ICSP Video Codec Enhancement

Students:
Alpaslan Kubilay
Beltrami Alberto
Petrov Boris
Starygin Evguéniy

Professor:
Lafruit Gauthier

June 05, 2025

Contents

1. Introduction

The aim of the project was to put the students in the shoes of MPEG consortium members. The primary task was to achieve compression improvements over the basic version of MPEG-1[1], but it also includes analysis of the results and regular meetings held in roughly the same vein as the real ones.

The original MPEG-1 video codec has been implemented based on fundamental video compression principles. The codec includes several core components commonly found in standard compression systems. Some characteristics of the codec are:

- the codec operates on 16x16 macro-blocks, each of which is subdivided into four 8x8 blocks to apply the compression;
- both intra and inter prediction, including all intra prediction modes;
- Discrete Cosine Transform (DCT) application followed by quantization to reduce spatial redundancy;
- entropy coding techniques implementation to improve compression efficiency;
- only the YUV420 color format and is limited to CIF resolution (352x288);
- multi-threaded encoding capabilities to improve processing speed. However, this feature has not been studied and/or applied at all during this work.

1.1. Video Format support

To pursue scalable coding height and width parameter have been restored so, we could encode QCIF and 4CIF videos easily with the codec.

1.2. Entropy coder selection

For ease of data, generation a `-e` parameter has been added to select which entropy coding to use during the codec run.

1.3. Data

All data generated to compare the original entropy coder to a another one, has been generated using the following Qp's: 2, 7, 20 and 42.

1.4. Development process

The course of development took place over two months and a half during which several meetings were held for status updates and planning. Outside of these official meetings, several ad-hoc ones happened as well. Each participant was generally responsible for a specific task and worked independently from the others. Before each meeting, the *software coordinator* (Starygin Evguéniy) took on the responsibility of merging all work into one code-base, which was hosted on GitHub[2].

[1] https://github.com/JawThrow/ICSPCodec

[2] https://github.com/evgueniy/INFO-H516_Project

2. PSNR

To evaluate the compression quality of the original ICSP codec, the *Peak Signal-to-Noise Ratio (PSNR)* between the original video frames and the corresponding reconstructed frames produced by the codec were measured.

$$\mathrm{PSNR} = 10 \log_{10} \frac{255^2}{\mathrm{MSE}}, \qquad \text{where} \qquad \mathrm{MSE} = \frac{\sum_{\text{all pix}} \left(p_{\text{ori}} - p_{\text{dec}}\right)^2}{N} \tag{1}$$

This involved modifying the codec to output both the reconstructed frames and relevant encoding statistics during the compression process. The PSNR was computed frame by frame using the luminance (Y) channel of the YUV420 format. Alongside PSNR values, the size of the encoded bit-stream was extracted for each frame to calculate the corresponding bit-rate over time. With this data, two sets of graphs were generated: PSNR versus frames and PSNR versus bit-rate. These visualizations (over which we over in the next section) enabled a detailed assessment of the codec's efficiency and quality retention. In particular, different features[3] of the codec were studied and compared each other through PSNR graphs:

- *all intra*: all frames are considered as *I-frames*;
- *intraperiod 8*: only one I-frame each 8 frames (the other are *P-frames*);
- *intraperiod 16*: only one I-frame each 16 frames;
- *intraperiod 32*: only one I-frame each 32 frames;
- *1 intra*: only one I-frame and every other frame is a P-frame.

All these features should theoretically involve increasing "quality" in the compression result. In some cases, graphs have underlined a decreasing quality as intra-period increased. This required further in-depth analysis about entropy coding and motion estimation of the original codec (see Section 4 and Section 5 respectively). However, PSNR graphs highlighted acceptable results achieved by the original video encoder, with admissible PSNR values per bit-rate.

[3]Notice that no feature uses *B-frames* within the original codec. They were not implemented during the project neither to not enlarge the workload and the complexity of the codec.

3. Statistics

The first step before delving into optimizations was to setup an extensive and easy-to-use pipeline to collect statistics about the results of the codec. This consisted of first identifying which data was relevant, then where it was found in the code and collecting (or computing) it, and finally displaying it in an insightful manner (e.g. tables, or preferably plots).

With this information, we could identify which parts are the bottlenecks and where our efforts would be most warranted. It would also allow us to better gauge the effectiveness of our changes and which aspect they affected most, if at all. For example, one entropy coding strategy could lower the total size of the AC components, but increase that of the DC components.

On an implementation note, we decided to collect all information in CSV files and then process and plot the results using Python (and its `pandas` and `matplotlib` libraries). These scripts can be found in `./Scripts`, starting from the root of the project. Each has a slightly different way to be used, but everything is documented in the `README.md` or in the script itself.

3.1. Per frame statistics

The first visualization we implemented shows the bit-size of the AC components, the DC components