_error = 1;
            find first 8x8 block in sequence where vote > vote_threshold;
            mark that block as erroneous;
            STOP looping blocks for this slice;
        end
        else if block is error_candidate then
            vote++;
        end
        else if block power > power_threshold then
            vote+;
        end
        else if block power < power_threshold2 then
            vote-;
        end
        else
            vote–;
        end
    end
    if vote > vote_upper_threshold then
        visual_impairments_error_found = 1;
        find first 8x8 block in sequence where vote > vote_threshold;
        mark that block as erroneous;
        STOP looping blocks for this slice;
    end
    else if vote < vote_lower_threshold then
        clear_sequence;
    end
end
```

Figure 4.26: Voting system for I frame artifacts detection

Chapter 5

Algorithm implementation

In Chapter 4, SC and VIDC (the two interacting algorithms used to detect and then afterwards conceal the errors) were presented. Although quite detailed, the description of the algorithms included only small amounts of information about the actual implementation.

5.1 The JM and JM+SC decoder

The starting point was the JM 10.2 H.264/AVC decoder in which SC analysis had already been embedded (from now on JM+SC decoder) and a VIDC algorithm written in Matlab code. The objective was to embed the VIDC code in the JM decoder so it could interact with SC and the decoder's picture buffer. By doing so, it would be possible to use VIDC in real applications. Since the JM decoder is written in C code, the VIDC algorithm needed not only to be embedded in the decoder but also translated from Matlab to C.

The JM reference software [13], as it name implies, is a reference implementation, including source code, of the H.264/AVC codec. It provides all the capabilities defined in the standard as well of providing proof that such capabilities are feasible to program. As a reference implementation, it is not a commercial codec (it is provided cost-free). Its focus is not on code optimization (its performance in terms of coding speed lack behind considerably compared to commercial codecs) but rather providing a tool for developers. As such, it complies with the standard in the most complete way, so it can be used as a testbed of all the H.264/AVC features.

The standard JM 10.2 decoder is capable of reading and decoding H.264/AVC bitstreams (Figure 5.1). It has been simplistically represented as having a *READ* function, where the bistream is read and translated to symbols, and a *DECODE* phase where those symbols are interpreted and the picture is decoded. This which is done also interacting with a decoded picture buffer, where the previous pictures are stored. In our case, only the last picture is stored, since a buffer size of one has been used.

The task of implementing SC was already done before this project, and a JM 10.2 + SC decoder was already working and available (Figure 5.2). The SC is basically located in the `read_new_slice`, `decode_one_macroblock` (and its called functions) and `decode_one_slice` functions. For a detailed call graph of these function, please refer to the JM documentation [1]. The SC algorithms is programmed to do what in more modern programming

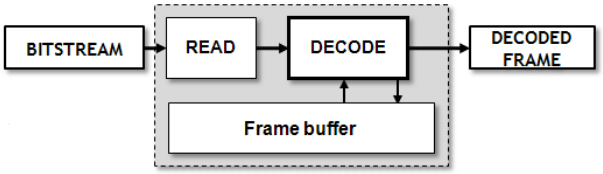


Figure 5.1: Standard decoder

languages would be a `try-catch` structure.

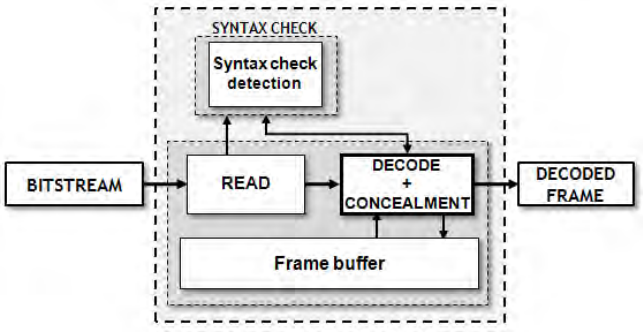


Figure 5.2: Standard decoder + SC

Whenever an error occurs in the *READ* phase, a variable is used to mark that an error has occurred and the read process is stopped. Then, the *DECODE* process is called. The decoding process will try, with whatever data is available to decode the image from the data. If an error occurs during the *DECODE* phase, a variable is used to mark that an error has occurred and the decoding process is stopped.

It should be mentioned that the standard JM decoder is not prepared to handle erroneous bitstream, and will almost invariantly crash when exceptions arise. For this, the JM decoder that has been used was modified to be error-tolerant.

Other additions were implemented in the decoder in order to be able to test the algorithm. Mainly, the ability to introduce errors in the bitstream in several ways and the option to turn on and off the algorithm (the latter needs recompilation).

Figure 5.3 shows the configuration file of the JM+SC decoder. The only lines that differentiate it from the standard decoder are the last two ones. In those lines, it can be specified which error mode will be used and, in case it is needed, where is the file that specifies where the errors are located in the bitstream.

In the JM+SC, the available error modes are:

- `error_mode=0`: the errors to be inserted are located in an error file (in the example, `error.txt`). The error file consists of a list of bit positions in which the errors will be inserted at NAL level. It is explained more in detail in Section 5.1.1. In the case the specified error file does not exist or is empty, no errors are inserted.

- `error_mode=1`: the algorithm will introduce an error in a random position in each slice. It works in this way because the error inserting code is located in the `read_new_slice` function.

```
foreman_700b_10i_1l_50f.264     ........H.26L coded bitstream
for_dec_700b_10i_1l_50f.yuv     ........Output file, YUV/RGB
foreman_QCIF_420.yuv  ........Ref sequence (for SNR)
1                     ........Write 4:2:0 chroma components for monochrome streams
1                     ........NAL mode (0=Annex B, 1: RTP packets)
0                     ........SNR computation offset
2                     ........Poc Scale (1 or 2)
500000                ........Rate_Decoder
104000                ........B_decoder
73000                 ........F_decoder
leakybucketparam.cfg  ........LeakyBucket Params
0                     ........Err Concealment(0:Off,1:Frame Copy,2:Motion Copy)
2                     ........Reference POC gap (2: IPP (Default), 4: IbP / IpP)
2                     ........POC gap
0                     ........Error Mode
error.txt             ........Error File

This is a file containing input parameters to the JVT H.264/AVC decoder.
The text line following each parameter is discarded by the decoder.
```

Figure 5.3: JM configuration file

- error_mode=2: no errors will be inserted.

5.1.1 Error insertion

The error location file determines where the decoder will insert errors. This is useful in order to be able to decode error patterns in a reproducible way. An error file contains only a sequence of bit positions that point where an error will be inserted. To illustrate this example, an error file which will insert an error in the bit position 122652 will be used. For debugging purposes, the implementation supports up to 100,000 error insertions.

In order to be able to know the correspondences between a position in a frame, slice and MB and its bit counter (the bit position in the bitstream), the encoder or decoder trace file must be used (recompiling the decoder with `#define TRACE 1` in `defines.h` may be necessary). In Figure 5.4, the trace file from one of the erroneous video sequences that have been used can be seen.

The trace file contains a complete transcript of the read bitstream and the symbols that correspond to each codeword. Figure 5.5 is the trace file of the same decoded sequence, but after the error had been inserted.

The error location file specifies an error in the bit position 122652, which in Figure 5.4 is instruction `intra4x4_pred_mode`, which has a value of 0000 in binary.

After the error has been inserted (Figure 5.5), position 122652 has the value 1 instead of 0000, This is due to the variable length coding. In fact, the bitstream was changed from 0000 to 1000. In the same way as the example in Chapter 3's Figure 3.4, due to this change the boundaries between the codewords are changed. In the trace file, it can be seen that although the bits after the error have not changed the codewords are different.

The result of this error is the damaged frame in Figure 5.6. The artifact starts in MB 44, the same one in which the error was inserted (in the trace file, the first MB is MB zero). Nonetheless, the MB in which the error was inserted may not be affected by an artifact.

```
[...]
********** POC: 50 (I/P) MB: 44 Slice: 2 Type 2 **********
@122609 mb_type                                                1  (  0)
@122610 intra4x4_pred_mode                                     1  (  2)
@122611 intra4x4_pred_mode                                     1  (  2)
[...]
@122644 intra4x4_pred_mode                                  0011  (  4)
@122648 intra4x4_pred_mode                                  0011  (  4)
@122652 intra4x4_pred_mode                                  0000  (  0)
@122656 intra_chroma_pred_mode                                 1  (  0)
@122657 coded_block_pattern                                  010  ( 31)
@122660 mb_qp_delta                                            1  (  0)
@122661 Luma # c & tr.1s vlc=0 #c=4 #t1=1                000000110  (  6)
@122670 Luma trailing ones sign (0,0)                          1  (  1)
[...]
```

Figure 5.4: Example trace file

```
[...]
@122644 intra4x4_pred_mode                                  0011  (  4)
@122648 intra4x4_pred_mode                                  0011  (  4)
@122652 intra4x4_pred_mode                                     1  (  4)
@122653 intra_chroma_pred_mode                           0001010  (  9)
@122660 coded_block_pattern                                    1  ( 47)
@122661 mb_qp_delta                                0000001101000  ( 52)
@122674 Luma # c & tr.1s vlc=0 #c=9 #t1=3            00000000100  (  4)
[...]
```

Figure 5.5: Example trace file with inserted error

Sometimes the desynchronization manifests itself in the following MB/s.

Figure 5.6: Artifact generated by the error pattern

Since now all of the configuration parameters are in the configuration file, it is really not necessary to input parameters into the decoder. So, for a config file named `decoder.cfg`, the call line would be as follows:

```
ldecod decoder.cfg
```

5.1.2 Output files

A part from the files that are outputted by the standard JM decoder, the JM+SC decoder adds some other useful output, always in Comma Separated Values (CSV) format:

- `err_pos.txt`: lists the positions in which the errors have been inserted. The position includes bit position, frame number and MB number.
- `det_pos.txt`: the same as `err_pos.txt`, but it lists the detected errors (by SC) instead of the inserted ones.
- `mb_slice.txt`: contains a sequence of numbers that are the size of all the slices in the video sequence.

5.2 Matlab implementation

Starting from the JM+SC decoder and the VIDC algorithm in Matlab, the objective was to first translate the Matlab code to C and then embed it in the JM+SC decoder. In order to do that, the code would need to interact with the frames stored in the Decoded Picture Buffer (DPB) and modify the information stored there, communicate with the SC algorithm and be able to output debug data.

The starting point was a standalone Matlab application, completely separated from the decoder. Figure 5.7 shows how the Matlab implementation with which the project started worked.

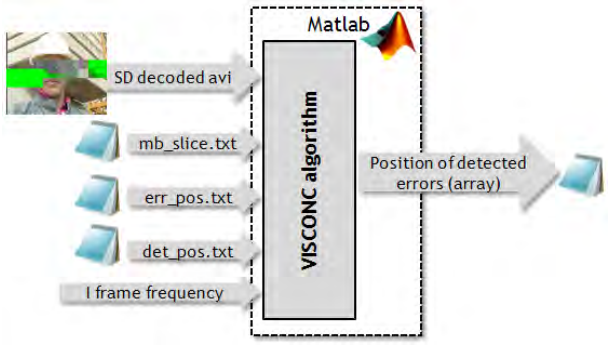


Figure 5.7: Matlab VIDC algorithm

The most important difference between the new C implementation and the Matlab one is that the Matlab-implemented VIDC did not provide concealment capabilities, just artifact detection. Of course one frame could be concealed, but since the application worked isolated from the decoder, the newly concealed frame obviously could not be stored in the DPB.

The Matlab program works by reading the following files:

- **Straight decoded avi file:** the program accepts only avi files as input, which presents a conversion problem, since the H.264/AVC JM decoder outputs only raw yuv files. So every file needed to first be converted to avi in order to be able to be analyzed by the algorithm. During the project that proved to be too cumbersome, so the original Matlab code was modified so it also accepted raw yuv files as input. Also, since the

avi is stored in RGB, in order to be processes it needs to be first converted to the YUV colorspace, which took also some time (small though). From the three YUV components, only the luma component is used in the algorithm.

- **Slice size file:** the `mb_slice.txt` file that the JM+SC decoder outputs. The algorithm needs this file to know the exact size in MBs of each slice.
- **Error position file:** actually this file (outputted `err_pos.txt` from the decoder) is not really necessary, as it is not needed to perform the detection (the algorithm would not normally know the information contained there). Nonetheless, it is provided to the algorithm for debug purposes.
- **Detected errors file:** these are the errors detected by SC. The I frame detection algorithm needs them to evaluate the sequences in the voting system. It is the outputted `err_det.txt` file from the decoder.
- **I frame frequency:** it is a variable that tells the algorithm which frames are I frames and which are P frames. Since the avi file does not contain this information, it needs to take it from another source. Depending on this variable, the algorithm will call the `detect_i` or `detect_p` functions for a given frame.

As output, the Matlab program gives an array containing a list of positions of detected errors. Figure 5.8 shows how this list looks like. Afterwards, this list can be dumped to a text file for storage or further analysis.

```
0     2     0    88     8     0
0     2     0    97     8     9
0     3     0    94     8     6
0     4     0     7     0     7
0     4     0    62     5     7
0     4     0    88     8     0
0     5     0    98     8    10
0     6     0    10     0    10
0     6     0    37     3     4
0     7     0    98     8    10
```

Figure 5.8: Matlab output of the VIDC algorithm

Matlab indexes array assigning the position `1` to the first element (Matlab uses one-based indexing), so each time a MB position is mentioned it should be noted that the range is [1,99] (99 MBs in total). The output file has the following structure:

`0 (padding)  frame_number  0 (padding)  MB_number  MB_row  MB_column`

Although the algorithm internally detects erroneous 8x8 blocks, since it conceals at MB level, it outputs only the MB positions. It should be noted that `MB_number` can be obtained easily as $11 \cdot$(`MB_row`-1) + `MB_column`.

5.3 Implementation process

Implementing the VIDC algorithm directly into the JM would have proved a very complex task and would have also produced a very buggy code. In order to make the implementation

easier, the process was split into parts, at the end of which the code would be tested for errors. The idea was basically to separate the implementation and integration parts of the project. So, only once the code was sure to perform its task it would be integrated into the JM+SC decoder.

The implementation process was divided into the following 6 steps, as to make the integration as easy as possible:

1. Generate standalone C application that works in the most similar way as the Matlab application. Feed both applications and check if the output is equal (Section 5.4).
2. Integrate the code into the JM+SC decoder. Check again if the output is still consistent. This includes having to decide where in the JM+SC code should the new code be located and how to interact with the already existing parts of the program (Section 5.5.
3. Add concealment capabilities to the algorithm.
4. Debug the application. Each part and after the whole application is tested in order to discover bugs, crashes and any abnormal functioning.
5. Test the performance of the finished JM+SC+VIDC H.264/AVC decoder.
6. If the results are not satisfactory, improve the algorithm. And test the algorithm again (this is when the artifact detection at MB level, not only at sub-MB level was introduced, see Section 4.2.1).

5.4 Prototype C implementation

The first C implementation of the VIDC algorithm is shown in Figure 5.9. The diagram is very similar to that in Figure 5.7 because they are meant to be equivalent programs, the Matlab one serving as a reference to test if the output from the C implementation is correct.

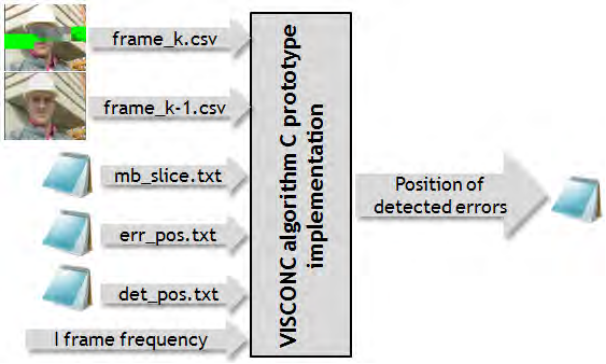


Figure 5.9: C prototype VIDC algorithm

The video is now fed as independent frames due to the fact that writing code to read an avi file and store it in memory, a part from consume time, would not be useful for the final implementation, as the code is intended to read frames from the decoded picture buffer and process them in this way. For simplicity, some test frames were outputted in CSV format from the Matlab implementation and used to test the C VIDC algorithm. Only two

frames are needed at any given time: the frame to be analyzed ($frame_k$) and its previous frame ($framek-1$), used to get the difference frame ($|frame_k - frame_{k-1}|$). The goal of this implementation was: being fed the same data, the outputs of both the Matlab and C implementations to be identical (Figure 5.10).

The Matlab implementation was designed to work with text files that contained information about the whole video sequence. This is because the algorithm was fed the files outputted by the JM+SC decoder. Since the C implementation works on a frame basis, the way the SC information was accessed had to be changed. The `detecti` and `detectp` functions were designed with this in mind, and access the information in a way that is more coherent for usage inside the decoder. Now these functions use data structures that contain SC data for only the current and previous frame. In this way, the embedment of this code in the JM (Section 5.5 is easier.

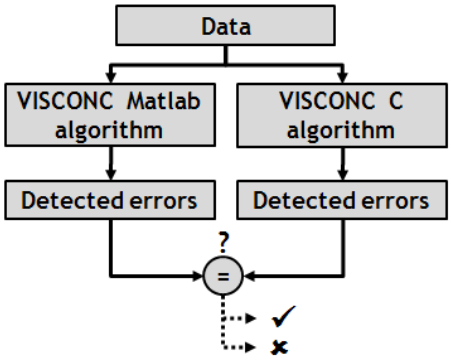


Figure 5.10: C prototype testing

The Matlab code (around 30 KB of source code), when translated into the C prototype implementation turned into 45 KB, 1700+ lines of code).

This explosion in size makes the C implementation more error-prone, as the bigger the program the easier it is that it contains errors, but it could not be avoided. The reason for the C code being much bulkier than the Matlab one is that Matlab is designed for easy matrix manipulation (its name stands for MATrix LABoratoy), and image frames are, in the end, nothing more than a matrix of pixels, composed of one luma and two chroma components (in the case a YUV colorspace is used, as in H.264).

The first necessary step was to understand the totality of the VIDC Matlab code. The alternative of not doing an as-literal-as-possible translation was considered, but in the end discarded, as it would make comparing both source codes in search for inconsistencies more difficult. Also, in this way the original author of the VIDC algorithm would be able to rapidly understand the new implementation.

The biggest difference and difficulty to overcome during the translation between the Matlab and C programming languages is that Matlab is weakly typed, while C is strongly typed. The other is that, while in Matlab it is possible to perform operations directly with matrices and arrays, in C there is no way to avoid having code that goes through every element using `for` or `while` loops and functions that perform specific tasks, such as performing a convolution (filtering an image, for example). Another issue that gave some problems was having to change from one-based to zero-based array indexing.

In a strongly typed language (compiled computer languages) like C, C++, Java and

```
//Define struct

typedef struct block
{
   float diff;
   float is_error;
   int i;
   int j;
   int i1;
   int j1;

} Block;
```

(a) Definition (header file)

```
// Declare and then use struct

[begin of the function]
Block block[9*11];

[...]

[codeblock]
block[0].i = 2;
block[0].j = 3;
block[0].i1 = 0;
block[0].j1 = 1;
block[0].diff = 35;

[...]
```

(b) Variable use

Figure 5.11: Strongly typed language example (C)

Pascal, the data type of every variable, parameter and function return value is known at compile time. The programmer has to provide the type information through declarations. This is known as static typing. Figure 5.11 shows an example of a simple operation such as assigning some values to a struct. In C, the struct first needs to be defined in a header file and then, before it can be used, declared (in C not in any part of the code, must be at the beginning of the function) and then used. In the example, the struct is located in an array, and the struct that is to be assigned the values is located in the first position (indexed as position 0 in C).

On the other hand, in languages with dynamic data typing, like Lisp, Perl, Python, Ruby, and Matlab, the data types are not (wholly) declared on variables. The type of the data associated with variables are not known until the point of execution. The advantage of dynamic data typing is more flexibility and less work for the programmer. But often the data type declarations help in organizing and understanding a program.

Figure 5.12 is the equivalent in Matlab for the code in Figure 5.11. In Matlab, the struct does not need to be either defined or declared. Not even because of it being in an array. The variable assignment can be directly typed. Extra variables could even be added to the struct (with some restrictions, though) a posteriori.

```
block(1).i = 2;
block(1).j = 3;
block(1).i1 = 0;
block(1).j1 = 1;
block(1).diff = 35;
```

Figure 5.12: Weakly typed language example (Matlab)

For ease of programming, each Matlab structure or array has been literally translated into C, taking into account that Matlab uses one-indexing while C uses zero-indexing when programming the access to these structures.

This prototype implementation consisted of 22 functions and 14 different types of data structures. There is no DoxyGen documentation available for this version, but the JM+SC+VIDC one is fairly equivalent as to its structure. A brief explanation of the implementation will be given here.

The standalone C VIDC implementation consisted of the following files:

- A simple `main.cpp` file that read the csv-stored frames and had its slice sizes and frame type (I or P frame) hard-coded. Since not many test frames were used this was possible to accomplish. The application was programmed so all the output was done through the screen. Since only one frame was tested every time, not that many errors are detected in each pass.
- `detector.h`: the definition of all the structures that are used in the program and all of the function prototypes.
- `detector.c`: all of the logic from the algorithm is located here.

The `main` function works as just a wrapper for the logic of the program, which is located in the `detect_i` and `detect_p functions`.

Below is a list of the most important functions of the program and its functions:

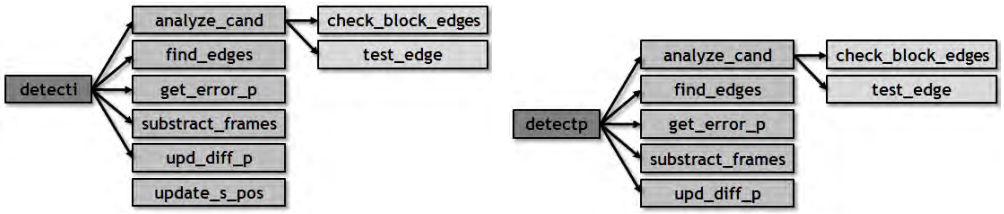
- `init_detector`: initializes all the variables needed to perform the detection. This is especially important, since in C, if this step is not performed, unexpected and difficult to trace unexpected behaviors could arise from memory positions with information left from previous calls.
- `detecti`: contains the code that performs the detection of artifacts in I frames, including the voting system (the algorithm depicted in Figure 4.25). It outputs a one-indexed list of detected errors, which means MB 1 is the first MB, not MB 0. This was done so the output of the Matlab and C implementation would be identical.
- `detectp`: contains the code that performs the detection of artifacts in P frames (the algorithm depicted in Figure 4.22. It also outputs a one-indexed list of detected errors.
- `substract_frames`: substracts two frames and returns the equivalent to Matlab's $|frame_k - frame_{k-1}|$.
- `find_edges`: performs horizontal and vertical filtering using a 2x2 Haar filter
- `get_error_i`: detects error candidates based on the difference calculated by `substract_frame` in I frames. This function works in an almost identical way as the `get_error_p`, but has separate threshold values.
- `get_error_p`: detects error candidates based on the difference calculated by `substract_frame` in P frames.
- `analyze_cand`: analyzes the information from `find_edges` and decides which 8x8 blocks will be considered erroneous.
- `test_edge`: called from `analyze_cand`. Checks an individual edge and quantifies it as an artifact (the sum of these values is then used by the `analyze_cand` to decide if the block (MB level analysis was not implemented yet) is erroneous or not.

- `check_block_edges`: called from `find_edges`. Calculates the edge values e, e1 and e2 for each 8x8 block based on the filtered image. For how these values are calculated see Section 4.2.1.1.

- `upd_diff_p`:calculates the `diff_frame` (explained in Section 4.2.1.1) value for the current frame. This function is called at the end of the detection process, and calculates the value that will be used in the next iteration.

- `update_s_pos`: used in the I frame voting system. It stores the artifact sequences and keeps track of the vote of each MB.

Figure 5.13 shows a simplified (only the most important functions are called) call graph for both the I frame and P frame VIDC implementation.



(a) I frame call graph (b) P frame call graph

Figure 5.13: VIDC call graphs

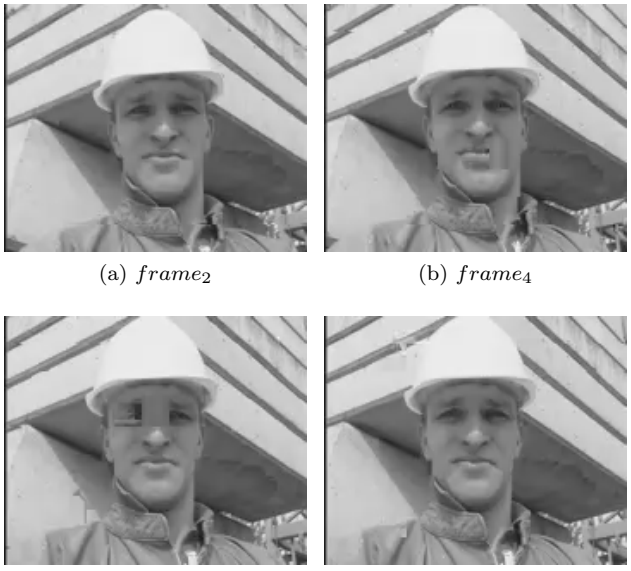
After a lengthy debugging process in which many implementation errors were found, most of them related to the zero-indexing issue, the prototype implementation finally yielded the same results for a random set of errors in I and P frames. Since only 4 frames were used for this early testing, some errors went undetected, but were successful detected in the next implementation phase (embedment). Figure 5.14 shows the 4 luma frames used for the testing. These frames are respectively frames 2, 4, 6 and 8 of the foreman sequence.

It should be strongly emphasized that, since this project deals only with QCIF video sequences, the implementation was done specifically for this video resolution. Making it flexible enough for use in any resolution would have required extensive use of dynamic memory allocation and produced more errors while programming, making the debugging phase longer. It is because of this that the use of QCIF resolution is hard-coded and cannot be changed.

Although the code contains the `#define WIDTH 176` and `#define HEIGHT 144` lines, changing those will probably just make the program inoperative, as the rest of the algorithm is not prepared to handle another frame size other than 176x144 pixels.

5.5 JM-embedded VIDC C implementation

After the implementation of a C VIDC algorithm, the following step is to interconnect the algorithm with the JM+SC decoder. In order to do this, all the inputs for the VIDC algorithm had to be replaced by memory accesses. For this, a way to access the decoded picture buffer had to be found. File (instead of screen) output and code that would call the VIDC functions from the JM+SC code had to be programmed. Naturally, this phase involved also extensive debugging. Also, some errors in the SC implementation that led to some unexpected behaviors were detected and had to be corrected.



(a) $frame_2$ (b) $frame_4$

(c) $frame_6$ (d) $frame_8$

Figure 5.14: Frames used to test the VIDC C implementation

After the translation into C and embedment, the resulting JM+SC+VIDC decoder is shown in Figure 5.15. As it can be seen, the VIDC block is situated just after the *DECODE* phase. This is due to the VIDC algorithm need of decoded frames (it works in the pixel domain).

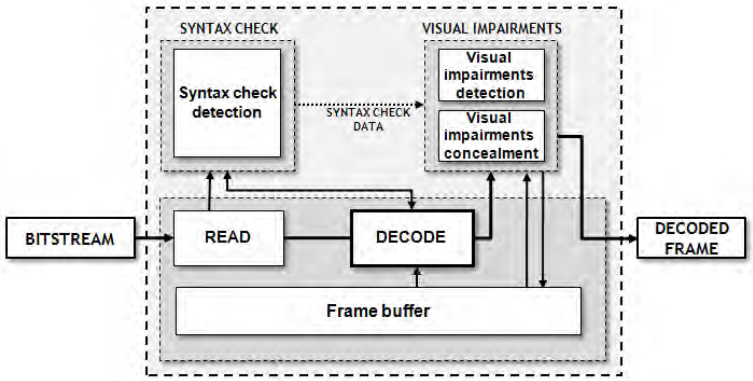


Figure 5.15: Standard decoder + SC + VIDC

The embedded VIDC algorithm reads the slice data and detected errors (SC) data directly from the same variables as the SC code and reads the video frames directly from the currently decoded picture and the picture buffer. Concealment was also added at the end of the detection phase, overwriting the currently decoded picture, which is subsequently stored in the decoded picture buffer and then outputted.

5.5.1 Data input

The input to the algorithm had to be modified so the data could come from memory instead of from a file. Because of the way the prototype implementation was done, this was a smooth transition. The functions were already prepared to be fed only the relevant information. Although this made it necessary to input data by hand and make some conversions while testing the first implementation, it was all made for the sake of simplicity in the final implementation. This can be observed by looking at the parameters that the `detecti` and `detectp` functions take (both functions take exactly the same arguments, so only the `detecti` function is shown:

```
int detecti(
    unsigned short frame[][WIDTH],       //the current frame
    unsigned short frame_prev[][WIDTH],  //the previous frame
    unsigned short det_pos[][WIDTH/16],  //errors detected by SC so far
    SliceData *slice_dim,                //slice dimensions of this frame
    int idx,                             //frame number
    unsigned short difference[][WIDTH]); //where to store the difference frame
```

In order to achieve connectivity, the algorithm had to be slightly modified. Figure 5.16 shows how is the data inputted into the codeblock that contains the VIDC logic.

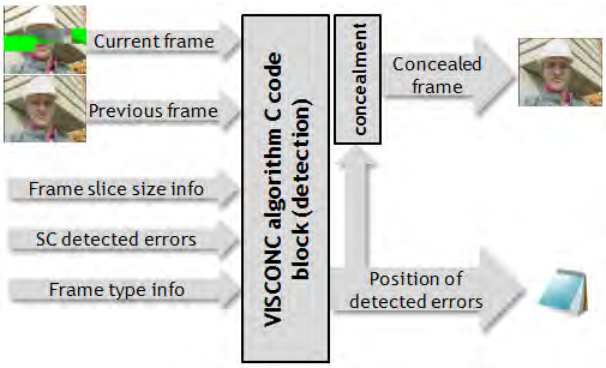


Figure 5.16: VIDC part in the JM decoder

The data needed for the VIDC is obtained in the following way:

- Currently decoded frame ($frame_k$): Since what the algorithm uses is not only the luma component, all three components have to be retrieved and combined in the proportion stated in Section 4.2.1 ($0.6Luma + 0.2C_b + 0.2C_r$). To obtain this, the following piece of code is used (in `image.c`, function `exit_picture`, `dec_picture` is a global variable):

```
unsigned short **frame_y;
unsigned short **frame_u;
unsigned short **frame_v;
unsigned short frame_gs[144][176];

frame_y = dec_picture->imgY;
```

```
frame_u = dec_picture->imgUV[0];
frame_v = dec_picture->imgUV[1];

frame2rgb(frame_y,frame_u,frame_v,frame_gs,y_weight,u_weight,v_weight);
```

Where `y_weight`, `u_weight` and `v_weight` are 0.6, 02 and 0.2, respectively.

The currently decoded frame is located in a `StorablePicture` struct, which in turn contains (among other info), the `imgpel ** imgY` matrix and the `imgpel *** imgUV` array of matrixes. `imgpel` is just a `typedef` for the `unsigned short` C type. So, `imgY` contains a 176x144 (the size is dynamically allocated, but for QCIF, this is the size of the array) matrix of `unsigned ints` (one byte each), with each position representing a pixel. `imgUV[0]` and `imgUV[1]` contain the U (C_b) and V (C_r) components, each 88x72 bytes big due to the 4:2:0 sampling.

The `frame2rgb` function (incorrectly named, since what it actually does is to convert from YUV to grayscale, but it was named this way and like this it remained...) rescales the U and V components by a factor of two and weight averages the three components. The result is written in the `frame_gs` matrix.

- Previously decode frame ($frame_{k-1}$): in order to be able to retrieve a picture previously stored in the decoded picture buffer, the code had first to be studied in search of where were the previously decoded frames exactly stored, since no information on this topic was available in the decoder documentation. After some research through the JM source code, the way to retrieve the previous frame was found. The following piece of code retrieves $frame_{k-1}$.

```
StorablePicture *prev_frame;
int frame_to_retrieve = dpb.used_size-1;

prev_frame = (dpb).fs[frame_to_retrieve]->frame;
```

The frame is retrieved from the `DecodedPictureBuffer dpb`, which in turn contains an array of `FrameStores` (`FrameStore ** fs`), containing each one frame and associated information (`StorablePicture * frame`). After, the frame has its luma and chroma components combined in an identical way as the current frame using the `frame2rgb` function.

- Frame slice size info: VIDC requires the size of each slice to be known, but this information is no longer available after the decoder has finished decoding any given slice. The decoder knows the size of each slice while decoding it, but not afterwards. Fortunately, the SC code keeps track of the size of each decoded slice. This information is kept in the `int mb_per_slice[10000]` array. It is no more than a list of numbers stating the size of all the decoded slices. Since only the slice sizes for the current frame are needed, the following code is used to extract the information from the (potentially very long) list of slice sizes.

```
(*slice).size = slices_in_frame;
for(i=0;i<(*slice).size;i++) {
    (*slice).mb_per_slice[i] = mb_per_slice[slice_index+1-(*slice).size+i];
```

```
}
(*slice).last_mb[0] = (*slice).mb_per_slice[0]-1;
for(i=1;i<(*slice).size;i++) {
    (*slice).last_mb[i] = (*slice).last_mb[i-1]+(*slice).mb_per_slice[i];
}
```

The `SliceData *slice` struct contains how many slices are in the current frame and two arrays holding both the size and the last MB of each slice. `slice_index` contains the last read slice, which is also the index that can be used in the `mb_per_slice` array. This data structure provides all the information concerning slice sizes that the VIDC algorithm needs.

- SC detected errors: the errors detected by SC are stored in an array of `struct errMB`, which contains an error index, frame number, slice number and MB number of the error. Such list contains all the errors detected so far, and although it is practical to output the errors at the end, it would require looking it up every time the VIDC algorithm needs to know if a specific MB was marked as erroneous by SC. In order to avoid this, a simple 9x11 matrix storing `1` or `0` depending if a MB is marked as erroneous by SC or not has been implemented with the following code:

```
unsigned short det_pos[HEIGHT/16][WIDTH/16];

for(i=(last_scanned_error+1);(i<dec_err_ins)&&(frame_idx!=0);i++) {
    mb_row = (int)((double)dec_err_in_MB[i].mb_nr/((double)img->width/16));
    mb_col = (dec_err_in_MB[i].mb_nr) - mb_row * img->width/16;
    if(dec_err_in_MB[i].frame_nr==frame_idx) {
        det_pos[mb_row][mb_col] = 1;
        last_scanned_error = i;
    }
}
```

Then, in order to check if the MB in row 0, column 7 is marked by SC, VIDC only needs to access `det_pos[0][7]` and check whether it stores a `1` (marked by SC) or `0` (not marked).

- Frame type info: getting what frame type the current frame is is quite easy, as this information is stored in the `StorablePicture * dec_picture` structure. and its numerical values already translated with `#defines` preprocessor instructions. Only the following code is needed:

```
int slice_type = dec_picture->slice_type;

if(slice_type == I_SLICE) do_something;
else if(slice_type == P_SLICE) do_something_else;
```

5.5.2 Code integration

The VIDC implementation, including the additional code needed to interact with the JM+SC decoder and the MB level edge analysis, contains a total of 16 data structure types and 38

functions. These perform frame analysis, calculations and data output and weight 2500+ code lines. A complete reference of all the functions and a call graph can be obtained by running DoxyGen on the JM+SC+VIDC decoder.

All of the VIDC logic is called from the `exit_picture` function, located in the `image.c` file. Basically, the code is called just before the current picture is stored in the buffer and the read and decode process for the next frame begins.

The pseudocode for the modified `exit_picture` function is as follows:

```
exit_picture() {
[...]

// VIDC code
initialize VIDC variables;
get frame slice size data;
get SC detected errors in frame;
if(frame!=first frame) {
    if(slice_type==I_SLICE) {
        detecti();
        conceal detected errors and following MBs until the end of the slice;
    }
    else if(slice_type==P_SLICE) {
        detectp();
        conceal detected errors only;
    }
}
VIDC output;
// End of VIDC code

store_picture_in_dpb(dec_picture);
screen output;
}
```

As it can be seen, since the VIDC code is not spread all over the JM code, it is easy to identify (its logic also resides in a separate `detector.c` and `detector.h` files) and easy to isolate if necessary (turn it off). It is possible to activate/deactivate the VIDC and also the SC algorithms via the decoder configuration file (the latter was added as an improvement to the SC implementation). This is explained in Section 5.5.5.

5.5.3 Algorithm improvements

Several improvements were added in this implementation. These improvements mainly improved the detection performance of the modified decoder and are explained in the subsections below. These improvements are:

- MB level edge detection (VI+).
- CRC checksum check.
- Additional error modes.
- Usage of a grayscale frame obtained from averaging the luma and chroma components (not detailed here since it is explained in Section 5.5.1).

5.5.3.1 MB level edge detection (VI+)

As mentioned in Section 4.2.1, originally the VIDC algorithm performed edge search only in 8x8 blocks. Since this was not enough to detect some artifacts, 16x16 block size edge search was added.

This extra code, which is labeled as `VI+` in the source, affects only the `analyze_cand` function, only needing a minimal change in the part that performs the concealment.

The data structure where the error candidates are stored was thought for 8x8 blocks, so a new structure is used for VI+. The resulting `struct` is shown below. It should be noted that not all of the information contained there is used to determine if a candidate is to be regarded as an error. Nonetheless, it was needed to decide the actual rules that would be used.

Since the goal is not to optimize the decoder, the extra information was left in the code, as it may be some day useful.

```
typedef struct mb_error_data
{
    float is_error;
    float edges[NUM_OF_EDGES];
    float unordered_edges[NUM_OF_EDGES];
    short number_of_times_candidate;
    short marked_as_error;
    short data_inputted;
    float sum_of_all_edges;
    float average_power;
    short marked_by_VI;
} MBErrorData;
```

The algorithm works with a 11x9 matrix of `MBErrorData` that represent the 99 MBs in the frame. Each variable serves the following purpose:

- `is_error`: sum of the `is_error` variable of all the 8x8 blocks in the MB.
- `unordered_edges[NUM_OF_EDGES]`: stores an edge value (referred as e in previous sections) for each edge. Each position in the array represents a specific edge, as depicted in Figure 5.17. `NUM_OF_EDGES` is equal to 12, as there are 12 edges.

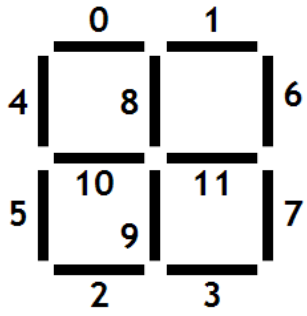


Figure 5.17: Edge numbering in VI+

- `edges[NUM_OF_EDGES]`: in order to be able to have a sum of the N highest edges, the edges list must first be ordered. This struct element contains the same information as `unordered_edges`, but ordered (although when ordered, one cannot know to which edge each value belonged).
- `number_of_times_candidate`: it counts how many of the 8x8 blocks that form the MB were marked as candidates.
- `marked_as_error`: tells if the MB has been marked as erroneous by VI+.
- `data_inputted`:marks if the data from the MB has been inputted.
- `sum_of_all_edges`: sum of all the edges (e var from the 8x8 blocks) from the MB.
- `average_power`: the average power of the 8x8 blocks from the MB.
- `marked_by_VI`: MBs that have blocks that were already found erroneous by the 8x8 block level edge analysis are not analyzed using VI+. This variable marks whether the previous analysis found an error in this MB or not.

Since the structure of the rest of the program was to remain unchanged, the way in which the errors are outputted could also not change. The `analyze_cand` function outputs a list of 8x8 erroneous blocks. In order for this output not to be modified, erroneous MBs marked by VI+ are also outputted in this way, just that the sub-MB position is marked as 9,9. For example, if in `analyze_cand`'s output list a MB like `0,1,9,9` appears, it means that MB `0,1` (first row second column) was found erroneous by VI+.

As it will be explained in Chapter 6, because of this new addition, the threshold for the 8x8 block search was lowered, as the search was now divided in two steps.

5.5.3.2 CRC checksum check

VIDC can detect as erroneous MBs that are actually correct (false positives). This can happen more in scenes with a high degree of movement. As VIDC works on top of SC, which is able to get the information of erroneous UDP packets, it is also possible to get from lower layers if the UDP CRC checksum was correct or not.

By only performing the concealment in slices in which the CRC checksum failed, false positives can be strongly reduced (Chapter 6). The implementation works by logging if the CRC of the packet (slice) was erroneous or not. For any particular slice, only when there is a CRC checksum failure will there be MB concealment.

5.5.3.3 Additional error modes

Apart from the aforementioned error modes (`error_mode 0` and `error_mode 1`), in order to test the algorithm, one additional error mode was inserted.

Previously, it was only possible to randomly insert errors in each slice or in a deterministic way (exact bit positions).

In `error_mode 3` the decoder inserts a random error in each slice every N frames, with the possibility to insert an error offset M.

In short, if N is 10 and M is 0, then errors will be inserted in frames 10, 20, 30,... (frame 0 is always error free). If N is 10 and M 1, then the errors will be inserted in frames 1, 11, 21, 31,... It should be noted that always $M < N$, if not the errors will be erroneously inserted or unexpected behaviors may arise.

5.5.3.4 SC concealment

When SC detects an error in the bitstream that leads to the undecodability of one or several MBs and SC concealment is off, it substitutes those with a green MB. This error is referred in the source code as a `too_few` error, as there is not enough information to try to decode the MB. In this way, `too_few` errors were visible when SD decoding. As no information was available to fill the gap of MBs about which there is not enough information, that gap is filled with the default luma and chroma values, which correspond to green.

This is only performed in the case in which the aforementioned MBs are absolutely undecodable. Since VIDC gets the frame once it is decoded, it is useless if it has to detect green artificially inserted MBs, as it was SC that inserted them.

This is true for P frames, as only discrete MBs are to be concealed anyway. Thus, the SC algorithm was modified so in these specific cases, it would perform a copy-paste concealment of the given MB instead of inserting a green MB.

For I frames, since those green MBs make an erroneous sequence more visible (and thus more detectable), SC will still insert green MBs.

Then, the SC algorithm (when VIDC is active) will behave in the following way for each decoded MB:

```
log_error;
if(too_few) {
   if(I_FRAME) {
       insert_green_MB;
   } else {
       copy_paste_conceal_MB;
   }
}
```

As the SC information is only used in the I frame voting system, this change does not affect how the VIDC algorithm works, since for I frames no change was made. As for P frames, now instead of green MBs, VIDC receives concealed MBs.

5.5.4 Error concealment

As mentioned several times, VIDC uses MBLC for error concealment in I frames, while it only conceals detected errors in P frames. This means that in I frames we conceal up the end of the slice, while in P frames only the detected artifact position.

Given that the previous frame is already loaded in memory in the `StorablePicture * prev_frame` structure, concealing a given MB is just a matter of knowing which pixel positions should be copied from the previous frame to the current one.

The following function conceals a given MB using a copy-paste method. `i` and `j` indicate the position of the MB (MB row and MB column in the 9x11 MB matrix that represents the frame).

```
int overwrite_mb(
    StorablePicture *frame,
    StorablePicture *frame_prev,
    int i, int j);
```

This function overwrites MB(i,j) in `frame` with MB(i,j) in `prev_frame`. So, the concealment code for I frames is:

```
if(visconc_on == 1 && is_mb_in_damaged_slice(error_slice_number,slice_err) == 1) {
    for(j=error_mb_num;j<=last_mb_in_slice;j++) {
        overwrite_mb(dec_picture,prev_frame,
           mb_num_to_mb_pos_i(j),mb_num_to_mb_pos_j(j));
    }
}
```

The mb_num_to_mb_pos_i and mb_num_to_mb_pos_j functions serve to translate a MB number to its coordinates (eg. MB 0 is MB(0,0), while MB 11 is MB(1,0)). For P frames the concealment is simplified to:

```
if(visconc_on==1  && is_mb_in_damaged_slice(error_slice_number,slice_err) == 1) {
    overwrite_mb(dec_picture, prev_frame,error_i,error_j);
}
```

As it can be seen, the concealment is only done in slices in which the CRC checksum failed, thus greatly reducing the probability of having a false positive.

5.5.5 Configuration file

The options that can be configured in the VIDC-embedded version of the JM decoder are more extensive that in the JM+SC version. A sample configuration file is shown in Figure.

Compared to the configuration file described in Section 5.3, the JM+SC+VIDC version offers the additional parameters:

- **Visual impairments MB concealment:** activates or deactivates the VIDC MB concealment. The detection is always done, this only controls the concealment.
- **get_error_i threshold:** allows to change the difference threshold used in the get_error_i function.
- **get_error_p threshold:** allows to change the difference threshold used in the get_error_p function.
- **Y, U and V component weight:** as mentioned in Section 4.2.1, the YUV frame is converted to a form similar to grayscale by means of the following formula: $frame_{GS} = 0.6 \cdot Luma + 0.2 \cdot C_b + 0.2C_r$. These 3 parameters allow for the weight of the luma and chroma components to be changed.
- **Error insertion interval:** this parameter and its following control the error insertion interval for error mode 3 (Section 5.5.3.3). In other error modes, these values are not used.
- **Edge thresholds:** the following four parameters configure the edge thresholds in the test_edge function.
- **conc:** activates or deactivates the SC algorithm. This parameter works together with the one that controls the VIDC concealment. The different combinations of these two parameters yield the following results:

```
foreman_700b_10i_1l_50f.264     ........H.26L coded bitstream
for_dec_700b_10i_1l_50f.yuv     ........Output file, YUV/RGB
foreman_QCIF_420.yuv  ........Ref sequence (for SNR)
1                     ........Write 4:2:0 chroma components for monochrome streams
1                     ........NAL mode (0=Annex B, 1: RTP packets)
0                     ........SNR computation offset
2                     ........Poc Scale (1 or 2)
500000                ........Rate_Decoder
104000                ........B_decoder
73000                 ........F_decoder
leakybucketparam.cfg  ........LeakyBucket Params
0                     ........Err Concealment(0:Off,1:Frame Copy,2:Motion Copy)
2                     ........Reference POC gap (2: IPP (Default), 4: IbP / IpP)
2                     ........POC gap
0                     ........Error Mode
error.txt             ........Error File
0                        ........Visual impairments MB concealment: 1=ON 0=OFF
20                    ........get_error_i threshold
20                    ........ger_error_p_threshold
0.60                  ........Y component weight in the yuv2grayscale conversion
0.20                  ........U component weight in the yuv2grayscale conversion
0.20                  ........V component weight in the yuv2grayscale conversion
20                    ........Error every N frames
0                     ........Error offset M in frames: M<N always!!!
150                    ........Edge threshold
400                    ........Edge threshold2
1.75                   ........Edge diff threshold 1
4.00                   ........Edge diff threshold 2
0                   ........conc

This is a file containing input parameters to the JVT H.264/AVC decoder.
The text line following each parameter is discarded by the decoder.
```

Figure 5.18: JM+SC+VIDC configuration file

5.5.6 Output files

In addition to the output files stated in Section 5.1.2, other useful output files have been added. As in the same case as in the JM+SC decoder, these files are in CSV format:

- `visual_impairments.txt`: lists the errors detected by the VIDC algorithm and the MBs that are concealed due to each error. In case the VIDC concealment is deactivated, those values only show the MBs that would have been concealed.
- `candidate_edges.txt`: in order to test the VI+ addition to the VIDC algorithm, this other output file was added. It describes the edges of all of the MBs that have been detected as erroneous by VI+.

5.5.6.1 VIDC detected errors

The `visual_impairments.txt` contains a set rows, each describing one error detected by VIDC. Each column contains information related to each error. The fields are the following:

- `#Error`: error number. A counter that keeps track of how many errors there are.

SC	VIDC	Description
0	0	SC deactivated, VIDC performs detection without concealment, but information from SC is not available.
0	1	VIDC active. concealment is performed, but information from SC is not available.
1	0	SC active. Concealment applied by SC (MBLC), VIDC only detects. Actual results are like in the JM+SC decoder.
1	1	VIDC active and performing concealment. SC information is available to VIDC.

Table 5.1: SC and VIDC configuration

- `#Frame`: frame number in which a given error occurred.
- `#Slice`: the slice number of the error occurrence.
- `#Total Slices`: total number of slices of that frame.
- `Type`: if the frame was an I or P frame.
- `Init Threshold`: the value of the `diff_frame` variable that was used for that frame (average difference of the previous frame, as stated in Section 4.22).
- `MB i` and `MB j`: row and column in which the erroneous (P frame) or beginning of the erroneous MB sequence (I frame) was detected.
- `MB i1` and `MB j1`: tells the position in which the erroneous 8x8 block is within the MB. If those values are 9 9, it means that the error was detected by VI+.
- `MB num`: `MB i` and `MB j` described the erroneous MBs in terms of coordinates, `MB num` numbers the MBs from 0 to 98 according to their position ($11 \times MB\_i + MB\_j$).
- `Last MB concealed`: for I frames, displays the last MB that was concealed due to this error (end of the slice). For P frames, it is the same number as `MB num` (only one MB per error is concealed).
- `MB power`: power of the erroneous 8x8 block. In VI+ detected artifacts, this value is not used.
- `Seq length`: for errors detected in I frames, it displays the sequence length at the moment the error was detected.
- `Vote`: for I frames, it displays the value of the vote that particular sequence had at the moment it was decided it was erroneous. The intended output was the vote of the whole sequence (measure of the magnitude of the error), but it was only possible to output the value at decision time (after that, the voting system algorithm stops, so no further information is available).

Unless stated otherwise, each outputted value is zero-indexed, which means that the first element is the number 0 (first frame is frame 0, first MB is MB 0, etc). Although the output of the `detecti` and `detectp` functions is one-indexed (so the output of the Matlab and C implementation would be identical), the final output, which is dumped into the

visual_impairments.txt file is zero-indexed, as it was decided that it would be confusing if it were one-indexed. Figure 5.19 shows an example visual_impairments.txt (the header line has been suppressed, as it only contains a description of the fields).

```
0, 26, 0, 2,P, 13.52, 3, 3,0,0,36,36, 36.38,  0,  0.00
1, 26, 0, 2,P, 13.52, 5, 4,1,1,59,59, 32.25,  0,  0.00
2, 26, 0, 2,P, 13.52, 6, 2,1,0,68,68, 22.06,  0,  0.00
3, 26, 0, 2,P, 13.52, 6, 2,1,1,68,68, 38.20,  0,  0.00
4, 26, 0, 2,P, 13.52, 3, 7,9,9,40,40,  0.00,  0,  0.00
```

Figure 5.19: VIDC sample output file

5.5.6.2 VI+

As this output CSV file was at the beginning used to find the adequate thresholds for the VI+, it contains almost all of the information relating edges for each macroblock detected by VI+. Its fields are as follow:

- #Frame: frame number.
- MB_i and MB_j: position of the MB.
- num_cand: number of 8x8 blocks from this MB that were error candidates.
- power_00, power_01, power_10 and power_11: the power of the upper leftmost, upper rightmost, lower leftmost and lower rightmost 8x8 blocks that form the MB.
- The value of the e parameter for all of the 12 edges, ordered by position.
- sum_of_all_edges: sum of the e value for all of the edges from the MB.
- The value of the parameter for all of the 12 edges, ordered by value (from lowest to highest).
- avg_pow: average power of the MB.
- is_err: sum of the is_err parameter of all the 8x8 blocks in the MB.

Figure 5.20 shows a sample candidate_edges.txt file. The edges with a value of 0 represent edges corresponding to the frame border (no neighboring MB). The '/' characters represent a continuation of the line. As in Figure 5.19, the first line, containing a description of the fields, has been suppressed.

```
025,002,008,001,17.06,14.06,19.61,24.70, /
    39, 93, 6,  6, 23,  7,302, 463, 95,106,129,  9, 1113, /
    6,  6, 7,  9, 23, 39, 93,  95,106,129,300,300,18.86,1.1
025,004,008,004,26.09,20.73,21.72,21.69, /
    139,205,  5,  8,545,382, 48,  3, 89, 71, 45, 40,1253, /
    3,  5,  8, 40, 45, 48, 71, 89,139,205,300,300,22.56,1.1
030,004,007,004,32.72,37.27,32.77,37.75, /
    7, 72,145,  6,  6,  8,302,296,  0,  0,  0,  0,840, 0, /
    0,  0,  0,  6,  6,  7,  8, 72,145,296,300,35.13,0.0
```

Figure 5.20: VI+ sample output file

Chapter 6

Algorithm testing and performance

This section of the project was thought as a performance test of the finished decoder. It also considered performing some minor modifications like adjusting thresholds in case those were needed, but that was not the main aim. It finally turned more into a repeating cycle of testing and improvement of the algorithm.

The following sections describe the testing process that led to many modifications in the implementation and the final results of the JM+SC+VIDC implementation.

6.1 Initial performance tests and modifications

At first, only adjusting the I and P frame difference threshold was deemed necessary. For this, several test decodings in which a pseudo-random error sequence was inserted. The decoder generated an error sequence that introduced an error in each slice (`error_mode=1`). The error pattern was always the same, as the random number generator was always initialized with the same seed.

With these tests, an optimum I and P difference threshold was obtained. These two values regulate how many candidates reach the edge-analysis stage. The lower the number the less strict the difference criteria is.

Although an optimum I and P thresholds were found, the results were not satisfactory, as chroma artifacts and green MBs were a lot of times not detected. For this reason a combination of the chroma and luma frames was implemented. After the implementation of this section into the decoder, the threshold optimization tests were repeated.

These were the results: Figure 6.1 shows the Y-PSNR results of the decoded sequence, while Figures 6.2 and 6.3 represent the U-PSNR and V-PSNR respectively.

Each point in the surface represents a decoding which was done using a specific I frame and P frame threshold. This decoding has a PSNR associated to it, as the decoder compares the decoded video to a reference one. This reference video is the original QCIF foreman sequence, whose encoded version is used at the decoder side. In the title of the figures the PSNR of the unconcealed sequence is shown, so it can be compared with the optimum value found in the surface (marked as *Maximum*).

The reason for including U-PSNR and V-PSNR in the tests was to judge if the optimum values for the luma and the chroma components were similar. While for the chroma compo-

nents the optimum value was the same, the results for the luma component showed that a more strict threshold was necessary.

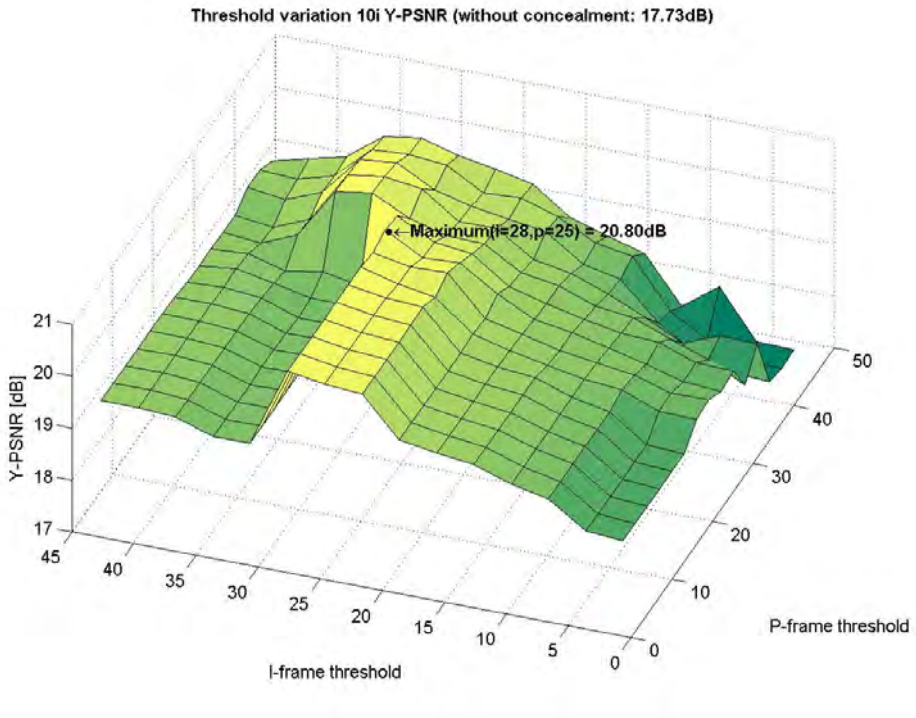


Figure 6.1: I and P threshold calibration (Y-PSNR)

These results were once more unsatisfactory, as some artifacts remained undetected (see Figure 4.21). For this, MB level edge detection was added. In order to find the right conditions to detect those previously undetected artifacts, a different approach was used.

In order to know the characteristics of the candidate MBs (the ones passed by the difference analysis to the edge detection) the `candidate_edges.txt` output was created. Then, the conditions needed for sorting out previously undetected MBs were manually found.

Since the problem was that the 8x8 block detection could not detect some artifacts, it was not necessary to change the edge detection thresholds (although that option was also shortly tested). Lowering those thresholds would provoked more false positives than improved the detection.

A parameter that needed to be slightly changed was the difference frame threshold. If this threshold is too restrictive, not enough candidates reach the edge detection phase. So, in order to avoid this the I and P thresholds were lowered to 20.

After finding the conditions that made it possible to filter the previously undetected artifacts, the decoder was tested against 20 complete foreman sequences (400 frames) each with an error pattern with $BER = 10^{-5}$.

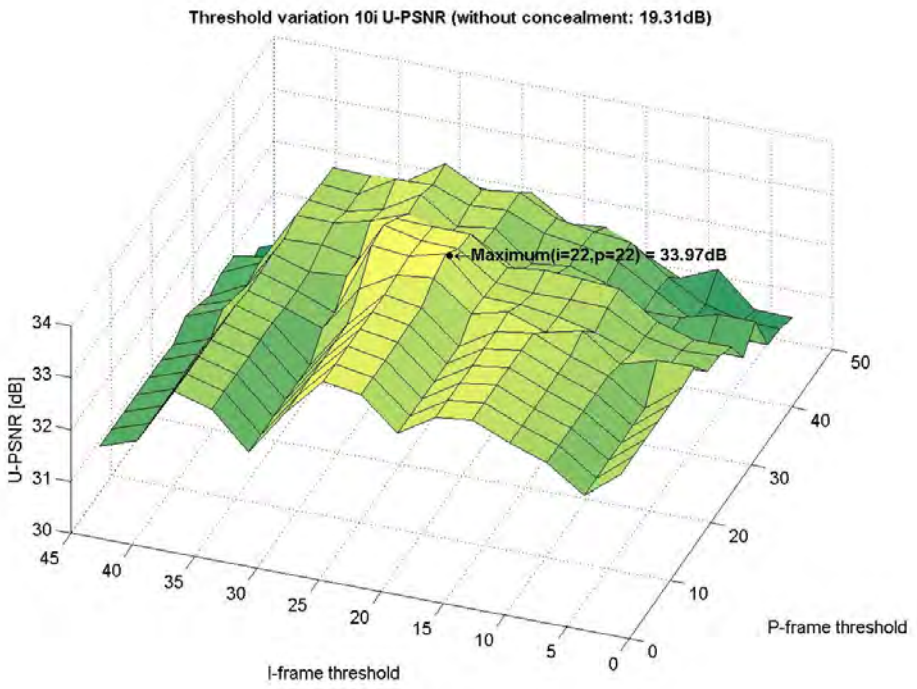


Figure 6.2: I and P threshold calibration (U-PSNR)

6.2 Y-PSNR results

The 20 error patterns were randomly generated with a $BER = 10^{-5}$ (these error patterns can be found in Appendix A). The source video was the complete foreman sequence and it was encoded with H.264/AVC baseline profile with $QP = 28$, $GOP = 10$ and a buffer size of 1 frame.

Figure 6.4 shows that these tests proved Y-PSNR improvements for all of the sequences. Although the mean gain is only 1 dB respective to the original straight decoded sequence, the gain taking into account only the frames where concealment was performed is around 6 dB, which is significant.

Although the algorithm effectively improves the objective quality (Y-PSNR) of all the erroneous sequences, false positives worsened the results in some sections of the sequences.

Figure 6.5 depicts the average gain for each of the 400 frames of the foreman sequence. As it can be seen, the decoder seems to detect and conceal artifacts that are in fact correct, thus worsening the quality of the video (negative Y-PSNR gain).

Since only 20 sequences were used to generate this average graph, it is quite spiky. Nonetheless, it serves to give an idea of the performance of the algorithm.

This occurs specially around frames 240 and 320. Around frame 240 the edge of foreman's helmet performs a vertical movement that makes it cross the MB boundaries almost parallel to them, thus creating a very strong edge. On the other hand, around frame 320 the camera performs a fast movement, which leads to a very strong frame difference.

Figure 6.6 shows the mentioned false positive. The grid shows the MB boundaries, while

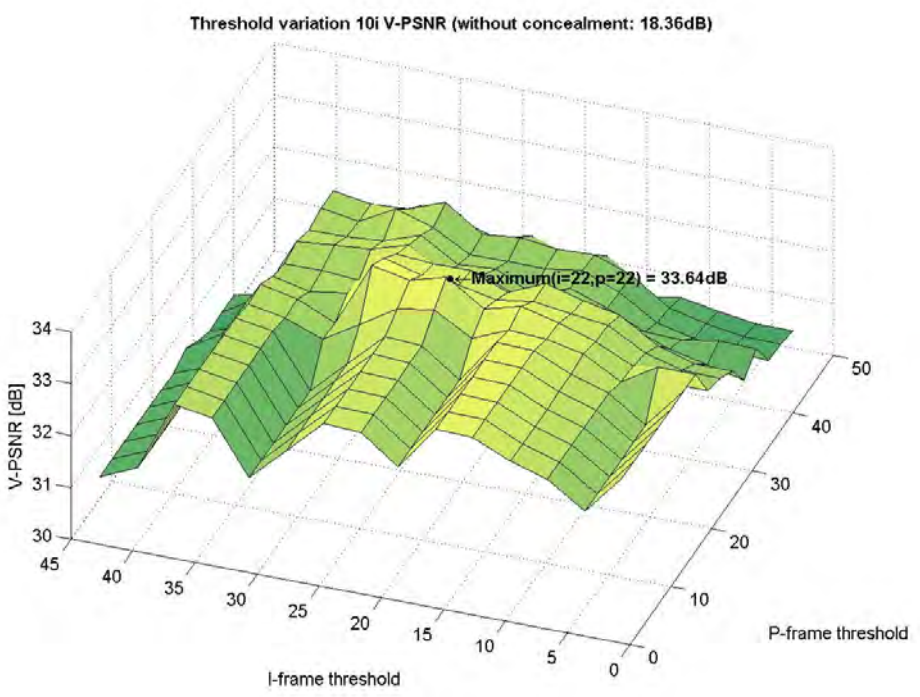


Figure 6.3: I and P threshold calibration (V-PSNR)

the blue mark shows the end of the slice. As it can be seen, the edge of the helmet crosses parallel to the MB boundary, thus triggering a false positive.

In order to further improve image quality, those false positives had to be eliminated. In order to be able to better detect when a strong edge was either product of causality or an artifact and to better detect artifacts in fast-moving scenes, additional processing should have been added.

6.3 Final Y-PSNR results

As a simple solution for the false positives, CRC checksum information was added to the algorithm. This method works by adding no additional complexity or processing and effectively reducing the false positives.

The improvement works by using the information from the UDP checksum of the received packets. Since whether the packet has a correct or incorrect UDP checksum is a knowledge that could be available to the decoder, it can be used to further refine the search for artifacts.

After the modifications, the algorithm performs artifact concealment only if in the considered slice the UDP CRC checksum failed. As is can be seen in Figures 6.8 and 6.7, the robustness against false positives (negative Y-PSNR gain) and Y-PSNR gain has been improved.

Although numerically the average gain per sequence (Figure 6.7) has not been improved greatly, Figure 6.8 shows how false positives have been considerably reduced. The negative Y-PSNR gain peaks have gone from reaching -4.5 dB to not going further than -0.5 dB.

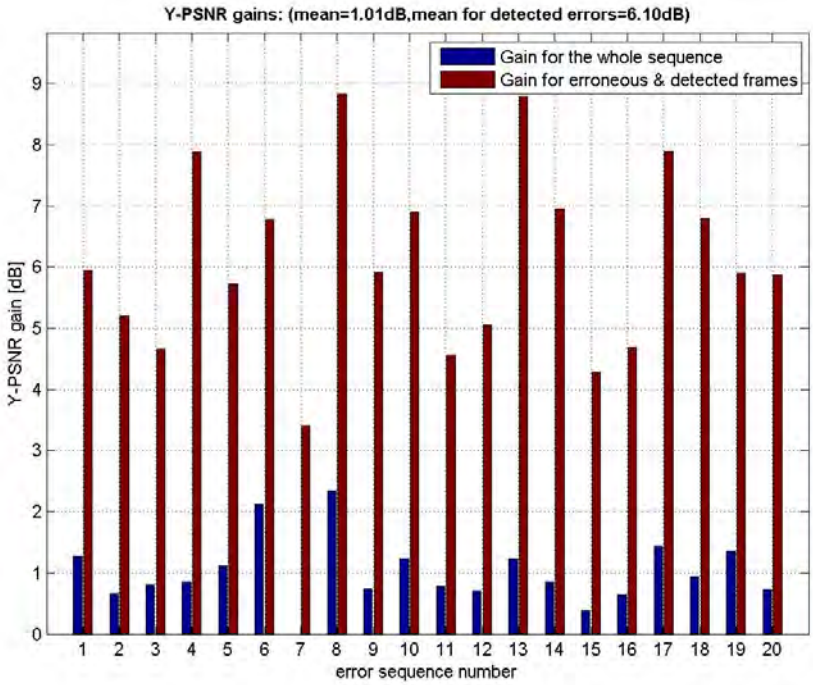


Figure 6.4: Y-PSNR gain for the 20 test sequences

Now, in order to have a concealed false positive, two conditions will have to be met:

- UDP CRC checksum of the given slice has failed.
- VIDC falsely detects an artifact.

The improvement in terms of Y-PSNR brought by the implementation of CRC checksum checking in the decoder can be observed in the following figures depicting the gain of test sequence number 11. This sequence was chosen for its representativity of a Y-PSNR improvement due to false positives reduction.

Figure 6.9 shows the Y-PSNR gain for each decoded frame of the foreman test sequence 11 without CRC checking. Figure 6.10 depicts the gain for the same sequence when using CRC checking. It can be observed how CRC checking has eliminated negative gain spikes (false positives).

The red dots indicate where the decoder introduced an error. So red dots where there is no gain either represent errors that did not produce any visible artifact or artifacts that were undetected.

This case depicts the best-case in which CRC validation eliminated all of the false positives. Obviously, this is not the case for al the sequences. The plots corresponding to the 20 test sequences can be found in Appendix B.

As mentioned in the introduction of Appendix B, sometimes the concealment can degrade the Y-PSNR. However, this does not mean that the picture necessarily looks worse to the human eye. Additional performance tests were performed for this reason, this time using a subjective metric. Although the JM+SC+VIDC is shown to work well in terms of Y-PSNR

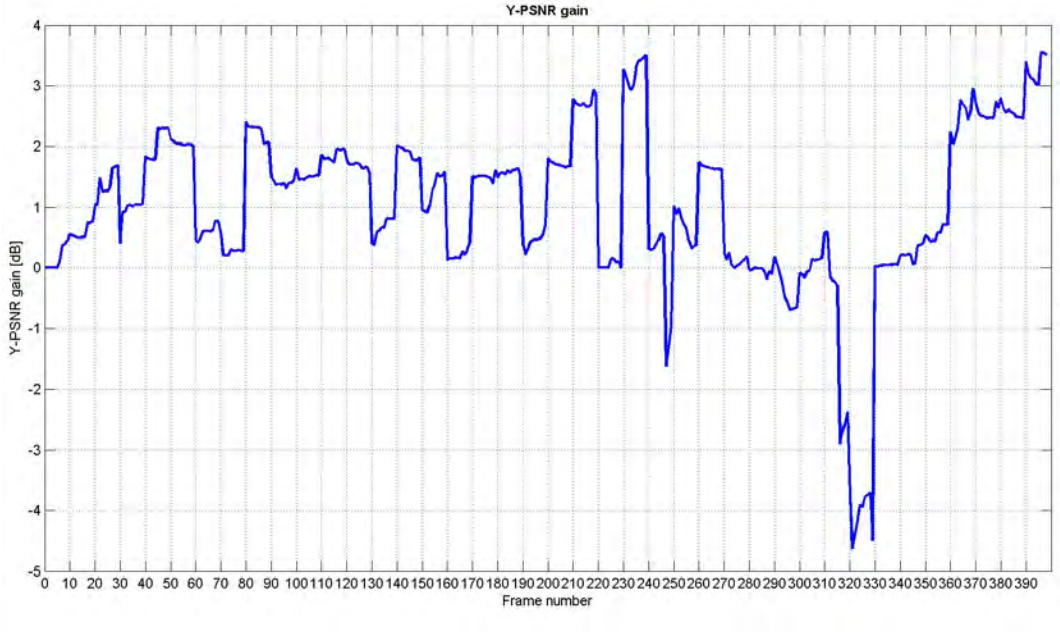
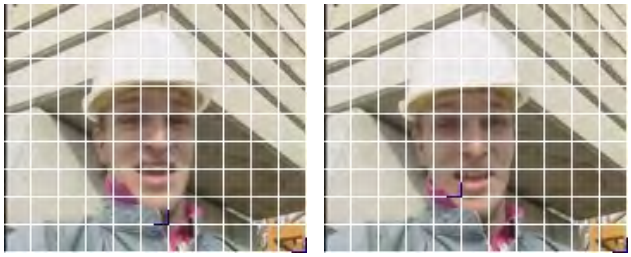


Figure 6.5: Average Y-PSNR gain



(a) Frame 241 (b) Frame 242

Figure 6.6: False positive in frame 242

(objective metric), tests using a subjective metric, the Mean Opinion Score (MOS) have been done. These tests are described in chapter 7.

6.4 Performance results

The results concerning the implementation of the VIDC algorithm into the JM+SC H.264/AVC decoder are, after all the modifications to the implementation, as follow:

- Average Y-PSNR gain: 1.21 dB.
- Average Y-PSNR gain in frames where concealment was performed: 6.11 dB.
- I frame Y-PSNR average gain (concealed frames): 3 dB.
- P frame Y-PSNR average gain (concealed frames): 0.3 dB.

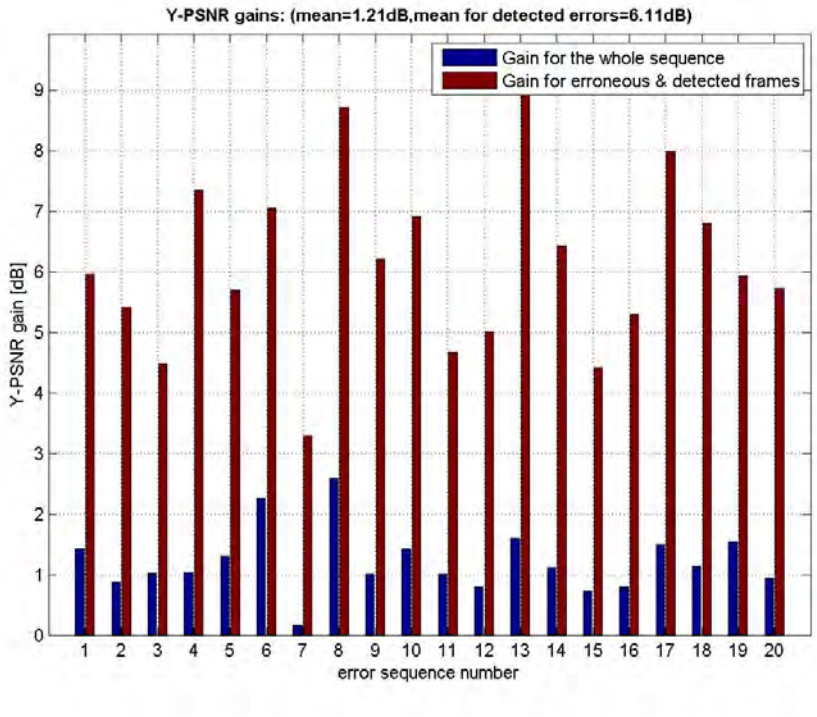


Figure 6.7: Final Y-PSNR gain for the 20 test sequences

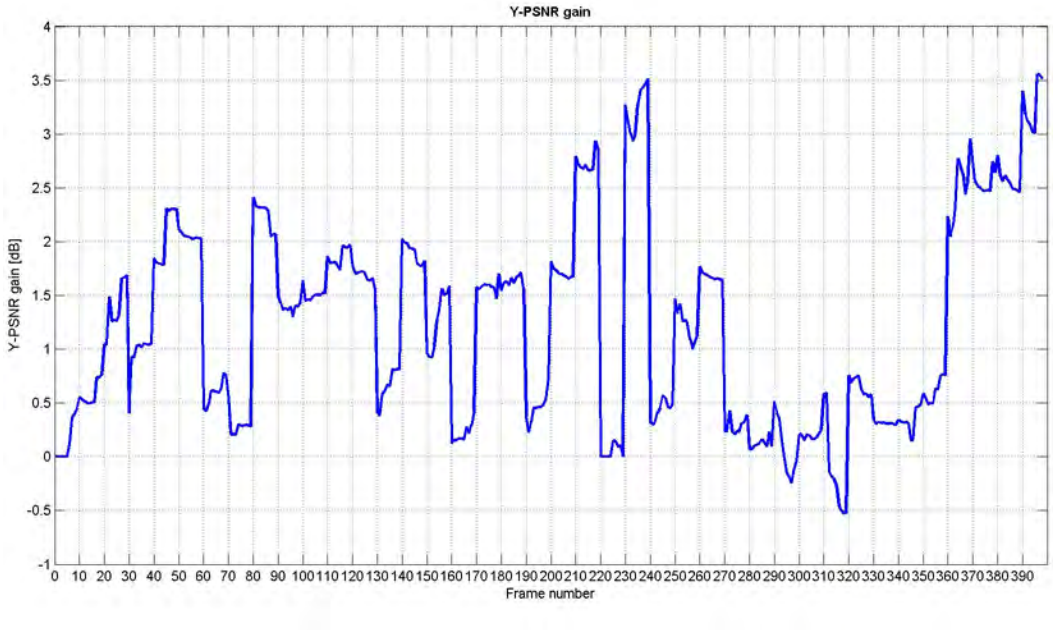


Figure 6.8: Final average Y-PSNR gain

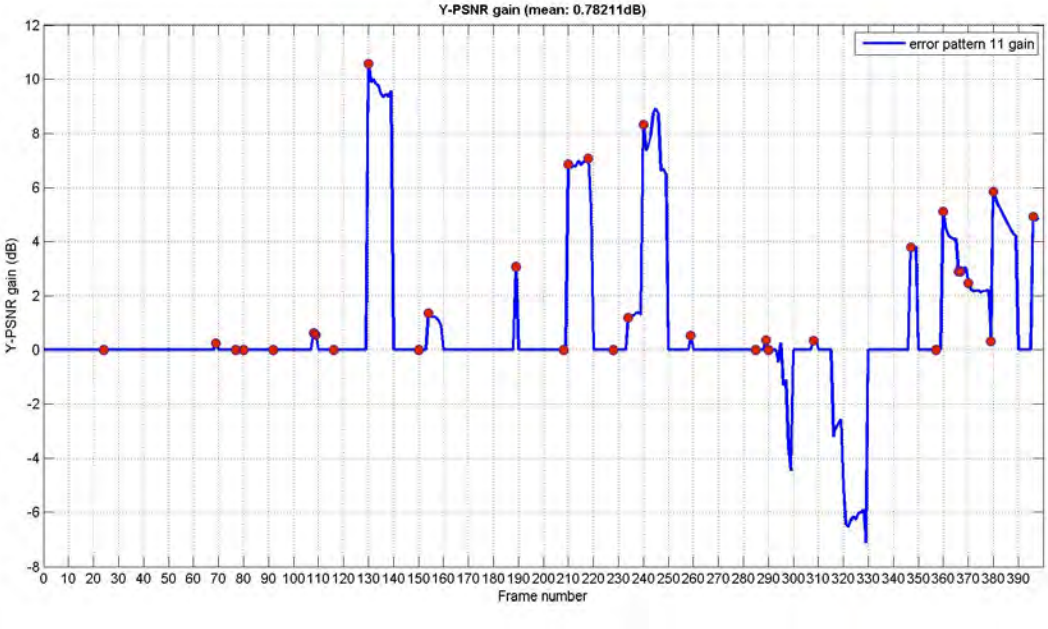


Figure 6.9: Test sequence 11 Y-PSNR gain (without CRC checking)

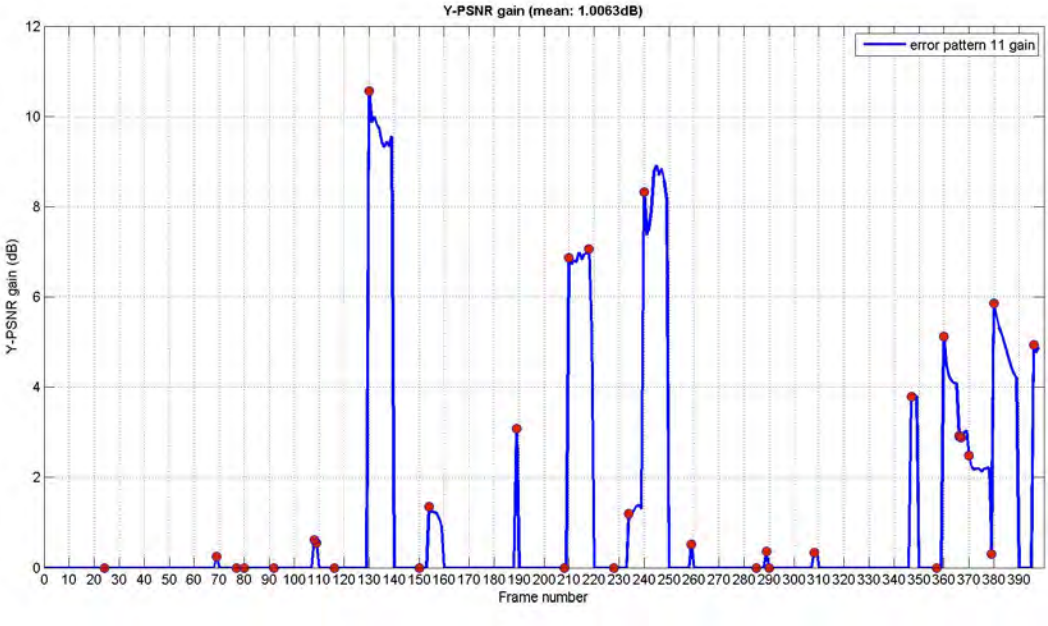


Figure 6.10: Test sequence 11 Y-PSNR gain (with CRC checking)

Chapter 7

MOS tests

As seen in chapter 6, the implemented algorithm improves the Y-PSNR of the erroneous H.264 video sequences. But it has also been mentioned (Figure B.8) that Y-PSNR does not always give an idea of the quality that a human would perceive.

Thus, the algorithm was tested using a subjective metric. Another reason for this is that this information is needed for the second part of the project. As mentioned in chapter 1, this work also presents the design of a quality estimator. Since the algorithm is designed to output the subjective quality of the decoded video stream, first such tests must be performed.

The data from these tests will not only be used to test the algorithm but also to design the quality predictor (chapter 8).

7.1 Testing procedure

The chosen subjective measure of quality has been the Mean Opinion Score (MOS). The MOS is generated by averaging the results of a set of subjective tests. In each test, a test subject is asked to rate the quality of a number of sequences. This process is repeated and the final mark for each sequence is the average give by the test subjects.

Each test subject will rate each sequence from 1 to 5, as defined by the ITU-T P.800 recommendation [14]. The marks represent the following subjective quality (Table 7.1):

MOS	Quality	Impairment
5	Excellent	Imperceptible
4	Good	Perceptible but not annoying
3	Fair	Slightly annoying
2	Poor	Annoying
1	Bad	Very annoying

Table 7.1: Mean Opinion Score (MOS)

The MOS is the arithmetic mean of all the individual scores, and can thus range from 1 (worst) to 5 (best).

7.1.1 Sequences used

The test each subject had to evaluate consisted of 46 individual sequences. The sequences consisted of a series of randomly ordered video sequences. The test contains a mixture of different error patterns in a fast moving sequence and in a slow moving sequence. There are a total of 22 different error patterns. The test contains each sequence straight decoded and concealed (44) and 2 control sequences that are error-free.

The individual sequences consist of a fragment of the foreman video sequence encoded with the following parameters:

- Baseline profile.
- $QP = 28$.
- frame buffer size of 1.
- $GOP = 25$.

In the test, each sequence contains a single bit error, be it either in an I or P frame. The whole sequence is comprised of 3 GOPs. The first is correct, the second contains the error and the last one is again correct. Thus, since the GOP size is 25 frames, each sequence is 75 frames long. The frame rate was set at 19, which is a little bit slower that normally. This was made so the viewer could appreciate better the quality of the sequences.

As mentioned, there are two types of sequences in the test: fast-moving sequences and slow-moving sequences. For each sequence type, the following fragments of the foreman video have been used:

- **Slow-moving:** frames 0-74 have been used (Figure 7.1a). In the part where the error is inserted (frames 25-50), only a slight movement in the protagonist's head is present.
- **Fast-moving:** frames 260-334 (Figure 7.1b). The central GOP of this sequence features a fast camera movement. It is because of the high degree of camera movement that it is in this area where most false positives are located.

(a) Slow moving sequence (b) Fast moving sequence

Figure 7.1: Foreman fragments used in the MOS tests

Table 7.2 describes all of the error patterns that have been used. It describes the following fields:

- **Seq ID:** identifies error pattern that the sequence uses. It goes from 1 to 22. Numbers 0 and 00 denote an error-free sequence (slow and fast moving respectively).

- **Sequence:** for a slow-moving sequence, it is 0-74, while for a fast moving 260-334. Those are the frame numbers of the original foreman sequence.
- **Frame:** the relative frame number in which the error is inserted. Since the errors are in the central GOP, it goes from 25 to 49.
- **Frame type:** either I or P frame, depending on the type of frame in which the error is inserted.
- **MB:** MB in which the error is inserted, as outputted in the `err_ins.txt` output of the decoder.
- **Slice:** slice within the single frame in which the error is inserted. The first slice is slice 0.
- **Error position:** the bit position in the NALU bitstream where the error is inserted. For each sequence there is only a single inserted error, except for sequence number 10, where two errors had to be inserted (the artifact was otherwise not visible).
- **Description:** for I frames, a description of how big is the artifact. Since in I frame the artifact propagates until the end of the slice, three different artifact sizes have been used in I frames: big (whole slice), medium (around two thirds) and small (one third of the slice). Also, for the slow-moving sequence, three different locations have been used for the erroneous slice: upper, middle (mid) and lower (low) part of the frame.

Seq ID	Sequence	Frame	Frame type	MB	Slice	Error position	Description
1	0-74	25	I	36	2	120178	mid-big
2	0-74	25	I	44	2	122652	mid-medium
3	0-74	25	I	53	2	124524	mid-small
4	0-74	26	P	17	0	138737	
5	0-74	34	P	40	0	166949	
6	0-74	43	P	51	0	201643	
7	0-74	25	I	88	5	120178	low-medium
8	0-74	26	P	44	0	139168	
9	0-74	27	P	40	0	141597	
10	0-74	27	P	42	0	141613 141614	
11	0-74	27	P	42	0	141613	
12	0-74	25	I	0	0	109106	upper-large
13	0-74	25	I	6	0	110874	upper-medium
14	0-74	25	I	12	0	113336	upper-small
15	0-74	25	I	83	4	130720	low-large
16	0-74	25	I	92	4	134445	low-small
17	260-334	25	I	25	1	192085	mid-big
18	260-334	25	I	34	1	194379	mid-medium
19	260-334	25	I	40	1	195292	mid-small
20	260-334	26	P	18	0	211031	
21	260-334	34	P	80	0	260561	
22	260-334	42	P	54	0	335601	

Table 7.2: Inserted errors for the MOS test sequences

All the error patterns produce visible artifacts, as the objective of the MOS test is to quantify the subjective quality degradation that artifacts produce. A snapshot of each frame with an inserted error can be found in Appendix C. Although temporal propagation cannot be appreciated in those snapshots, it can give an idea of the magnitude of the artifacts.

7.1.2 Test execution

At the time of the test, the subject was given a form in which he could tick a mark from 1 to 5 for each of the 46 sequences (16 slow-moving straight decoded, 16 slow-moving concealed, 6 fast-moving straight decoded, 6 fast-moving concealed and 2 error-free sequences).

Then, the test is comprised of a series of chained clips. First, an introduction repeating the instructions, two training sequences and then each test sequence in a pre-defined random order. The test sequences are intended to show the viewer what is considered an erroneous video and what a correct one.

Before each sequence, the viewer is informed of the sequence number, so he/she does not get lost filling out the form, and then given a short amount of time (5s) after the sequence to give the mark.

Figure 7.2 shows how each video sequence is put together for the test. The total length of each sequence is then 12 seconds (3s intro + 4s sequence + 5s voting). Including the instructions, the whole test has a duration of 10 minutes 23 seconds. Since the videos were in QCIF resolution intended to be viewed in a mobile device, a Pocket PC was used for that.

(a) Cut clip 1 (intro): 3s (b) Test sequence: 4s (c) Cut clip 2 (voting): 5s

Figure 7.2: MOS test sequence with cut clips

7.2 Test results

For the MOS measurements, the tests were performed 15 times. The results are shown in Table 7.3.

A more complete table, with the results of each individual test, can be seen in Appendix D.

From the results in Table 7.3, and analogous to the results in Section 6.4, the error detection and concealment provided the following gains (MOS):

- **I frames:** 2.48.
- **P frames:** 0.47.

Not surprisingly, the subjective quality improvement is much higher for errors in I frames than in P frames. This is due to errors in I frames producing much more visible artifacts. Also, regarding P frames, although all of the errors produced visible artifacts, the viewer may

Seq ID	MOS	Seq ID	MOS	Seq ID	MOS	Seq ID	MOS
1	4.67	13	4.27	25	3.53	37	4.80
2	4.67	14	1.60	26	4.87	38	4.87
3	1.07	15	1.60	27	4.87	39	4.73
4	1.13	16	2.00	28	4.20	40	4.80
5	2.27	17	2.20	29	3.60	41	2.53
6	3.07	18	2.87	30	3.87	42	2.73
7	2.60	19	1.07	31	4.80	43	3.60
8	3.60	20	1.53	32	4.13	44	2.07
9	2.07	21	1.73	33	4.00	45	4.00
10	4.07	22	2.20	34	4.00	46	3.53
11	3.73	23	3.07	35	4.33		
12	3.07	24	3.40	36	4.80		

Figure 7.3: MOS test results

or may not perceive them. The fact that an artifact is there does not necessarily mean that the viewer will notice to it.

This information, although useful for just performance testing, will be used to design a quality estimator (chapter 8). Matching the information outputted by the decoder for each sequence and the results of the MOS tests, it is possible to make an estimation of the quality of the decoded sequence.

Chapter 8

Quality estimator

By using the information from the SC and VIDC algorithms, it should be feasible to estimate the quality of the received video. One of the aims of this work was to enable the estimation of the perceptual quality of the sequences outputted by the decoder. It was not the aim of this work to integrate the quality estimator in the decoder, but rather to find a function that relates the artifact information to the perceptual quality estimation.

Figure 8.1 shows how the final JM decoder implementation with SC and VIDC interacts with with the quality estimator.

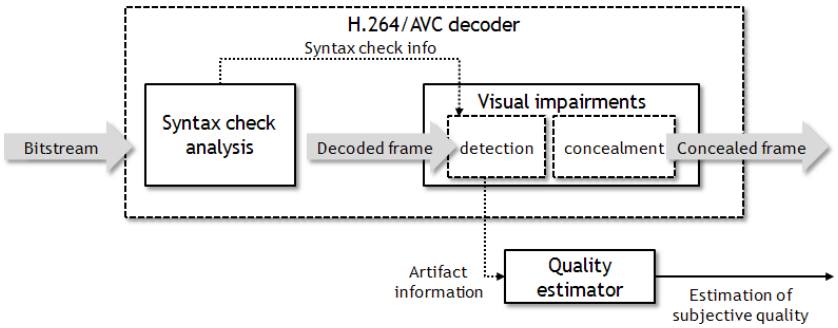


Figure 8.1: Quality estimator

The decoder outputs information about the decoded sequences, mainly speaking a list of artifacts and parameters describing those artifacts. Taking this information, the estimator is then capable of predicting the perceptual quality of the sequence. Since at this moment the estimator is just a design, it works completely isolated from the decoder. Once the decoding of the sequence is finished, the outputted data is fed into the estimator, which then outputs the predicted quality of the sequence.

Since the artifact information is obtained in the detection part of the VIDC algorithm, before any concealment is done, one could argue if the quality estimation is performed for a concealed or unconcealed sequence. As it will be seen, actually both things are accomplished. We estimate the quality (outputted as a MOS value) of both concealed and unconcealed sequences at the same time.

The MOS estimator was designed using the results from the MOS tests (chapter 7), so

its results are based foreman sequences encoded with $QP = 28$, $GOP = 25$ and 75 frames long. These sequences contain an error-free GOP followed by one which contains one error, ended by an error-free one. The sequences are the same as in the MOS tests, so there are a total of 22 sequences. 16 slow-moving and 6 fast-moving. The sequences and the artifacts in them are described in Section 7.1.1 and Appendix C

8.1 Estimation

In order to design the quality estimation, what basically needs to be done is a regression from the detected video artifacts and relate that to a MOS score. Figure 8.2 shows a block diagram of this design.

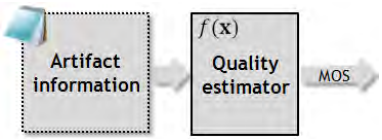


Figure 8.2: Initial quality estimator design

The data outputted by the decoder describing the detected artifacts by the algorithm (`visual_impairments.txt`) is analyzed and taken into account several metrics regarding the artifact, a MOS estimation is outputted.

However, creating a function that takes into account all of the parameters and correlates well with the data is not that simple. Whenever possible, it is advisable to break down the problem in simpler parts. Also, with the approach in Figure 8.2 we would only get the MOS estimation for a concealed sequence, and it would also be interesting to be able to estimate unconcealed (SD) sequences.

Figure 8.3 depicts the design that was finally used.

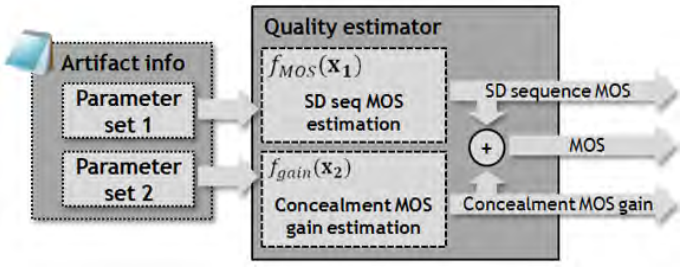


Figure 8.3: Quality estimator design

Instead of feeding the predictor a huge number of variables at the same time, there are now two smaller sets of variables which feed two separate regression equations.

So, the prediction is a two-step process that, formed by simpler parts that a as whole perform the same function as Figure 8.3 and even provide more information.

After splitting the process, the MOS estimation consists of the following steps (Figure 8.4):

1. Estimation of the MOS of the sequence in the case no concealment is introduced (f_{MOS}).

2. Estimation of the MOS gain the concealment will provide (f_{gain}).

f_{MOS} is obtained by performing a regression of the detected artifacts and MOS score of the SD sequences, while f_{gain} is obtained from the concealed sequences.

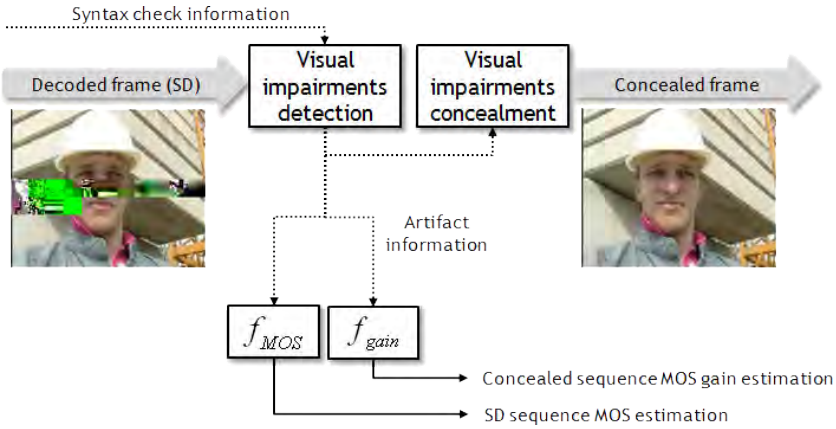


Figure 8.4: Realization of the quality estimator

This approach offers another advantages, besides simplifying the parts. By separating the estimator, it is possible to use it with SD sequences. If only f_{MOS} is used, this method can be potentially used on any H.264/AVC video sequence to estimate its perceptual quality. If, because of transmission errors concealment is also used, then f_{gain} can also be used.

It also helps isolating a source of inaccuracies, which is mentioned in Table 5.1. Due to the prior JM+SC implementation, it is not possible to perform only detection with both SC and VIDC. As the table explains, detection-only can only be done with VIDC (SC deactivated), so the SD sequences cannot use information from SC. Splitting the estimation process also helps in isolating this inaccuracy. For concealed sequences, which are used for MOS gain estimation, VIDC has SC information available, so it is used.

For both the MOS estimation and MOS gain estimation, the information used for the prediction is the one provided by the modified decoder in the `visual_impairments.txt`. See Section 5.5.6.1 for details on the outputted artifact information. The description of the datasets used in the analysis can be found in Appendix E.

8.2 MOS estimation

The MOS estimation function is obtained performing a regression of the data in the `visual_impairments.txt` and the results of the MOS tests.

The MOS tests contained SD and concealed versions of the same error pattern. In order to perform the MOS estimation (f_{MOS}, only the MOS tests from the SD are used. For the SD sequences, only artifact detection (not concealment) was enabled, so the artifact information is available.

The aim is to obtain a function that relates the parameters of the detected artifact and the MOS score of the sequence.

The relevant variables that have been considered and their expected influence in the MOS are:

- **Detected artifact size:** Measured in MBs. The bigger the artifact size, the lower the expected MOS. Big artifacts belong exclusively to I frames, as in P frames, commonly they do not propagate spatially.

- **Position of the error within the GOP (GOP position):** Distance to the last I frame. The closer the damaged frame is to the beginning of the GOP, the bigger temporal propagation is, hence a lower expected MOS.

- **Sequence length:** Useful only for I frames. The number of MBs the voting system needed to detect a given artifact sequence. The more visible the artifact, the bigger it should be (which at a later point proved to be false, see Section 8.2.1).

- **Difference frame:** Measures the average difference of the frame respective the last one. The errors are expected to propagate more in fast-moving sequences, hence it was expected that the MOS would be lower in the fast-moving sequences. How this parameter is calculated is explained in Section 4.2.1.1.

- **Frame type:** The frame type in which the error was detected (I or P). Artifacts in I frames are almost always much more visible that in P frames, where they tend to be only errors in motion vectors.

Figures 8.5, 8.6, 8.7 and 8.8 show 4 scatter graphs that depict the relationship between the various parameters and the MOS value.

While the relationship between the artifact size (Figure 8.5) and the difference frame (Figure 8.8) can be considered to have exponential and linear behavior respectively, with different gradients for I and P frames, the relationship with the other two parameters is not clear. It should also be noted that difference frame could also be assumed to behave exponentially.

The data points where the difference frame is 0 represent sequences where the algorithm didn't detect any artifact and have not been used for the regression analysis. The estimator will only work when there is data from the VIDC algorithm available. The same applies to the scatter plots of the MOS gain.

8.2.1 Dropped variables

- **Sequence length:** the variance of this variable is regarded too small to include it in the equation. As alternatives, averaging it with the *vote* value in the I frame detection algorithm and using the vote itself were tried. These alternatives made the variance decrease even more. Thus in the end the use of this variable has been dropped.

8.2.2 Further simplification

In order to further simplify the problem, two different functions have been defined (one for I frames and one for P frames). Since the sequence length has no meaning for P frames (it is only used in the I frame voting system) and for I frames the GOP position is always 0, the number of independent variables can be reduced to 3 instead of 4 (8.1).

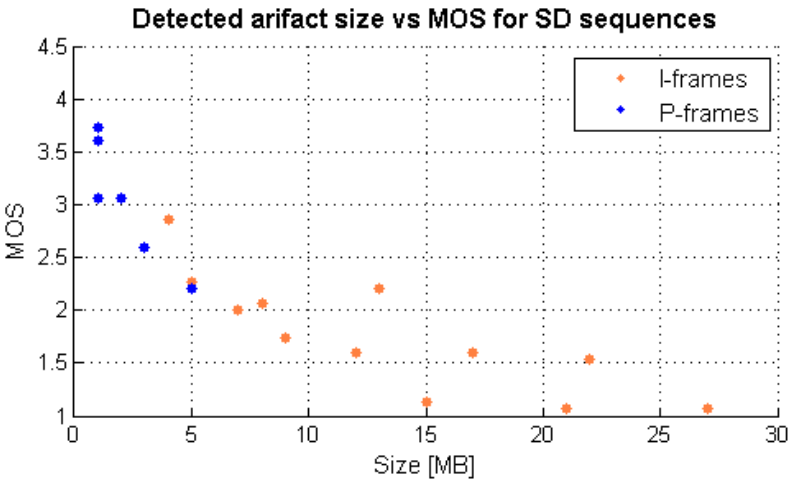


Figure 8.5: Detected artifact size vs. MOS

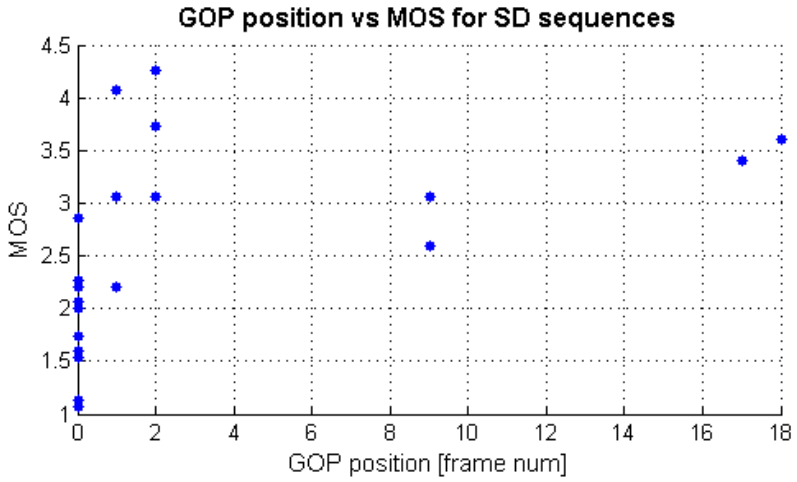


Figure 8.6: GOP position vs. MOS

$$
\begin{aligned}
f_{MOS}(n,g,d,t) &= \begin{cases} f_{MOS\text{-}I}(n,d) & \text{if } t = \text{I frame} \\ f_{MOS\text{-}P}(n,g,d) & \text{if } t = \text{P frame} \end{cases} \\
g &= \text{GOP position} \\
d &= \text{Difference frame} \\
t &= \text{Frame type (I frame or P frame)} \\
n &= \text{Artifact size}
\end{aligned} \tag{8.1}
$$

The DataFit software from Oakdale Engineering has been used to perform the curve fitting. Out of the available models, the exponential model proved to perform best (discarding

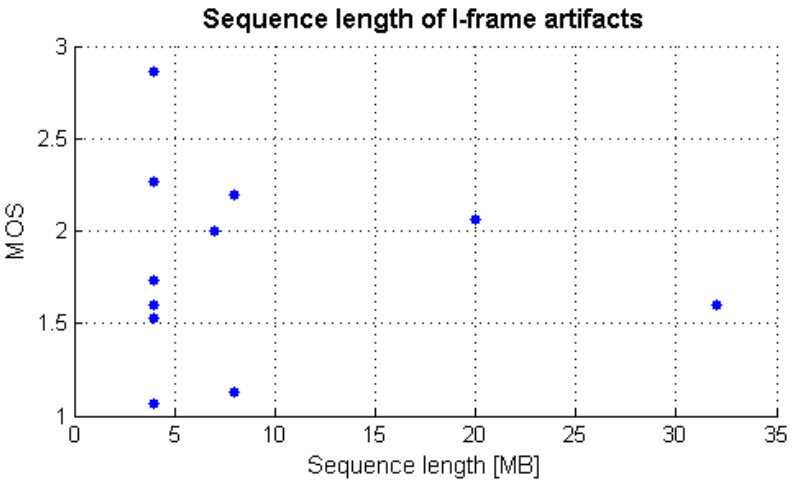


Figure 8.7: Sequence length vs. MOS

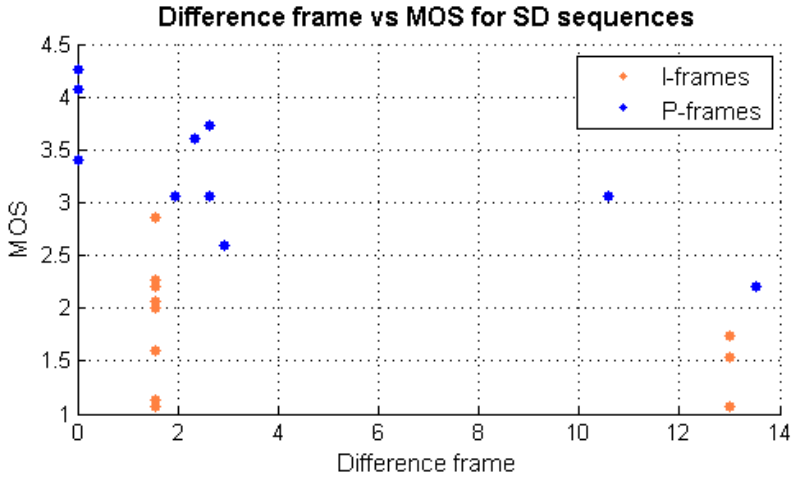


Figure 8.8: Difference frame vs. MOS

models that merely adjusted to the data, not being coherent) for both I frames and P frames, so the models in (8.2) and (8.3) have been used.

It should be noted that a model in which *difference frame* was linear was also tested, but the all-exponential model was a better fit.

$$f_{MOS\text{-}I} = \exp(An + Bd + C) \tag{8.2}$$

$$f_{MOS\text{-}P} = \exp(An + Bg + Cd + D) \tag{8.3}$$

8.2.3 Regression equations

The following are the regression equations found for I and P frames, and how they correlate to the data. The cross-correlation between the data and the estimations has been calculated by means of the Matlab `corrcoef` function and is shown in Table 8.1. The data used for the calculations can be found in Appendix F.

I frames

$$\begin{aligned} f_{MOS\text{-}I} &= \exp(An + Bd + C) \\ A &= -3.9314 \cdot 10^{-2} \\ B &= -1.6673 \cdot 10^{-03} \\ C &= 1.0591 \end{aligned} \tag{8.4}$$

P frames

$$\begin{aligned} f_{MOS\text{-}P} &= \exp(An + Bg + Cd + D) \\ A &= -9.7622 \cdot 10^{-02} \\ B &= 0.0019 \\ C &= -8.9328 \cdot 10^{-03} \\ D &= 1.3318 \end{aligned} \tag{8.5}$$

	$f_{MOS\text{-}I}$	$f_{MOS\text{-}I}$
Cross- correlation coefficient	0.850	0.905

Table 8.1: Correlation of the estimation with the data (MOS)

8.3 MOS gain estimation

In order to estimate MOS gain, the data has been obtained from the concealed sequences in the MOS tests. The MOS gain for each sequence is the difference between the MOS score of the concealed sequence and the SD sequence. The aim is to obtain f_{gain}, a function that will relate the characteristics of the artifacts detected by VIDC with the MOS gain of each sequence.

The variables taken into account have been:

- **Number of concealed macroblocks:** If the artifact is big (a lot of concealed MBs), the gain was expected to be significant, whereas for small errors, the possibility of the user not noticing a big difference was higher.

- **Frame type:** artifacts in I frames degrade quality much more that artifacts in P frames, hence the gain for I frames should be bigger than for P frames.

- **Difference frame:** Since the used concealment method is copy-paste, the concealment is expected to perform worse in sequences with a high degree of movement (the concealment will not fit as well as in almost-still sequences).

Since the MOS gain estimator proved to work well with this simple approach (Section 8.3.1), it was chosen not to make it more complex by also considering the GOP position. From what figure 8.9 shows, the relation between the concealed MBs and the MOS gain seems to be quadratic. For the difference (Figure 8.9) frame linear, this relation seems to be linear, with different slopes for I and P frames. It should also be noted, that as with the previous regression function, difference frame values of 0 are ignored (means nothing was detected).

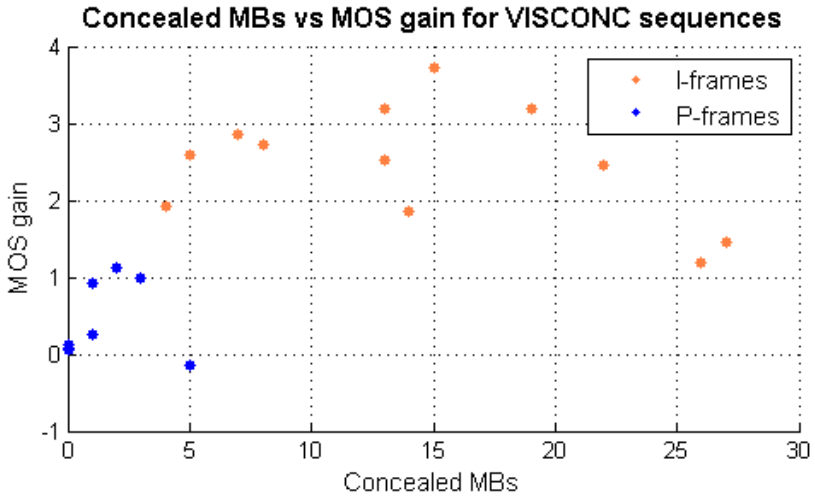


Figure 8.9: Concealed MBs vs. MOS gain

The regression function will be then as (8.6):

$$\begin{aligned} f_{gain}(n,d,t) &= \begin{cases} f_{gainI}(n,d) & \text{if } t = \text{I frame} \\ f_{gainP}(n,d) & \text{if } t = \text{P frame} \end{cases} \\ n &= \text{Concealed macroblocks} \\ d &= \text{Difference frame} \\ t &= \text{Frame type (I frame or P frame)} \end{aligned} \tag{8.6}$$

The DataFit software has also been used for finding this equation. From the shape of scatter graphs in figures 8.9 and 8.10 the following model has been used: $ax_1{}^2+bx_1+cx_2+d$.

Equations (8.7) and (8.8) show the exact regression models used for MOS gain estimation.

$$f_{(}gain\text{-}I) = An^2 + Bn + Cd + D \tag{8.7}$$

$$f_{(}gain\text{-}P) = An^2 + Bn + Cd + D \tag{8.8}$$

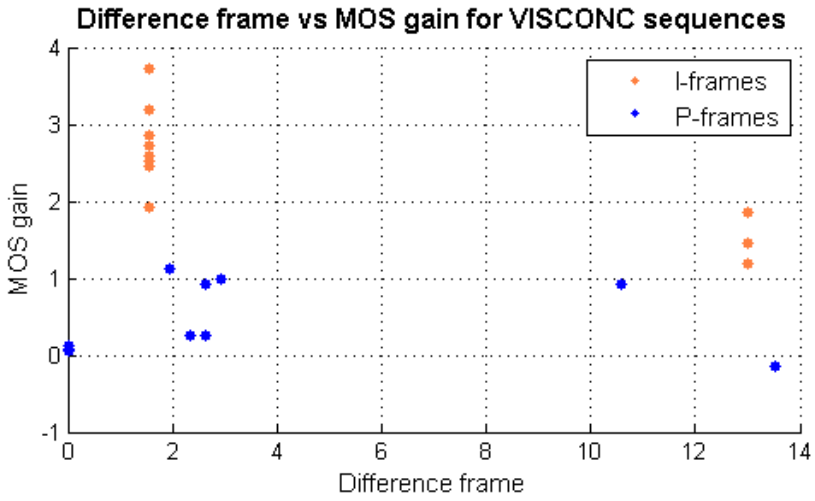


Figure 8.10: Difference frame vs. MOS gain

8.3.1 Regression equations

The following are the regression equations found for I and P frames, and how they correlate to the data. The cross-correlation between the data and the estimations has been calculated by means of the Matlab `corrcoef` function. The data itself can be found in annex G.

I frames

$$\begin{aligned} f_{(gain\text{-}I)} &= An^2 + Bn + Cd + D \\ A &= -6.4831 \cdot 10^{-03} \\ B &= 0.2039 \\ C &= -9.7279 \cdot 10^{-02} \\ D &= 1.6829 \end{aligned} \tag{8.9}$$

P frames

$$\begin{aligned} f_{(gain\text{-}P)} &= An^2 + Bn + Cd + D \\ A &= -0.169638919043378 \\ B &= 0.7850 \\ C &= 1.5430 \cdot 10^{-02} \\ D &= 2.3774 \cdot 10^{-02} \end{aligned} \tag{8.10}$$

	$f_{gain\text{-}I}$	$f_{gain\text{-}P}$
Cross- correlation coefficient	0.900	0.830

Table 8.2: Correlation of the estimation with the data (MOS gain)

8.4 Further applications

Although the MOS estimator should be considered preliminary work, this work could be used to develop a fully reference-free and codec-independent video quality estimator. This is because the estimator was not tested with sequences encoded with different GOP sizes and the chosen GOP size of 25 may be too small to fully appreciate temporal propagation.

Also, in a codec-independent video quality estimator, probably the use of syntax check should be dropped, as codecs normally don't have access to the data of UDP packets where the CRC checksum failed.

The employed 2-step approach makes it possible to use the algorithm not only to estimate the quality of error-concealed sequences but also unconcealed sequences.

Since the most promising use of this quality estimator is the mentioned fully reference-free codec-independent video quality estimator, it should be mentioned that due to an unintended side-effect, this is easier to implement now.

As mentioned in Section 8.1, f_{MOS} is obtained from VIDC output without SC information available. So, since f_{MOS} does not use the syntax check information (f_{gain} does, though), it could be potentially used as an add-on for any particular codec. Such a method, since it works only in the pixel domain would not need to interfere with the decoding process and should be possible to implement as a post-processing add-on of sorts.

Chapter 9

Conclusions

This work consists of the implementation, integration and testing of a Visual Impairments Detection and Concealment (VIDC) algorithm into a modified JM H.264/AVC decoder and the design of a quality predictor.

The objective of the work is to detect transmission errors in H.264 baseline-profile encoded QCIF resolution (176x144 pixels) video sequences and the estimation of their impact on subjective quality by means of the quality estimator. Concealment of the detected artifacts is also performed, but it is not one of the main goals of the project.

Transmission errors can cause a desynchronization of the codewords in the bitstream, caused by the usage of Variable Length Coding (VLC). Such desynchronizations can lead to visual artifacts that could spatially and temporally propagate, due to spatial and temporal prediction exploitation by the codec. Those artifacts will most times cause the perceptual quality of the video sequence to greatly decrease. It is by means of SC and VIDC that those visual artifacts are detected and concealed. It should be noted that the aim of this work is on detection though, and as such a very simple concealment method has been used.

In order to detect transmission errors, two interacting strategies are used: Syntax Check (SC) analysis and the mentioned VIDC. While both work at different levels, they are designed to work together.

SC works by detecting when the decoder is lead to perform an illegal action due to illegal or incorrect codewords being read from a desynchronized bitstream. VIDC, on the other hand, works by analyzing the decoded frames at pixel level in search of visual artifacts.

The modified decoder, in which the VIDC algorithm was embedded, was a JM v.10.2 reference decoder in which the SC algorithm had already been added. Such decoder is capable of decoding erroneous stream without crashing, producing what we call Straight Decoded (SD) sequences. In those sequences the transmission errors manifest themselves as visual artifacts of varying visibility. The objective was to translate the VIDC algorithm from an existent standalone Matlab version to C and then embed it into the JM+SC decoder. In this way, VIDC would be able to read and write from and to the decoded picture buffer and read directly from SC the detected errors. After the implementation, the code underwent some changes and modifications so as to increase its performance. Tests showed that it effectively increased both the objective (Y-PSNR) and subjective (MOS) quality of sequences with transmission errors.

We propose SC+VIDC as a means to detect and correct transmission errors in H.264/AVC

sequences. In Y-PSNR and MOS tests, it has been shown that SC+VIDC boosts both objective and perceived quality of the decoded sequences notably, specially in I frames (Table 9.1). Although the Y-PSNR gain may not seem impressive, the gain in terms of MOS is very significant.

Frame type	Y-PSNR gain	MOS gain
I frame	3 dB	2.48
P frame	0.3 dB	0.47

Table 9.1: Video quality improvement

As mentioned before, one of the objectives of the work was to design a quality estimator. Such estimator would use the data about detected artifacts outputted by the SC+VIDC algorithm and be able to estimate the subjective quality of the decoded sequence.

The estimator is capable of outputting an estimated perceptual quality of an H.264/AVC decoded sequence by using the following parameters regarding the detected artifacts:

- Frame type: the appearance of the artifact will vary greatly depending of the frame type. I frames have much more visible than P frames, for instance.
- Artifact size: size of the artifact in MacroBlocks (MBs).
- GOP position: expresses the distance in frames between the frame that contains the artifact and the last I frame. Is serves as a measure of temporal propagation.
- Difference frame: measures the pixel-wise difference between the frame with the artifact and the previous one and gives an idea of the camera movement. The more movement there is, the more the artifact could spread temporally and the worse the detection algorithm works.

The MOS estimation process has been split into two. The algorithm first estimates the MOS of an SD sequence and then the gain that an error concealment would introduce. This approach allows to estimate the MOS of concealed as well of unconcealed sequences.

Figure 9.1 depicts how the process has been split. First, the quality of the SD sequence is estimated (*SD sequences MOS*). Also, the MOS gain that performing concealment in that sequence would introduce is estimated (*Concealment MOS gain*). Combining these two values, the estimated MOS of a concealed erroneous sequence can be obtained.

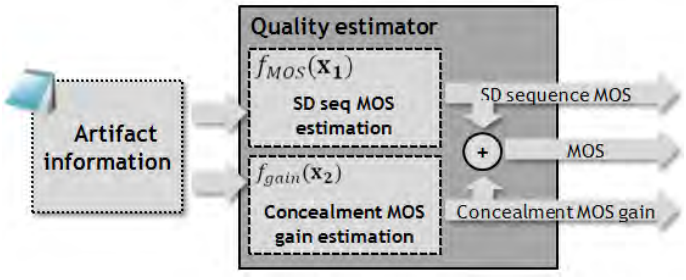


Figure 9.1: Quality estimator design

As it can be seen in Table 9.2, the estimations correlate well with the data obtained from the MOS tests (Chapter 7). In those tests, people were asked to rate the perceived quality of

a series of concealed and unconcealed erroneous sequences. From the obtained results, data correlates best for I frames. This is caused by errors in I frames producing more predictable and visible artifacts than in P frames.

	I frames	P frames
MOS	0.850	0.905
MOS gain	0.900	0.830

Table 9.2: Correlation of the estimations with the data

An interesting application of this quality estimator would be in the implementation of a fully reference-free and codec-independent estimation of video quality. Since VIDC works only in the pixel domain, such an application could work as a post-processing video quality estimator for any given codec. Although it would not be able to use information from SC, as seen in chapter 8, this does not pose a big problem. This quality estimator would analyze the decoded frames and could generate a MOS value measuring the perceptual quality of the decoded video.

Future work with the SC+VIDC decoder could involve improvements focusing on the concealment methods or adding additional error detection strategies. Rather than copy-paste, the method used for artifact concealment, other methods may yield better results [12]. Also, other detection methods, such as watermarking [15] could improve detection probability. Since VIDC is only post-processing, if watermarking detects an error, VIDC could decide whether the MB affected by the error will be affected by impairments or not.

Bibliography

[1] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG. H.264/mpeg-4 avc reference software manual, 2007. [cited at p. 1, 33]

[2] ITU-T. Rec. H.264: Advanced video coding for generic audiovisual services, 2005. [cited at p. 3, 10]

[3] ISO/IEC. 14496-10:2005, coding of audio-visual objects part 10: Advanced video coding, 2005. [cited at p. 3, 10]

[4] N. Kamaci and Y. Altunbasak. Performance comparison of the emerging H.264 video coding standard with the existing standards. In *IEEE Int. Conf. Multimedia and Expo*, 2003. [cited at p. 3]

[5] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7), July 2003. [cited at p. 3]

[6] Technical Specification Group Services 3rd Generation Partnership Project and System Aspects. Transparent end-to-end packet-switched streaming service (pss); protocols and codecs (release 6), ver. 6.8.0, 2006. [cited at p. 4, 9]

[7] R. Schaefer, T. Wiegand, and H. Schwarz. The emerging H.264/avc standard. *EBU Technical Review*, January 2003. [cited at p. 6]

[8] Technical Specification Group Services 3rd Generation Partnership Project and System Aspects. Packet switched conversational multimedia applications; default codecs (release 6), ver. 6.4.0, 2005. [cited at p. 9]

[9] J. Postel. Rfc 768, user datagram protocol. Technical report, IETF, 1980. [cited at p. 9]

[10] L. Superiori, O. Nemethova, and M. Rupp. Performance of a H.264/avc error detection algorithm based on syntax analysis. *International Conference on Advances in Mobile Computing and Multimedia (MoMM), Yogyakarta, Indonesien*, March 2006. [cited at p. 15, 17, 22]

[11] L. Superiori, O. Nemethova, and M. Rupp. Detection of visual impairments in the pixel domain of corrupted H.264/avc packets. *Picture Coding Symposium, Lissabon, Portugal*, July 2007. [cited at p. 15, 23]

[12] O. Nemethova, A. Al Moghrabi, and M. Rupp. An adaptive error concealment mechanism for h.264/avc encoded low-resolution video streming. *Proc. of 14th European Signal Processing Conference (EUSIPCO)*, September 2006. [cited at p. 19, 83]

[13] Karsten Suehring. Jm software webpage, http://iphome.hhi.de/suehring/tml/. [cited at p. 33]

[14] ITU-T. Rec. p.800: Methods for subjectivedetermination of transmission quality, 1996. [cited at p. 65]

[15] Olivia Nemethova, Gonzalo Calvar Forte, and Markus Rupp. Robust error detection for h.264/avc using relation based fragile watermarking. *Proc. of Int. Conf. on Systems, Signals and Image Processing (IWSSIP), Budapest, Hungary*, September 2006. [cited at p. 83]

Appendix A

Y-PSNR testing error patterns

Tables A.1 and A.2 show the positions in the H.264 encoded bitstream where errors were inserted to test final decoder implementation by means of the Y-PSNR of the decoded video. The patterns were randomly generated with Matlab using a $BER = 10^{-5}$ and taking into account that the H.264 encoded file was 372,678 bytes.

1	2	3	4	5	6	7	8	9	10
69552	20223	3351	69252	52793	72189	60618	64354	136497	275822
145866	51410	297518	350900	141223	153945	136704	173269	274206	332591
201841	127876	484429	497242	149842	245769	158943	178745	342086	405321
206610	146211	491790	540964	176261	296251	231577	216083	452489	813804
211980	185743	515618	562949	250487	297628	268287	402566	518785	855671
355293	217515	631576	581183	259266	494121	365053	510533	616706	1000894
464794	303084	655068	587572	331410	598904	547856	555915	664941	1235754
496023	541146	731580	696441	393776	619301	563674	685888	804565	1369318
524179	571842	787966	1221652	397231	675299	595730	751130	993163	1476730
606129	587481	848449	1323669	408320	753316	735157	799498	999207	1615413
711884	670745	987024	1447055	560190	844366	1110475	805916	1051249	1641366
721421	722200	1051924	1521117	685085	846315	1261751	883245	1363972	1659540
904824	787553	1057250	1628935	753294	848624	1528099	1073394	1662524	1676580
954331	1127649	1061950	1642049	825431	892694	1679082	1204836	1690805	1750439
975684	1150495	1093474	1827835	907998	923453	1711824	1216073	1779883	1983339
1016244	1305991	1231101	1840874	972405	947924	1738796	1471042	1791582	2027512
1067059	1449185	1285197	1985405	992524	1065603	1841626	1472184	1939208	2063170
1091575	1665551	1328045	1992033	1381680	1081747	1918512	1490342	2000596	2102786
1137908	1676253	1345237	2100116	1473356	1088642	1994178	1515308	2176450	2139127
1154829	1781637	1457342	2121546	1714465	1148570	2031145	1566823	2185766	2197876
1178488	1814328	1478398	2160946	1792308	1227030	2049227	1591401	2212769	2535364
1328142	1872818	1660035	2206445	2151932	1237275	2135559	1835406	2252423	2640257
1620337	2072907	1671779	2211435	2204204	1499958	2178133	1882645	2776109	2665504
1802319	2518075	1741658	2621125	2219644	1519388	2294590	1998717	2847563	2733824
1851386	2535340	1756939	2720510	2354969	1568159	2845546	2030002	2900075	2809995
2102493	2573953	1829522		2465935	1603533		2105188		2814905
2287098	2612003	1841115		2495839	1694945		2306626		
2466776	2702829	1860798		2651918	1827321		2417516		
2525729	2705039	1891050		2654177	2006168		2451743		
2706140	2726254	2011967		2814806	2006452		2566701		
2726940	2737347	2086815		2876820	2085211		2602890		
2738136	2847307	2280633			2271925		2682322		
2786153	2893761	2373610			2273459		2758964		
2795934	2937590	2775400			2306563		2910785		
2799341		2790526			2494765		2925149		
		2797015			2900448				
					2910438				
					2961452				

Table A.1: Y-PSNR testing error patterns 1-10

11	12	13	14	15	16	17	18	19	20
153238	59159	391212	47345	21007	210092	161195	8890	78387	112292
412287	306594	741008	168397	105300	243241	249286	114397	86424	309255
481016	381592	827202	259475	293597	371709	257941	149014	317806	839216
510711	386847	906639	338470	320471	412424	346577	473848	443080	844242
579260	423473	990571	496388	324462	446281	485795	488386	464762	1013379
667914	434709	997926	537514	425931	448826	795663	762823	657085	1095124
674192	446309	1056439	547092	686851	838916	890288	821937	753476	1191145
735140	463541	1219397	588294	793397	929910	982454	890494	866239	1264529
817854	550336	1311038	613988	1144844	1157551	1071887	981837	875048	1286260
942062	634959	1327581	845926	1236866	1209228	1169048	1032999	880670	1478599
976108	943488	1486309	1098990	1259273	1225735	1190461	1100407	891390	1603859
1225622	960011	1536421	1149780	1429202	1228787	1321637	1241177	917917	1678220
1352669	987103	1547241	1233431	1456766	1417442	1327478	1403736	1135692	1707414
1377143	1140906	1635940	1301409	1524398	1756796	1375582	1475935	1140876	1733419
1402913	1175675	1663055	1403332	1709870	1877001	1440047	1500311	1172490	1748975
1465073	1407467	1742364	1565054	1812286	1890702	1629081	1506365	1275434	1881634
1518148	1487454	1758117	1641099	1967384	1953722	1790786	1927423	1300176	1953329
1554143	1695202	1779077	1649440	2041684	1982911	1845813	1927584	1304124	2222731
1724966	1765293	1789501	1674929	2108988	2022422	1898965	1933113	1358654	2521903
1959091	1792861	1854210	1787762	2135627	2066096	2031739	2044224	1449292	2639488
1991215	1991616	1867985	1858161	2180342	2250006	2254754	2210518	1475294	2723565
1996664	2232432	1965461	1879544	2246033	2589852	2258446	2340393	1479819	2727496
2145816	2282523	2088999	1922591	2331880	2720196	2374114	2550290	1551881	2814477
2467757	2409097	2105578	2153693	2357521	2744463	2495096	2646178	1572369	
2549545	2517838	2123262	2162012	2518238	2922538	2570433	2663034	1707522	
2587770	2749707	2130389	2572406	2633841		2623499	2696169	1842554	
2627937	2808733	2268738	2769177	2743669		2632360	2884730	1962252	
2632027	2881681	2568822	2888432	2785648		2646682	2918777	2027674	
2661999	2924998	2615632	2924406	2857312		2792637	2948362	2154753	
2711364	2946878	2673431	2936736	2936524		2838620		2159864	
2751335		2765512				2917588		2498876	
2844694		2792707				2924472		2607236	
2862034		2950798							
2866376		2976366							
2871819									

Table A.2: Y-PSNR testing error patterns 11-20

Appendix B

Y-PSNR test results

This appendix consists of the results of the Y-PSNR for all of the 20 test sequences mentioned in section 6.2. The 20 figures depict both the Y-PSNR gains for each frame for the foreman sequence using the CRC checksum and without.

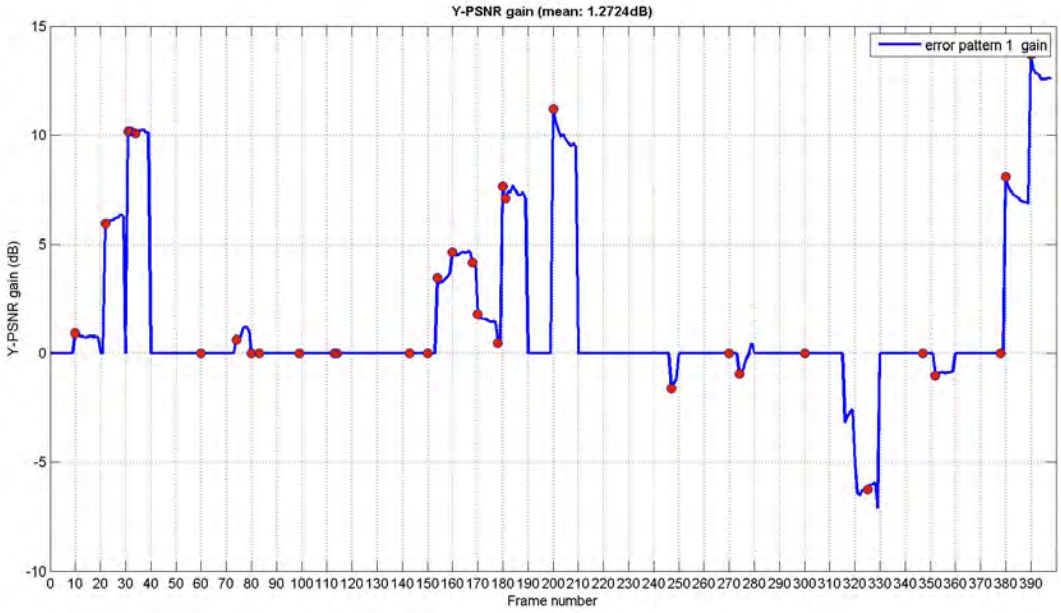
The upper plot represents always the gain without using the CRC validation. The lower plot represents the gain when using the CRC validation.

From the results shown in the appendix, maybe one result will seem surprising. It is the one shown in figure B.8. Although the artifact is correctly detected, the quality degradation is still high. And adding the CRC validation does not make things better.

This specific case is due to the fact that the concealment that is used is very simple. Although to the human eye the subjective quality of the concealed picture would be in most cases perceived as superior to the unconcealed one, Y-PSNR shows a degradation.

(a) SD frame (b) Concealed frame

Figure B.1: Y-PSNR degradation due to concealment



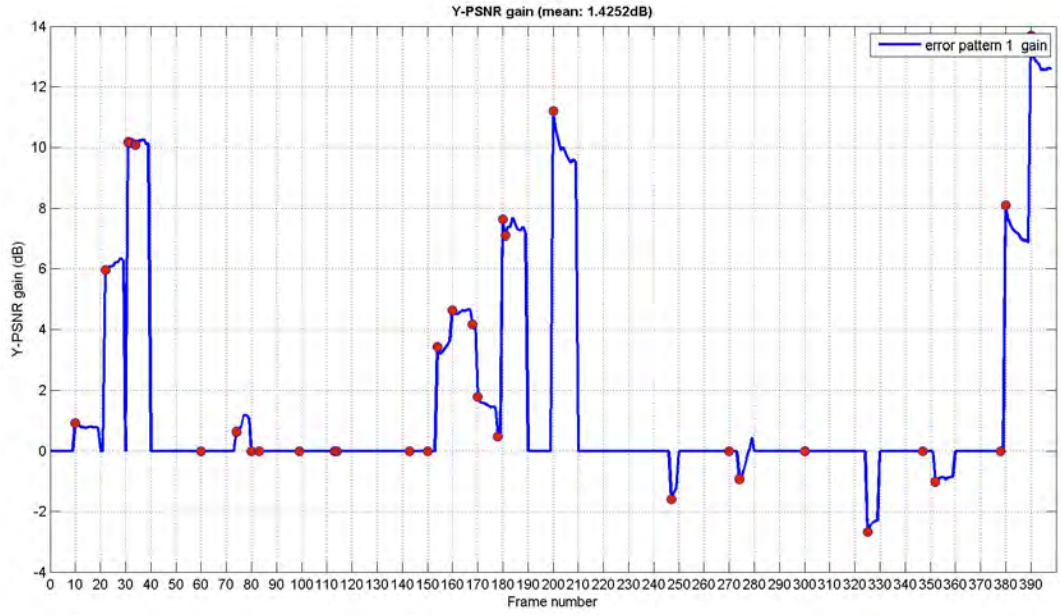


Figure B.2: Y-PSNR gain for sequence 1

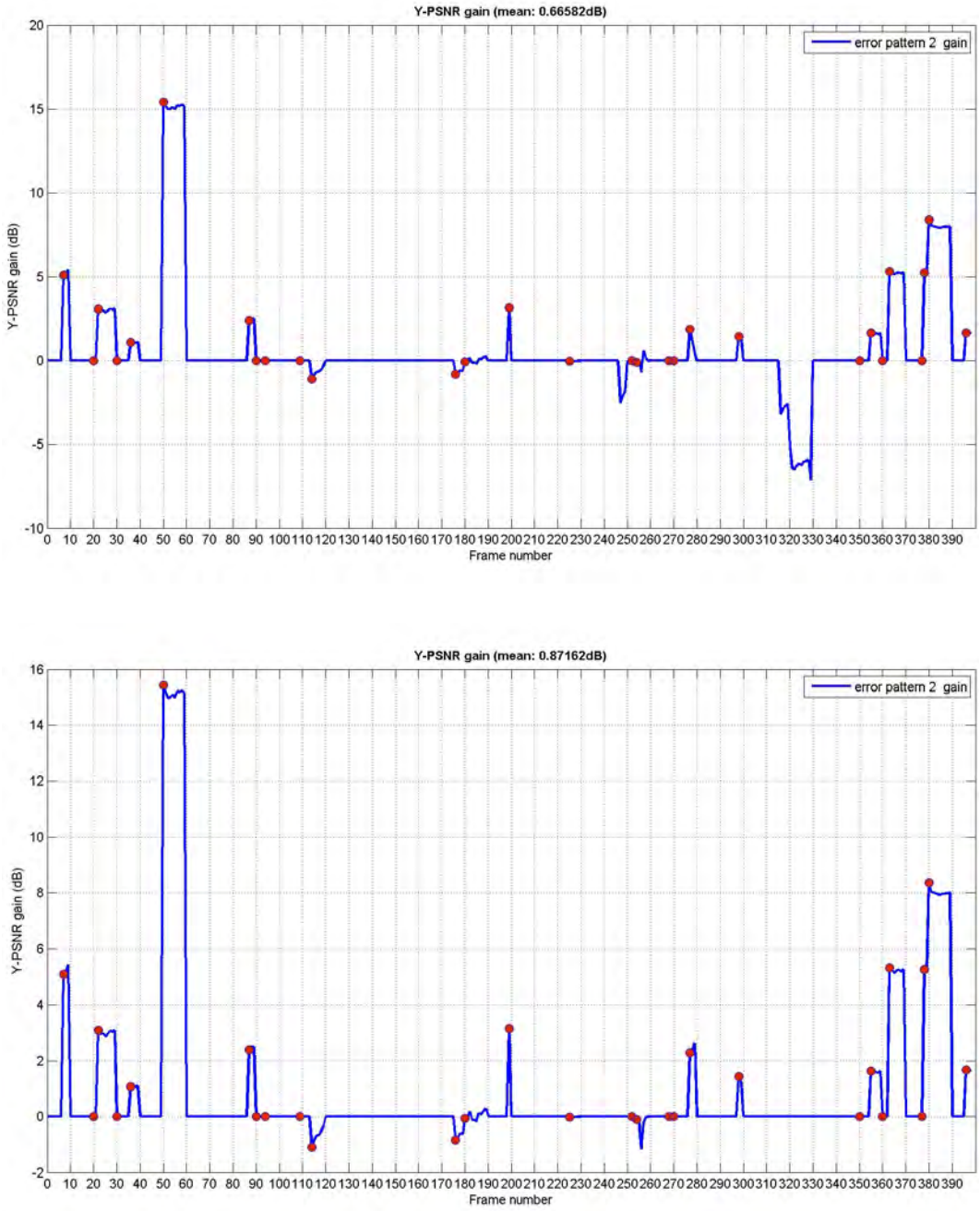


Figure B.3: Y-PSNR gain for sequence 2

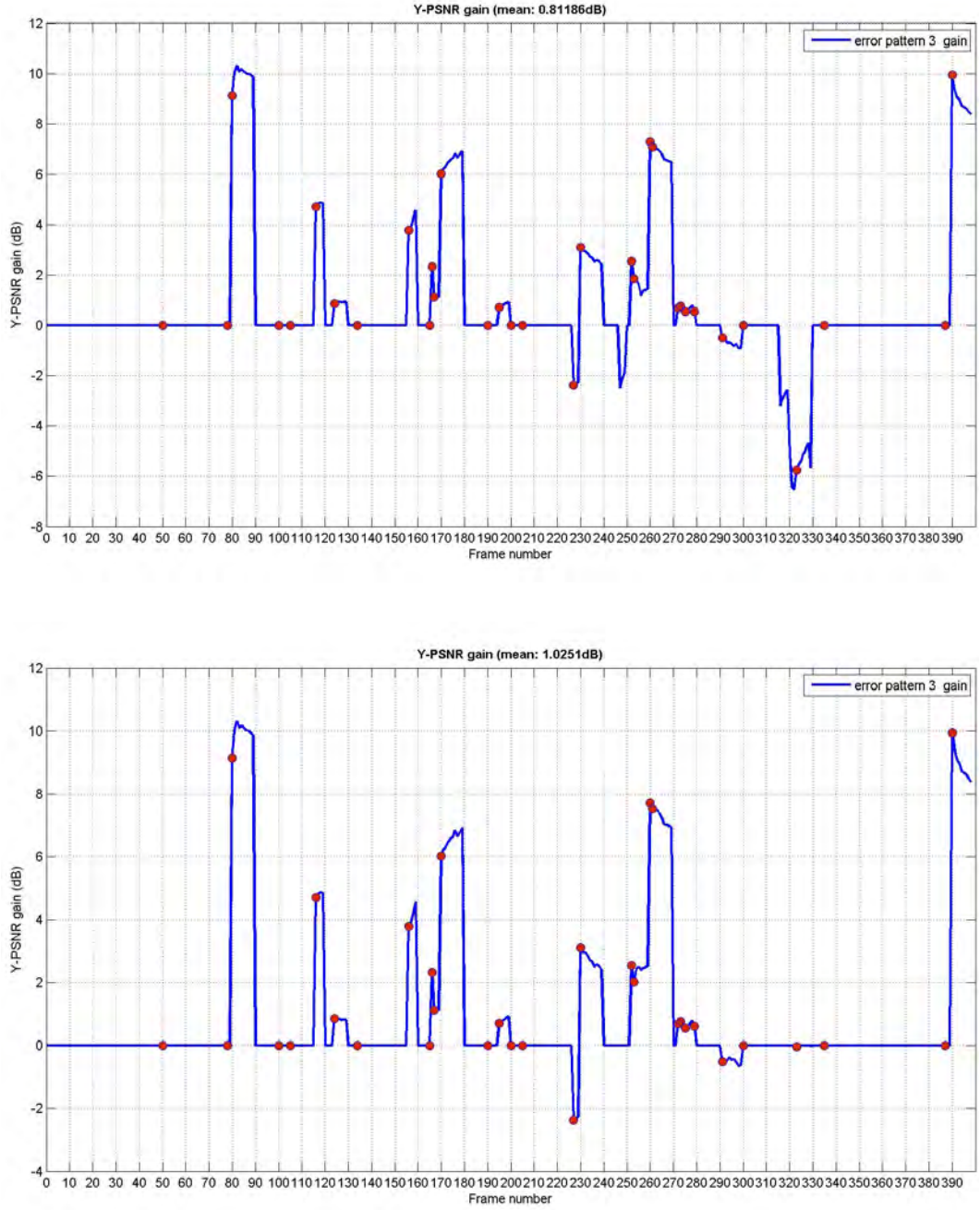


Figure B.4: Y-PSNR gain for sequence 3

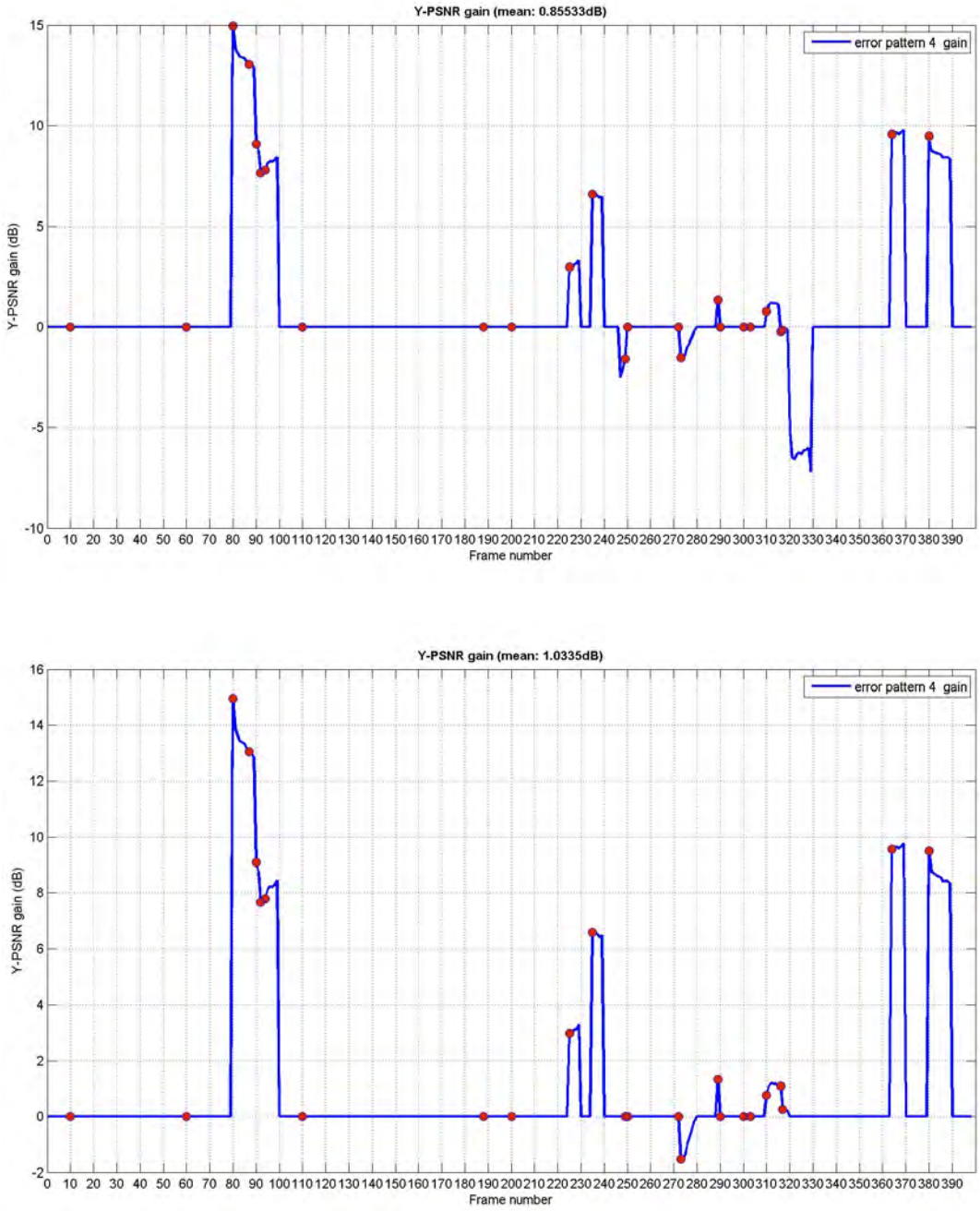


Figure B.5: Y-PSNR gain for sequence 4

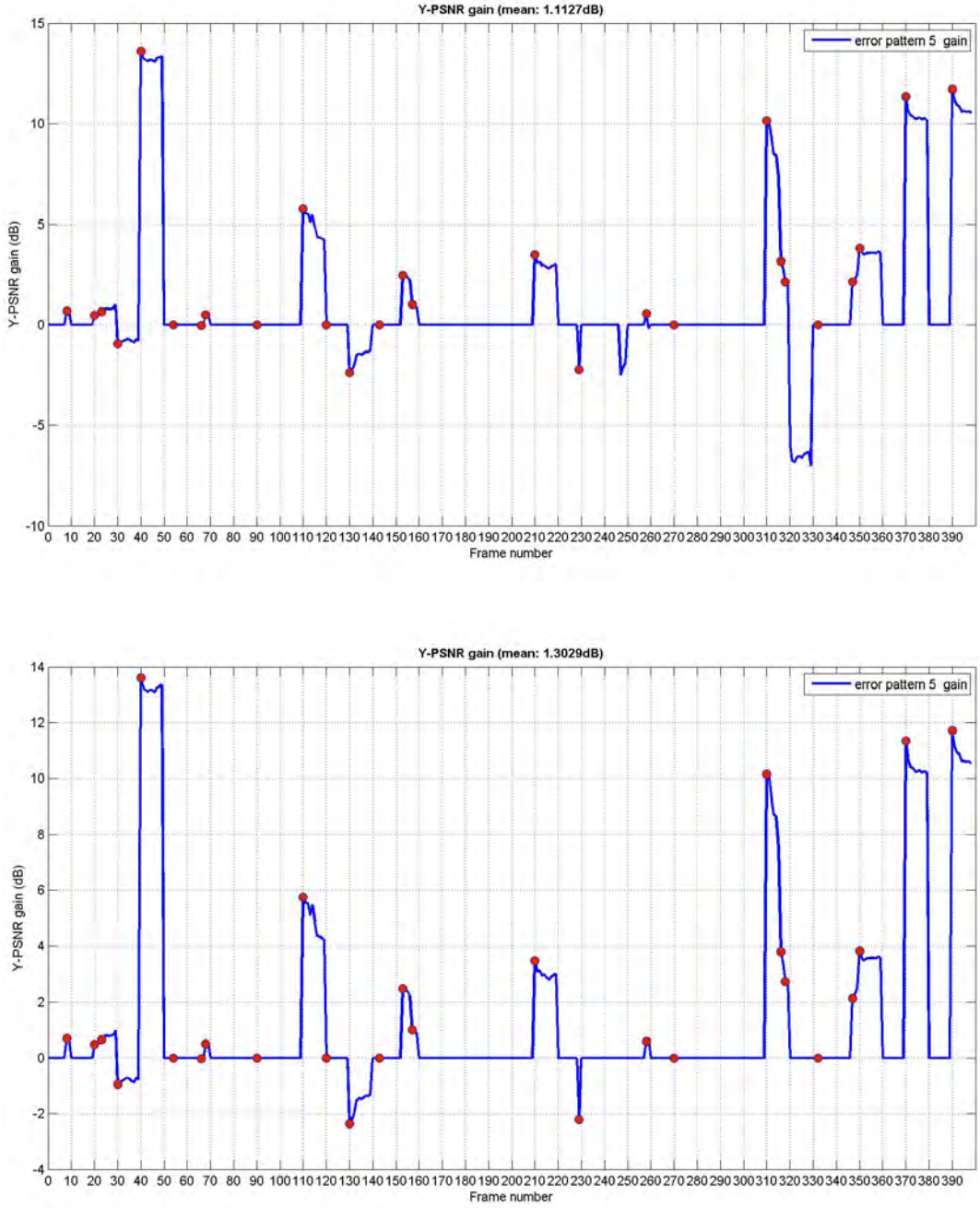


Figure B.6: Y-PSNR gain for sequence 5

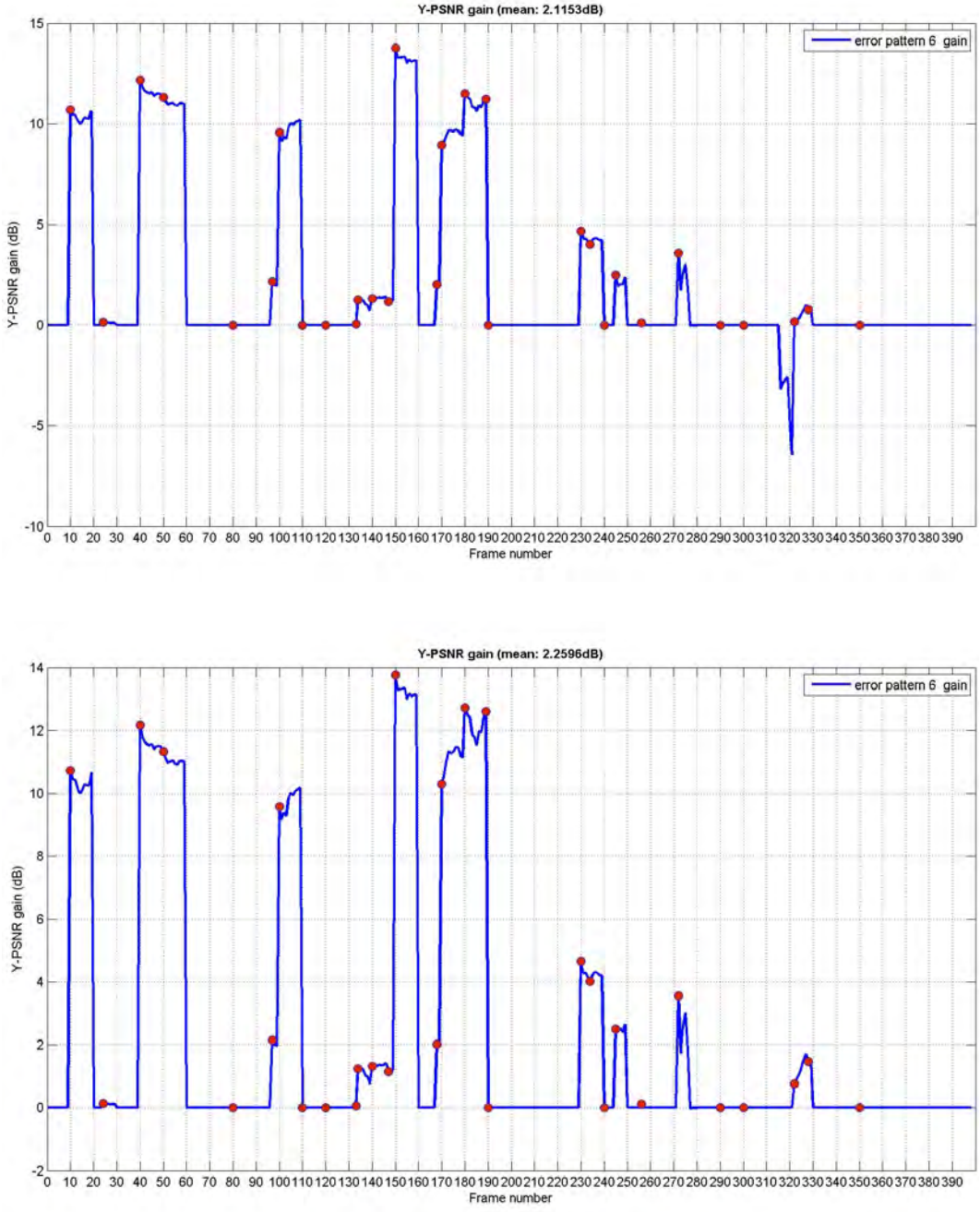


Figure B.7: Y-PSNR gain for sequence 6

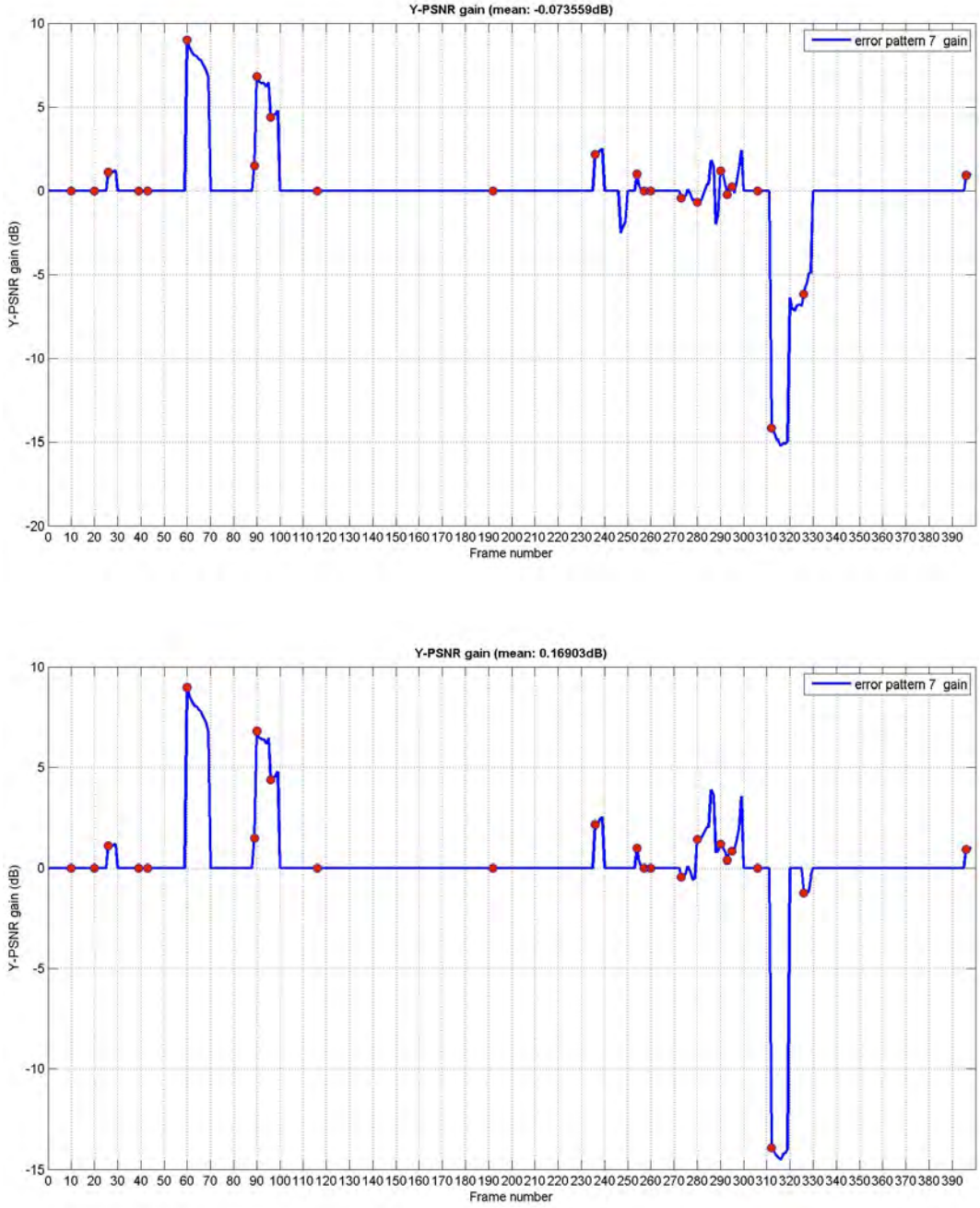


Figure B.8: Y-PSNR gain for sequence 7

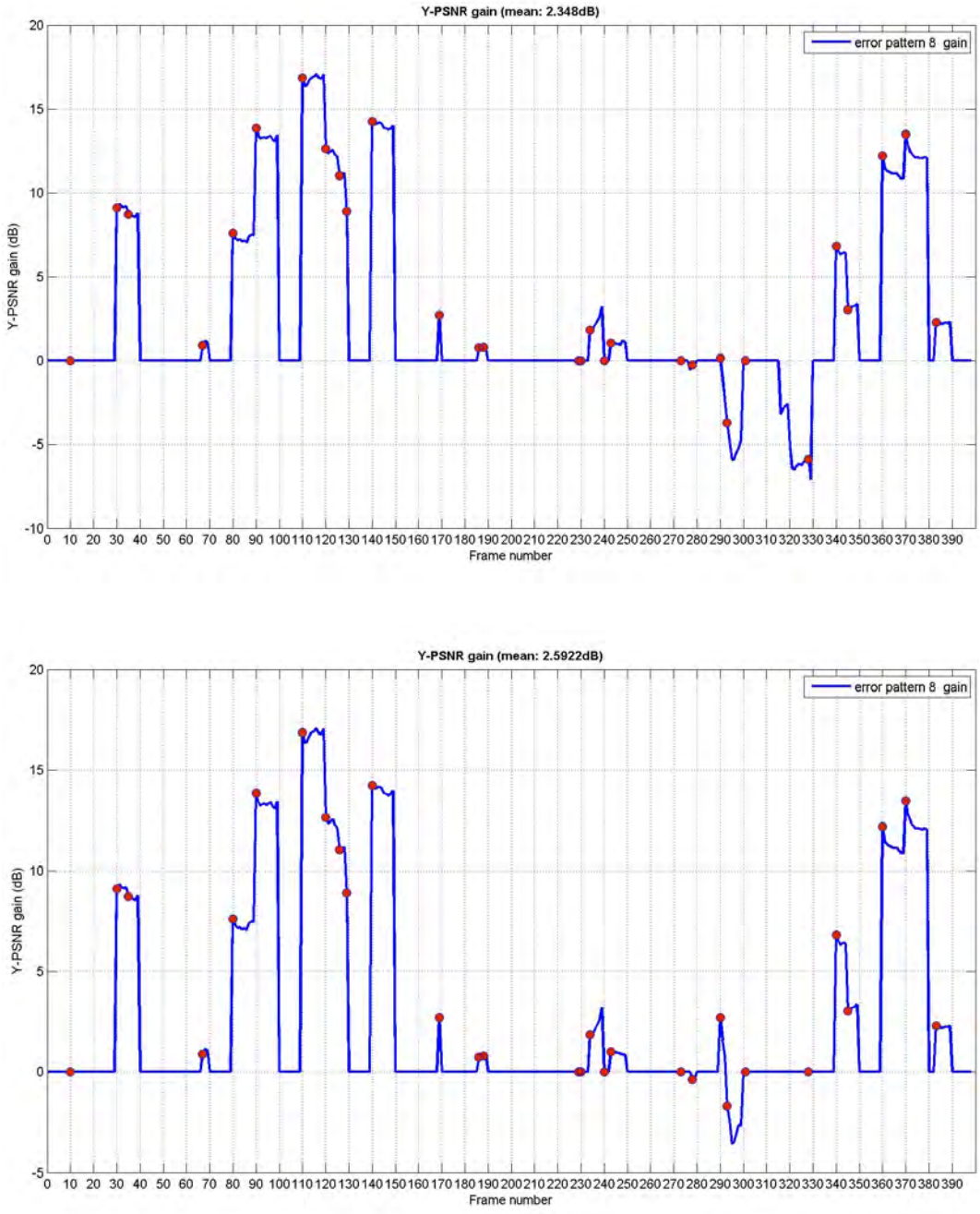


Figure B.9: Y-PSNR gain for sequence 8

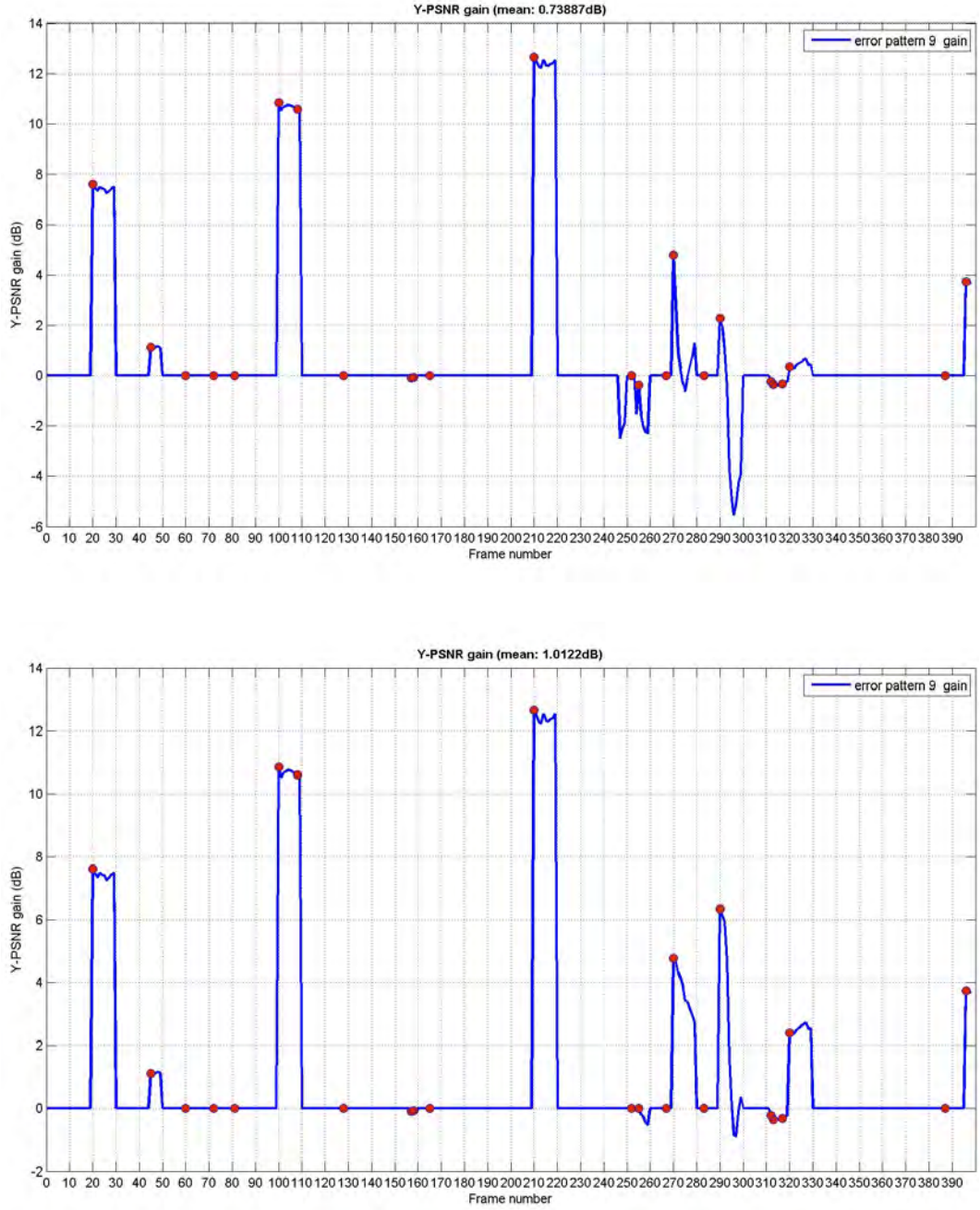


Figure B.10: Y-PSNR gain for sequence 9

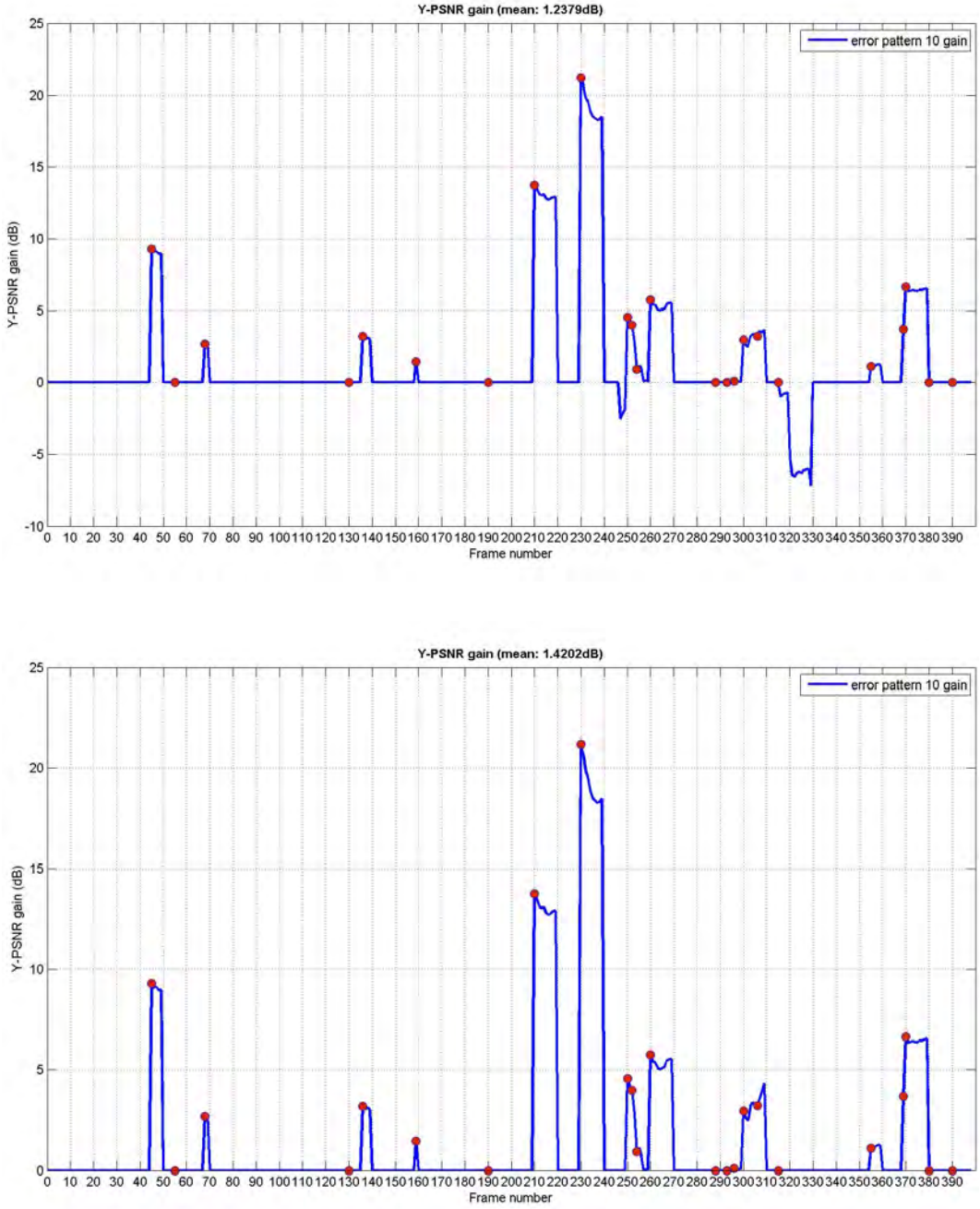


Figure B.11: Y-PSNR gain for sequence 10

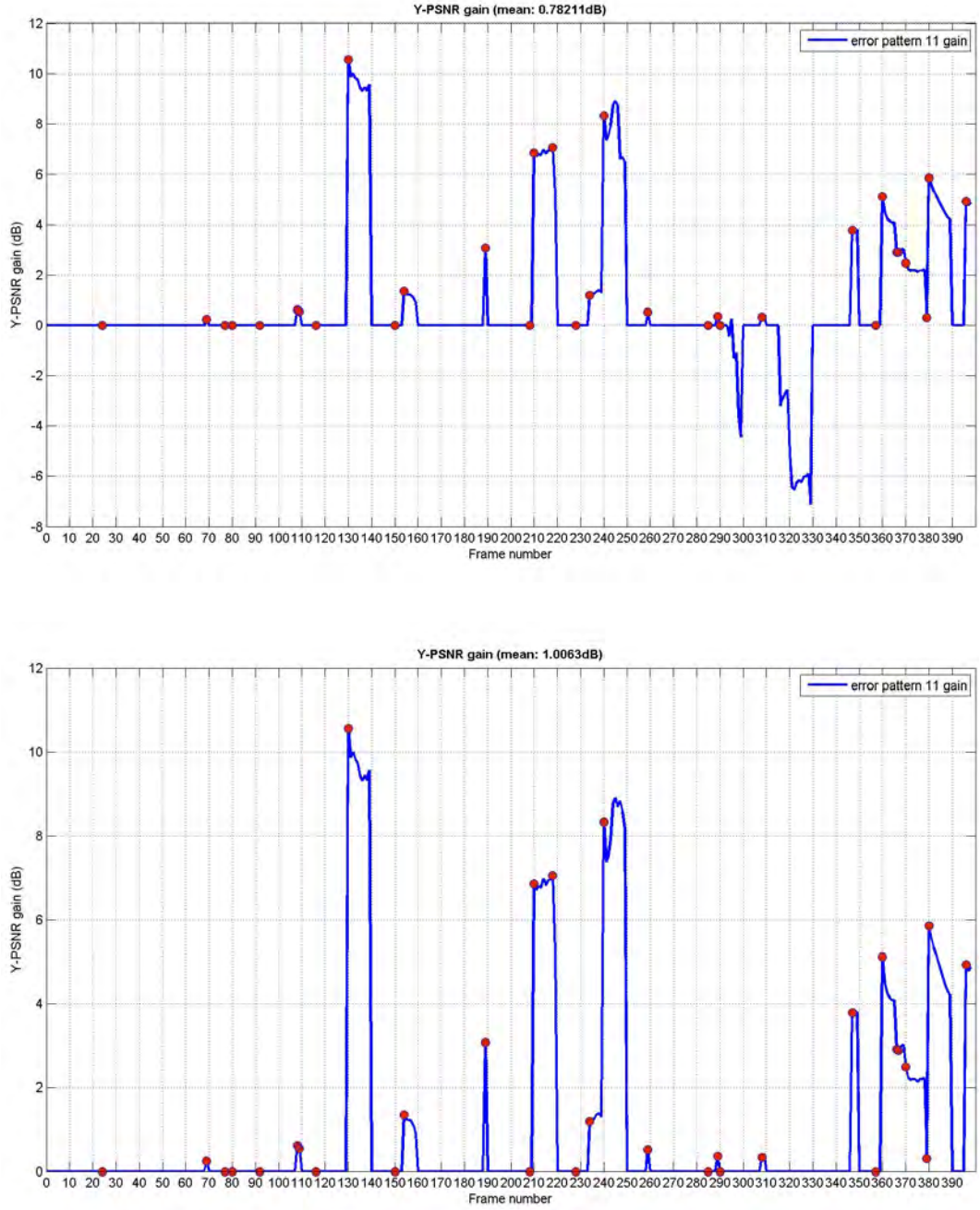


Figure B.12: Y-PSNR gain for sequence 11

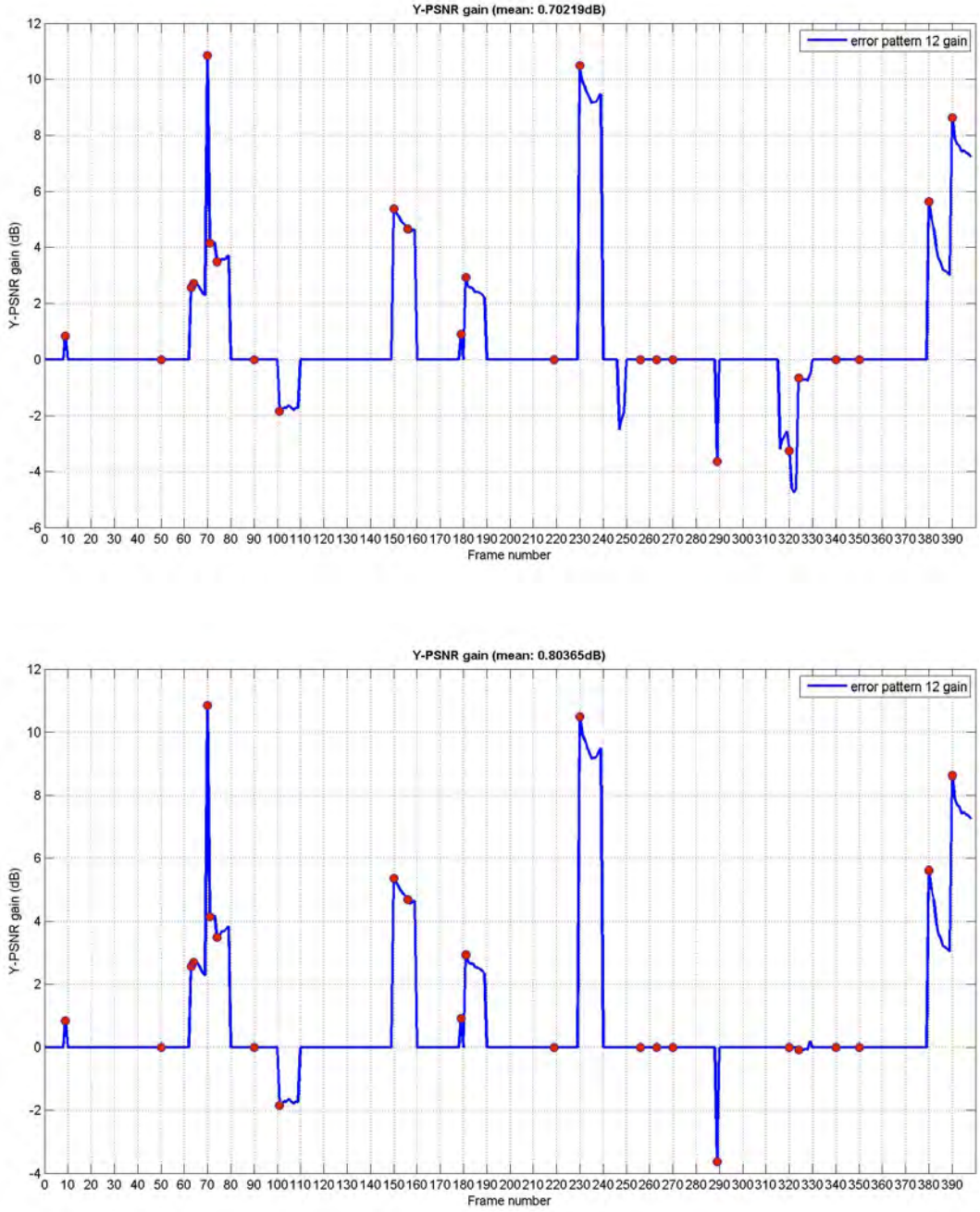


Figure B.13: Y-PSNR gain for sequence 12

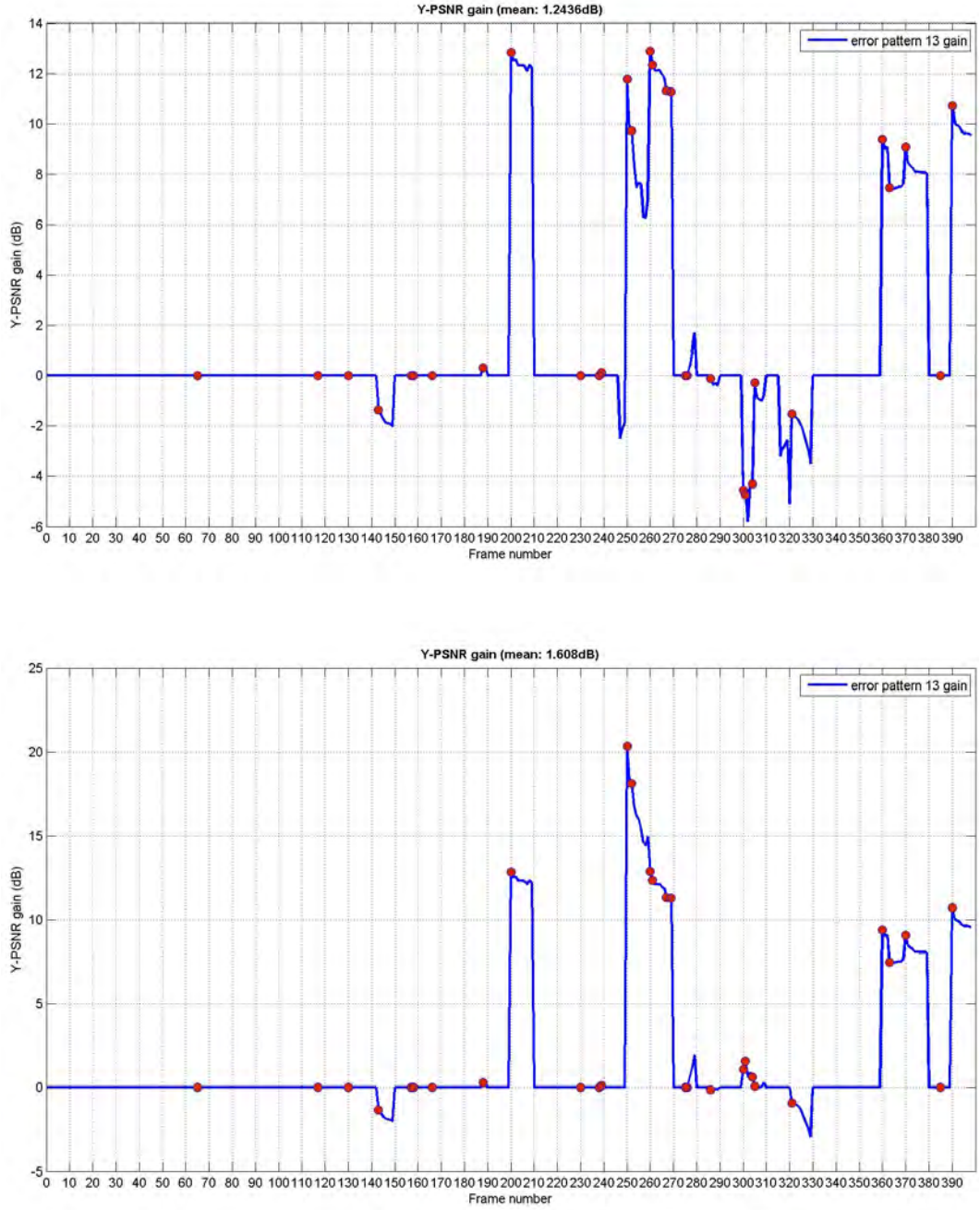


Figure B.14: Y-PSNR gain for sequence 13

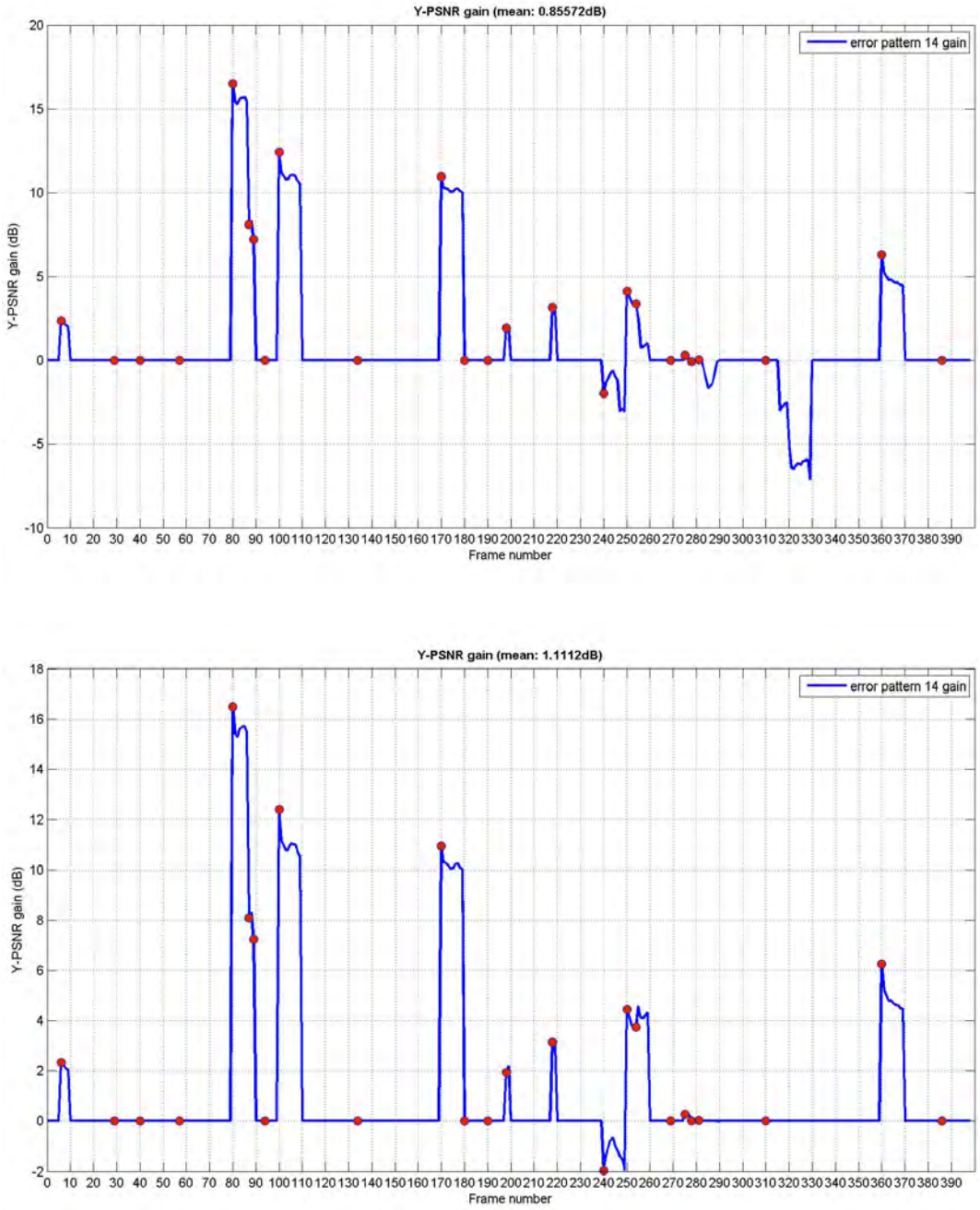


Figure B.15: Y-PSNR gain for sequence 14

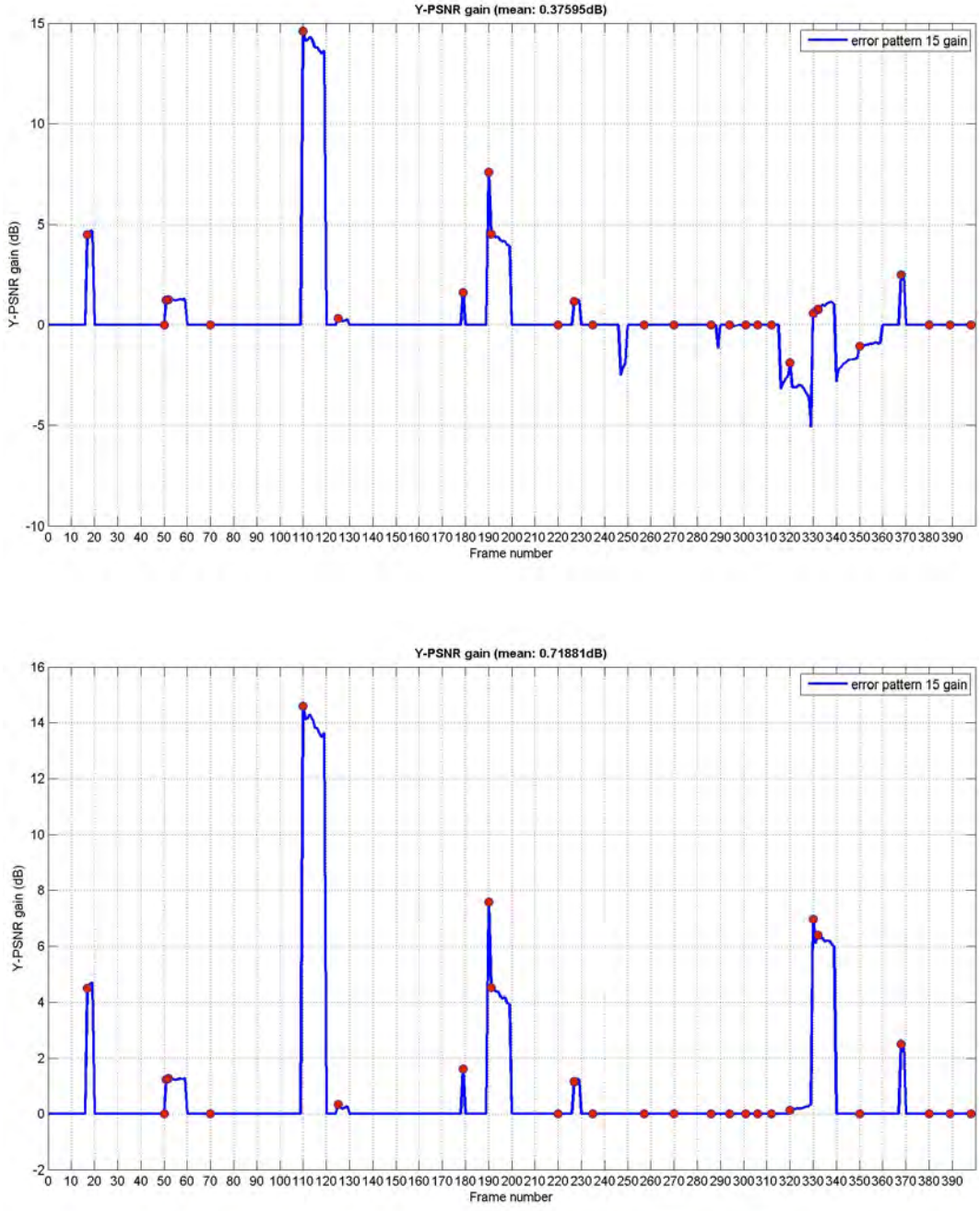


Figure B.16: Y-PSNR gain for sequence 15

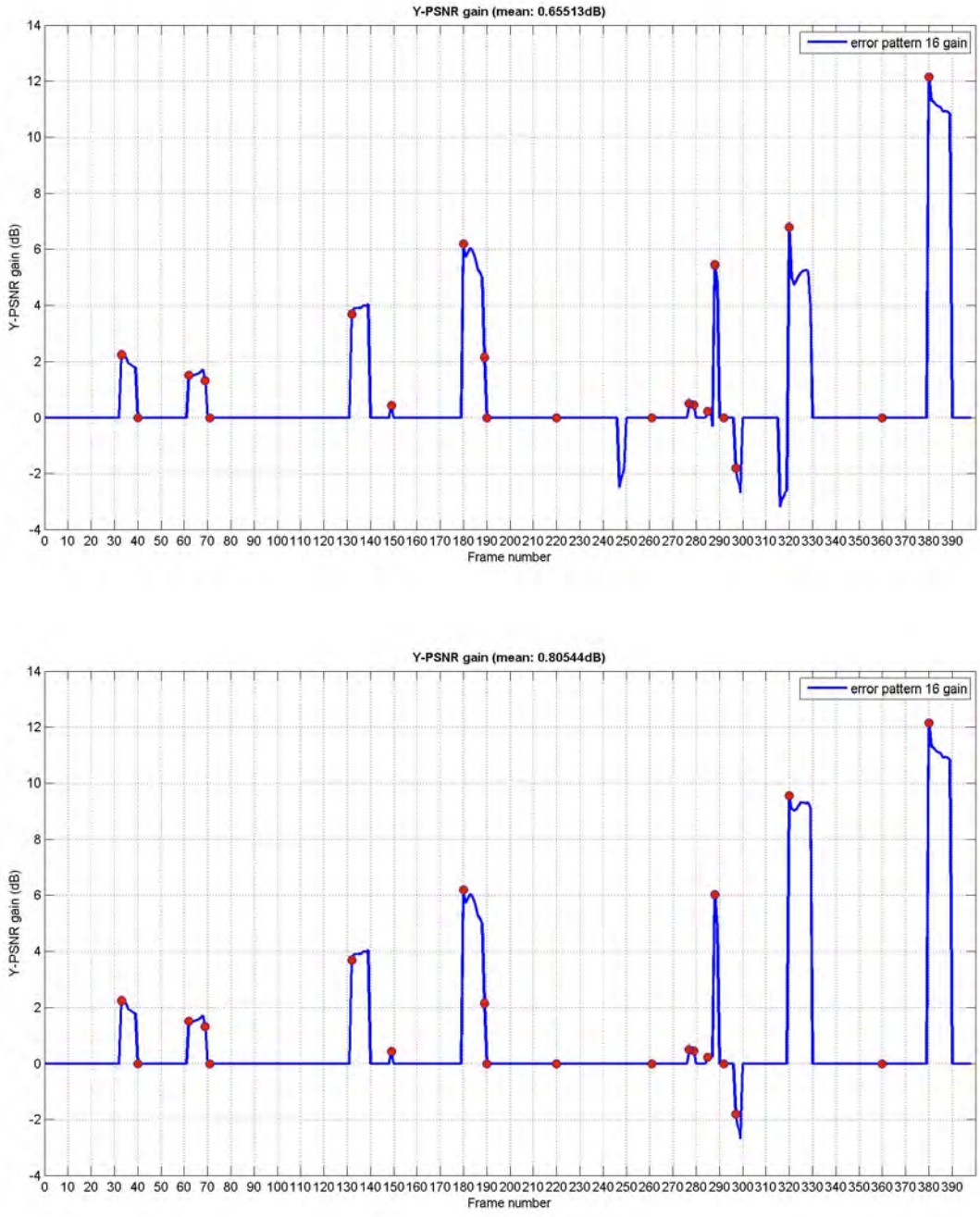


Figure B.17: Y-PSNR gain for sequence 16

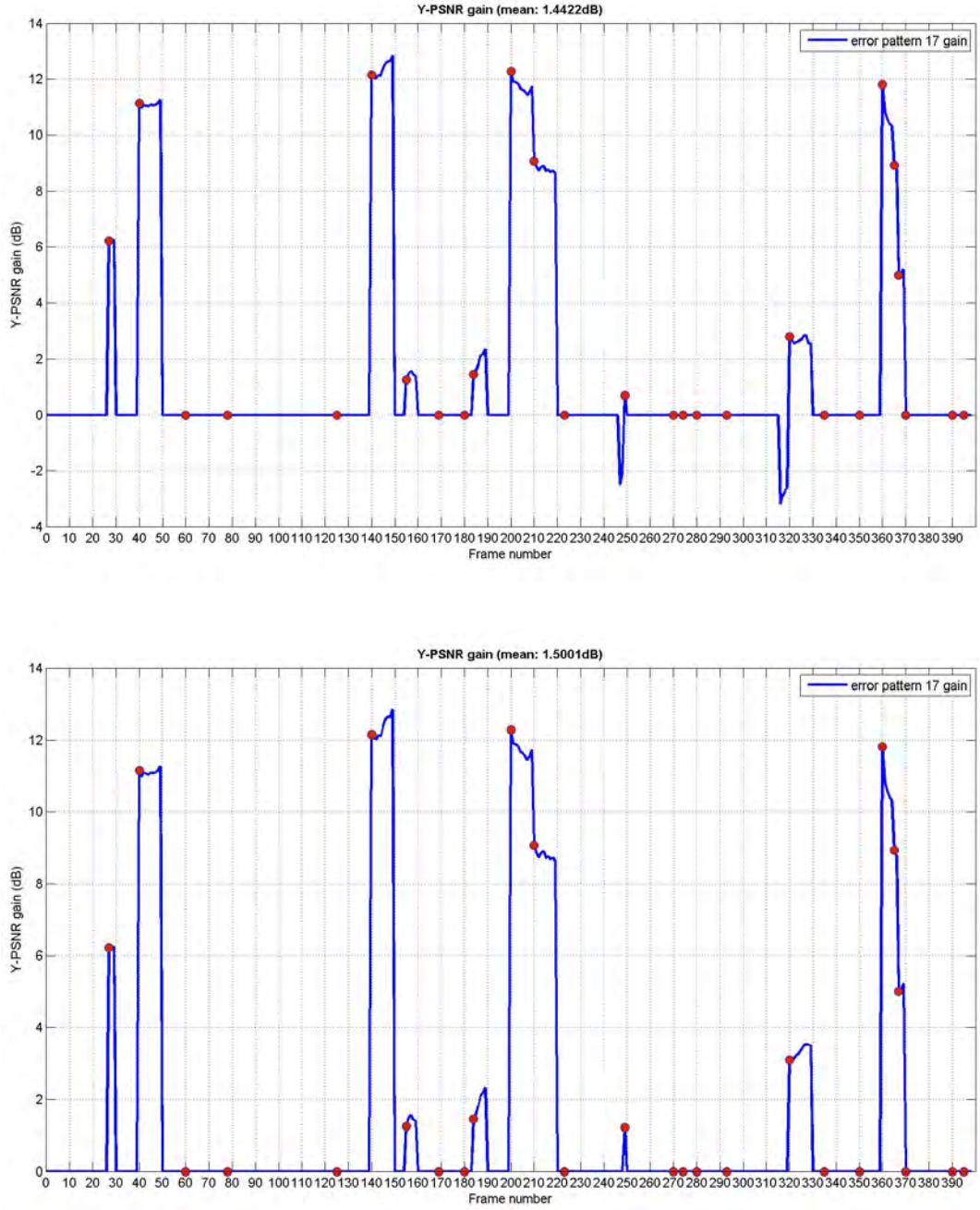


Figure B.18: Y-PSNR gain for sequence 17

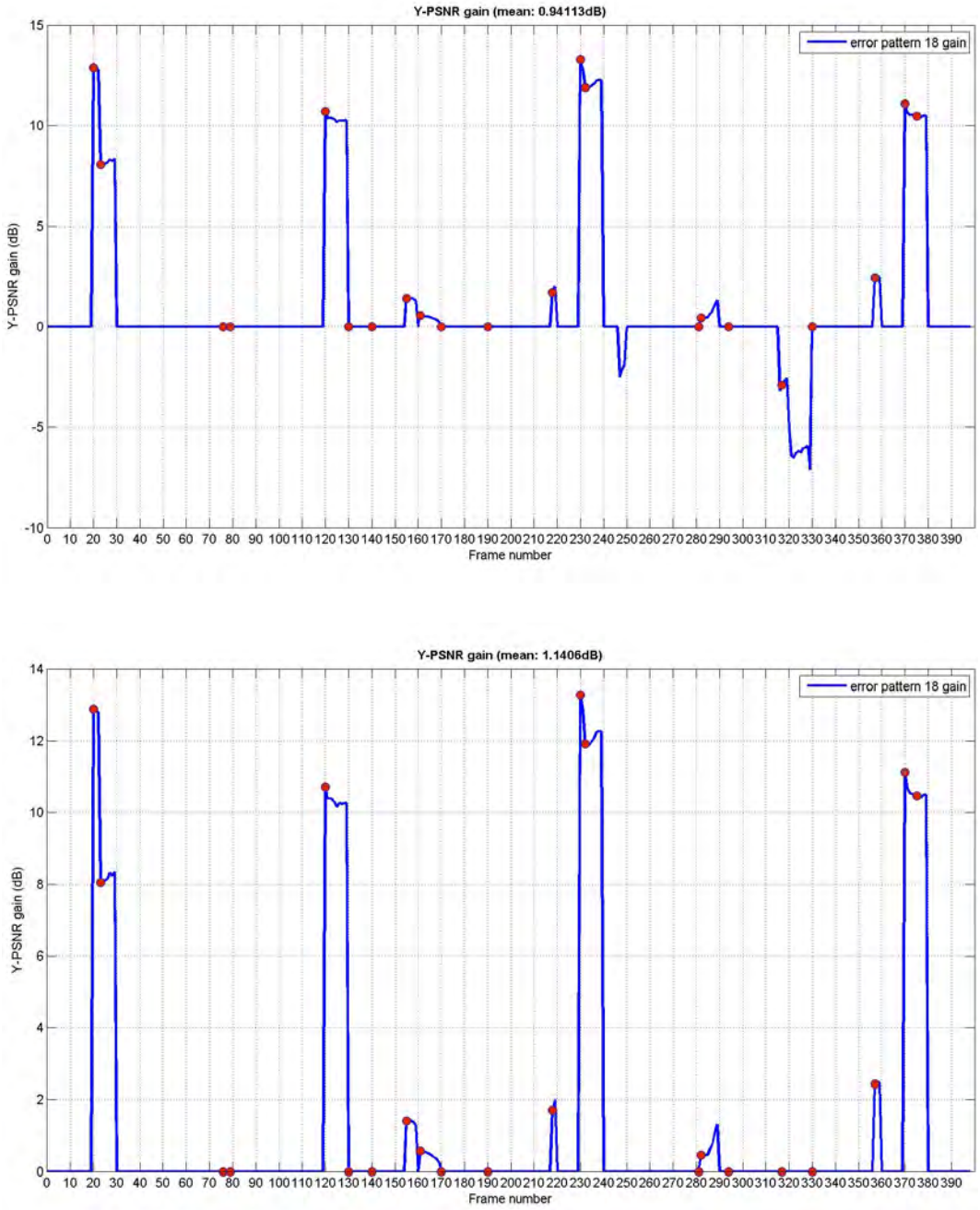


Figure B.19: Y-PSNR gain for sequence 18

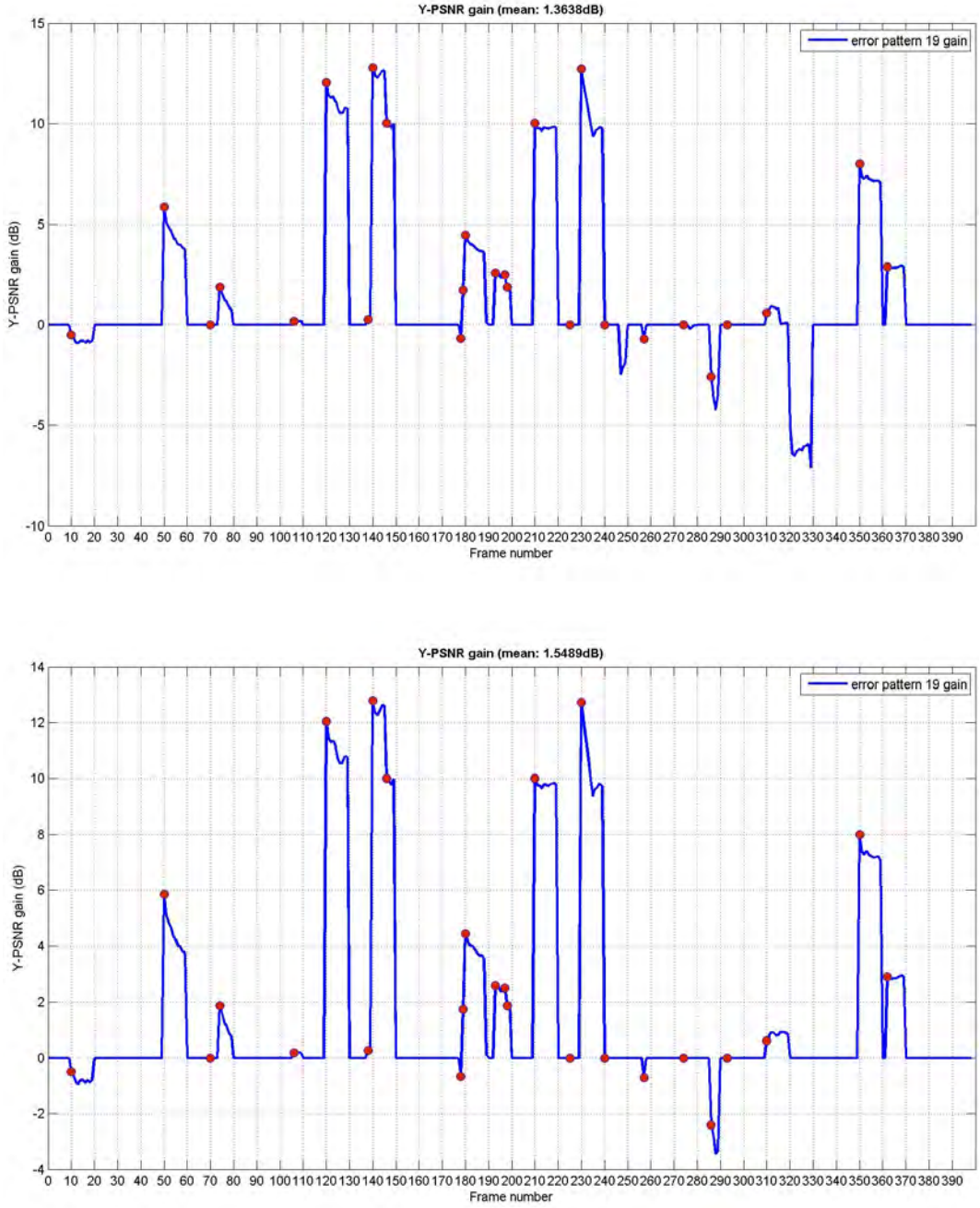


Figure B.20: Y-PSNR gain for sequence 19

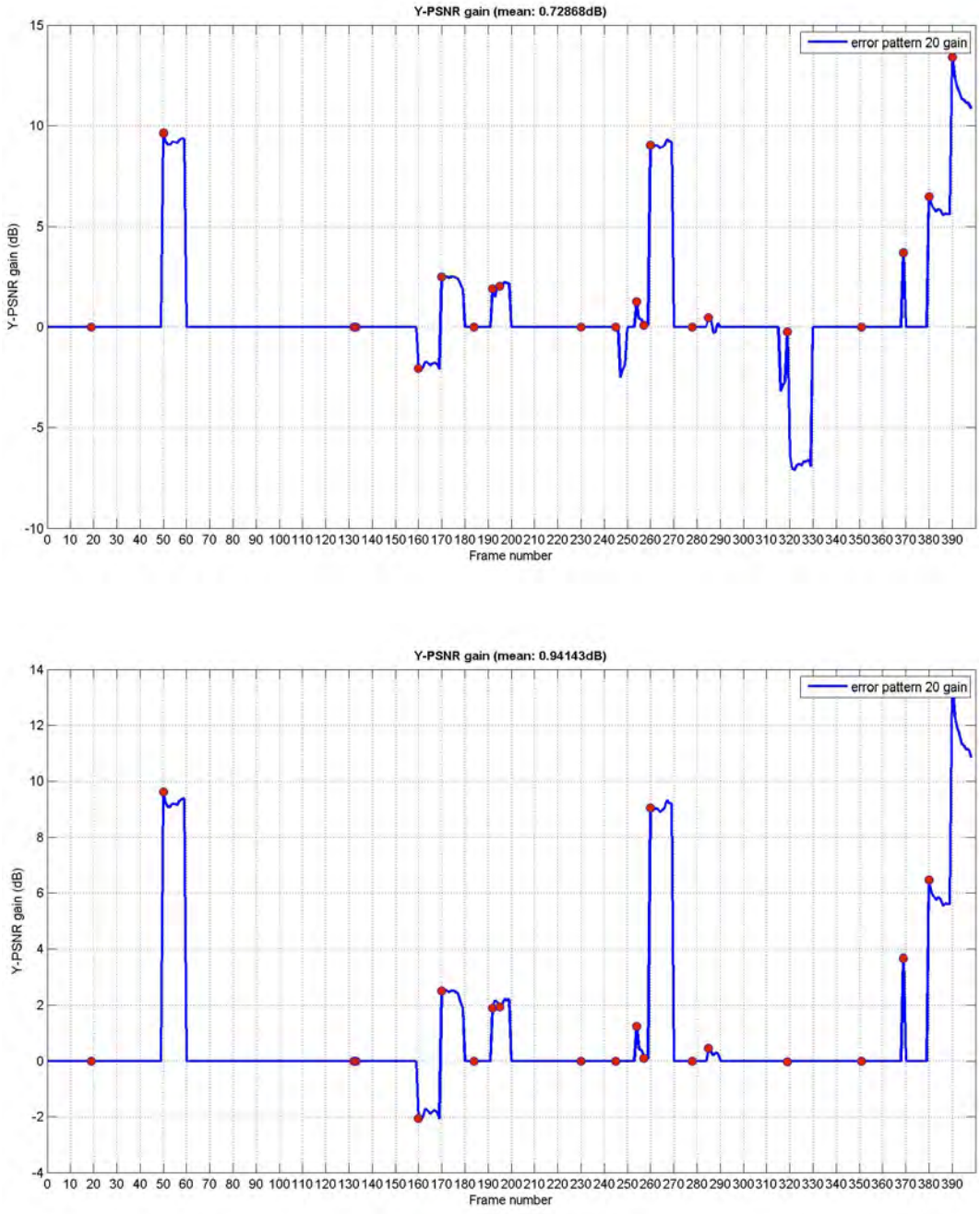


Figure B.21: Y-PSNR gain for sequence 20

Appendix C

MOS test sequences

In figures C.1 and C.2, the artifacts product of the inserted errors in the MOS test sequences are depicted. For each figure, the sequences are ordered from left to right and from top to bottom. So, the upper left figure is generated by the error pattern number 1, the next one by number 2, etc.

Figure C.1 depicts the artifact in each one of the 6 fast-moving test sequences, while Figure C.2 is for the 16 slow-moving sequences.

A description of the position and frame type for each error occurence can be found in table 7.1.

Figure C.1: Artifacts: fast-moving sequences

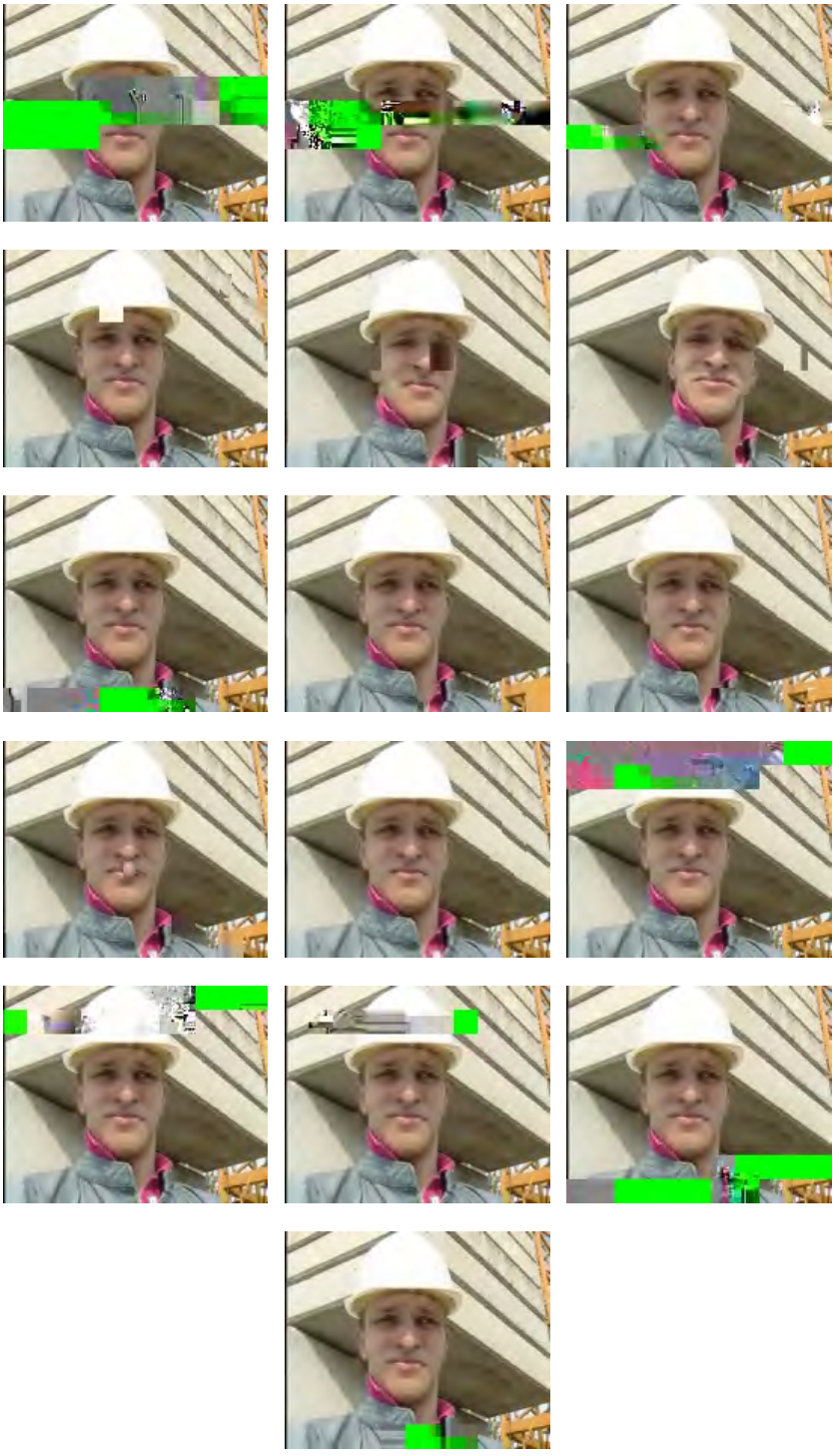


Figure C.2: Artifacts: slow-moving sequences

Appendix D

MOS tests results

Table D.1 contains the results of the MOS tests. The columns represent the following parameters:

- **Sequence ID:** identifies the sequences, as specified in section 7.1.1. Sequences 0 and 00 are error-free, 1-22 straight decoded and 23-44 the same as 1-22 but concealed.
- **Order:** the MOS tests was composed of the 46 sequences, randomly ordered. This is the display order of the sequences.
- **1-15:** the results of each individual MOS test.
- **MOS:** Mean Opinion Score. Arithmetic mean of the results for each sequence.
- δ**:** standard deviation of the MOS results for each sequence.

Sequence ID	Order	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	MOS	δ
0	24	5	4	5	5	5	5	2	5	5	5	4	5	5	5	5	4.667	0.816
00	10	5	5	5	5	3	5	4	5	5	5	5	5	3	5	5	4.667	0.724
1	18	1	1	1	1	1	1	2	1	1	1	1	1	1	1	1	1.067	0.258
2	22	1	1	1	1	1	2	1	1	1	1	1	2	1	1	1	1.133	0.352
3	35	3	2	1	2	1	4	2	3	3	2	3	3	1	2	2	2.267	0.884
4	21	4	3	2	3	2	4	2	3	4	3	3	4	3	3	3	3.067	0.704
5	30	3	2	2	3	2	3	2	3	4	3	3	2	2	3	2	2.600	0.632
6	05	3	3	3	4	3	4	3	4	4	3	4	4	4	4	4	3.600	0.507
7	17	3	2	1	1	1	2	2	2	3	2	3	3	2	2	2	2.067	0.704
8	09	5	4	4	4	5	4	3	4	4	4	4	4	4	4	4	4.067	0.458
9	46	3	3	4	4	4	5	2	4	3	4	4	4	4	4	4	3.733	0.704
10	06	2	3	3	3	3	3	2	3	4	3	4	4	3	3	3	3.067	0.594
11	14	4	4	4	4	5	5	3	4	5	4	5	5	4	4	4	4.267	0.594
12	40	2	1	1	1	1	3	3	2	2	1	2	2	1	1	1	1.600	0.737
13	15	1	2	1	1	1	2	2	2	2	1	3	2	1	2	1	1.600	0.632
14	36	3	1	2	1	1	4	2	2	2	2	2	3	1	2	2	2.000	0.845
15	25	3	3	1	2	1	4	3	2	2	2	3	3	1	2	1	2.200	0.941
16	07	3	3	3	2	2	4	2	3	4	2	4	4	3	2	2	2.867	0.834
17	16	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1	1.067	0.258
18	43	2	2	1	1	1	2	1	1	1	2	2	3	1	2	1	1.533	0.640
19	29	3	2	1	1	1	3	3	2	1	2	2	1	1	2	1	1.733	0.799
20	44	2	2	2	2	2	3	2	2	1	3	3	3	1	3	2	2.200	0.676
21	26	3	3	1	3	5	4	3	3	3	3	4	2	3	3	3	3.067	0.884
22	02	3	3	4	3	2	4	3	4	3	3	4	4	4	3	4	3.400	0.632
23	31	3	2	3	4	3	4	2	3	4	4	4	5	4	4	4	3.533	0.834
24	32	5	3	5	5	5	5	5	5	5	5	5	5	5	5	5	4.867	0.516
25	42	5	5	5	5	5	5	4	5	5	5	5	5	5	4	5	4.867	0.352
26	20	3	4	4	4	5	5	4	4	4	4	5	5	4	4	4	4.200	0.561
27	03	2	3	4	4	3	4	3	5	3	4	4	4	4	4	3	3.600	0.737
28	34	5	3	3	4	3	5	3	4	4	4	4	5	4	3	4	3.867	0.743
29	33	5	5	5	5	5	5	3	5	4	5	5	5	5	5	5	4.800	0.561
30	04	4	3	4	4	5	5	3	4	4	4	5	5	4	4	4	4.133	0.640
31	01	3	4	4	4	4	5	4	4	4	4	4	3	5	4	4	4.000	0.535
32	45	4	4	4	4	4	4	3	4	4	4	4	5	4	4	4	4.000	0.378
33	11	4	4	4	4	5	5	3	5	5	4	4	5	4	4	5	4.333	0.617
34	13	5	5	5	5	5	5	3	5	5	5	5	5	4	5	5	4.800	0.561
35	19	5	5	5	5	5	5	3	5	5	5	4	5	5	5	5	4.800	0.561
36	12	5	5	5	5	5	5	3	5	5	5	5	5	5	5	5	4.867	0.516
37	08	5	5	5	5	3	5	3	5	5	5	5	5	5	5	5	4.733	0.704
38	41	5	4	5	5	5	5	3	5	5	5	5	5	5	5	5	4.800	0.561
39	37	3	2	2	2	2	4	1	3	2	3	3	4	2	3	2	2.533	0.834
40	28	2	3	2	2	3	4	1	3	2	3	4	4	3	3	2	2.733	0.884
41	27	4	3	2	3	4	4	3	4	4	4	4	4	3	4	4	3.600	0.632
42	23	3	3	1	1	2	3	1	2	1	2	2	3	2	3	2	2.067	0.799
43	39	4	3	4	4	5	5	2	4	4	4	4	5	4	4	4	4.000	0.756
44	38	4	3	3	3	4	4	2	4	3	4	4	5	3	4	3	3.533	0.743

Table D.1: MOS test results

Appendix E

Datasets

Table E.1 contains the data used for the regression analysis for the quality estimator. This data is obtained by putting together the results of the MOS tests and the information in the `visual_impairments.txt` output when those files were decoded. The columns represent the following parameters (more extensively explained in section 5.5.6.1):

- **Sequence ID:** identifies the sequence.
- **Frame num:** the frame in which the error was introduced.
- **GOP num:** the distance between the erroneous frame and the last I frame.
- **Type:** frame type (I or P). In the error free sequences it is not specified, as no error was inserted.
- **Artifact size:** the size of the artifact in MBs as detected by the detection algorithm. 0 means that no artifact was detected. For straight decoded sequences it's the size of the detected artifact, while for concealed sequences the number of concealed macroblocks.
- **MOS:** Mean Opinion Score of the sequence.
- **Seq Length and Vote (only for I frames):** how long was the sequence and how big its vote in the voting system that detects artifacts in I frames (explained in section 8.2).
- **Difference frame:** value of the `diff_frame` variable.
- **Concealment:** what type of concealment was used for that particular sequence:
  - **SD:** Straight Decoding. Only detection was performed.
  - **VISCONC:** detection + concealment of errors.
  - **Error free:** reference error-free sequences.
- **Range:** the range of frames from the original foreman sequence that were used in that particular sequence. 0-74 (slow-moving) or 260-334 (fast-moving). See section 7.1.1.
- **Sequence:** identifies which error pattern has been used. Each combination of range and error pattern is different, so error_pattern_1 for the range 0-74 is different from error_pattern_1 for the range 260-334. The SD and VISCONC sequences for a given combination of range and sequence use the same error pattern.

Sequence ID	Frame num	GOP num	Type	Artifact size	MOS	Seq length	Vote	Difference frame	Concealment	Range	Sequence
0	0	0		0	4.67	0	0	0	error-free	0-74	error_free
00	0	0		0	4.67	0	0	0	error-free	260-334	error_free
1	25	0	I	21	1.07	4	3.75	1.56	SD	0-74	error_pattern_1
2	25	0	I	15	1.13	8	3.71	1.56	SD	0-74	error_pattern_2
3	25	0	I	5	2.27	4	3.75	1.56	SD	0-74	error_pattern_3
4	26	1	P	2	3.07	0	0	1.93	SD	0-74	error_pattern_4
5	34	9	P	3	2.6	0	0	2.92	SD	0-74	error_pattern_5
6	43	18	P	1	3.6	0	0	2.35	SD	0-74	error_pattern_6
7	25	0	I	8	2.07	20	3.75	1.56	SD	0-74	error_pattern_7
8	26	1	P	0	4.07	0	0	0	SD	0-74	error_pattern_8
9	27	2	P	1	3.73	0	0	2.62	SD	0-74	error_pattern_9
10	27	2	P	1	3.07	0	0	2.62	SD	0-74	error_pattern_10
11	27	2	P	0	4.27	0	0	0	SD	0-74	error_pattern_11
12	25	0	I	17	1.6	32	4.23	1.56	SD	0-74	error_pattern_12
13	25	0	I	12	1.6	4	3.75	1.56	SD	0-74	error_pattern_13
14	25	0	I	7	2	7	6.75	1.56	SD	0-74	error_pattern_14
15	25	0	I	13	2.2	8	5.12	1.56	SD	0-74	error_pattern_15
16	25	0	I	4	2.87	4	3.75	1.56	SD	0-74	error_pattern_16
17	25	0	I	27	1.07	4	3.75	13.01	SD	260-334	error_pattern_1
18	25	0	I	22	1.53	4	3.75	13.01	SD	260-334	error_pattern_2
19	25	0	I	9	1.73	4	3.75	13.01	SD	260-334	error_pattern_3
20	26	1	P	5	2.2	0	0	13.52	SD	260-334	error_pattern_4
21	34	9	P	1	3.07	0	0	10.58	SD	260-334	error_pattern_5
22	42	17	P	0	3.4	0	0	0	SD	260-334	error_pattern_6
23	25	0	I	22	3.53	0	0	1.56	VISCONC	0-74	error_pattern_1
24	25	0	I	15	4.87	0	0	1.56	VISCONC	0-74	error_pattern_2
25	25	0	I	5	4.87	4	3.75	1.56	VISCONC	0-74	error_pattern_3
26	26	1	P	2	4.2	0	0	1.93	VISCONC	0-74	error_pattern_4
27	34	9	P	3	3.6	0	0	2.92	VISCONC	0-74	error_pattern_5
28	43	18	P	1	3.87	0	0	2.35	VISCONC	0-74	error_pattern_6
29	25	0	I	8	4.8	4	1.55	1.56	VISCONC	0-74	error_pattern_7
30	26	1	P	0	4.13	0	0	0	VISCONC	0-74	error_pattern_8
31	27	2	P	1	4	0	0	2.62	VISCONC	0-74	error_pattern_9
32	27	2	P	1	4	0	0	2.62	VISCONC	0-74	error_pattern_10
33	27	2	P	0	4.33	0	0	0	VISCONC	0-74	error_pattern_11
34	25	0	I	19	4.8	0	0	1.56	VISCONC	0-74	error_pattern_12
35	25	0	I	13	4.8	0	0	1.56	VISCONC	0-74	error_pattern_13
36	25	0	I	7	4.87	7	6.75	1.56	VISCONC	0-74	error_pattern_14
37	25	0	I	13	4.73	0	0	1.56	VISCONC	0-74	error_pattern_15
38	25	0	I	4	4.8	4	3.75	1.56	VISCONC	0-74	error_pattern_16
39	25	0	I	27	2.53	0	0	13.01	VISCONC	260-334	error_pattern_1
40	25	0	I	26	2.73	6	4.02	13.01	VISCONC	260-334	error_pattern_2
41	25	0	I	14	3.6	16	7.02	13.01	VISCONC	260-334	error_pattern_3
42	26	1	P	5	2.07	0	0	13.52	VISCONC	260-334	error_pattern_4
43	34	9	P	1	4	0	0	10.58	VISCONC	260-334	error_pattern_5
44	42	17	P	0	3.53	0	0	0	VISCONC	260-334	error_pattern_6

Table E.1: Data used for the regression analysis and MOS tests

Appendix F

MOS estimation results

Table F.1 contains the data used to calculate the MOS estimation and the correlation coefficients. The regression formula used is stated in section 8.2.3 (8.1).

Seq ID	Type	Size	GOP frame	Diff frame	MOS	Estimation	Error
1	I	21	0	1.56	1.07	1.26	0.19
2	I	15	0	1.56	1.13	1.59	0.46
3	I	5	0	1.56	2.27	2.36	0.09
4	P	2	1	1.93	3.07	3.07	0.00
5	P	3	9	2.92	2.6	2.80	0.20
6	P	1	18	2.35	3.6	3.48	-0.12
7	I	8	0	1.56	2.07	2.10	0.03
8	P	0	1	0	4.07	3.80	-0.27
9	P	1	2	2.62	3.73	3.37	-0.36
10	P	1	2	2.62	3.07	3.37	0.30
11	P	0	2	0	4.27	3.80	-0.47
12	I	17	0	1.56	1.6	1.47	-0.13
13	I	12	0	1.56	1.6	1.79	0.19
14	I	7	0	1.56	2	2.18	0.18
15	I	13	0	1.56	2.2	1.73	-0.47
16	I	4	0	1.56	2.87	2.46	-0.41
17	I	27	0	13.01	1.07	0.98	-0.09
18	I	22	0	13.01	1.53	1.19	-0.34
19	I	9	0	13.01	1.73	1.98	0.25
20	P	5	1	13.52	2.2	2.06	-0.14
21	P	1	9	10.58	3.07	3.18	0.11
22	P	0	17	0	3.4	3.91	0.51

Table F.1: MOS estimation results

Appendix G

MOS gain estimation results

Table G.1 contains the data used to calculate the MOS gain estimation and the correlation coefficients. The regression formula used is stated in section 8.3.1 (8.6).

Sequence ID	Type	Artifact size	Difference frame	Gain	Estimation	Error
23	I	22	1.56	2.47	2.88	0.41
24	I	15	1.56	3.73	3.13	-0.60
25	I	5	1.56	2.6	2.39	-0.21
26	P	2	1.93	1.13	0.94	-0.19
27	P	3	2.92	1	0.90	-0.10
28	P	1	2.35	0.27	0.68	0.41
29	I	8	1.56	2.73	2.75	0.02
30	P	0	0	0.07	0.02	-0.05
31	P	1	2.62	0.27	0.68	0.41
32	P	1	2.62	0.93	0.68	-0.25
33	P	0	0	0.07	0.02	-0.05
34	I	19	1.56	3.2	3.06	-0.14
35	I	13	1.56	3.2	3.09	-0.11
36	I	7	1.56	2.87	2.64	-0.23
37	I	13	1.56	2.53	3.09	0.56
38	I	4	1.56	1.93	2.24	0.31
39	I	27	13.01	1.47	1.20	-0.27
40	I	26	13.01	1.2	1.34	0.14
41	I	14	13.01	1.87	2.00	0.13
42	P	5	13.52	-0.13	-0.08	0.05
43	P	1	10.58	0.93	0.80	-0.13
44	P	0	0	0.13	0.02	-0.11

Table G.1: MOS gain estimation results

List of Symbols and Abbreviations

Abbreviation	Description
AVI	Audio Video Interleave
CSV	Comma Separatd Values
DPB	Decoded Picture Buffer
DCT	Discrete cosine transform
GOP	Group Of Pictures
H.264/AVC	H.264/Advanced Video Coding
HD	High Definition
IP	Internet Protocol
MAM	Macroblock Allocation Map
MB	MacroBlock
MBLC	Macroblock Level Concealment
MOS	Mean Opinion Score
NAL	Network Abstraction Layer
NALU	Network Abstraciton Layer Unit
PPS	Picture Paremeter Set
QCIF	Quarter Common Intermediate Format
QP	Quantization Parameter
RTP	Real-time Transport Protocol
SC	Syntax Check
SDP	Session Description Protocol
SH	Slice Header
SLC	Slice Level Concealment
SPS	Sequence Parameter Set
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VCL	Video Coding Layer
VIDC	Visual Impairments Detection and Concealment
Y-PSNR	Luminance Peak Signal-to-Noise Ratio

List of Figures

List of Tables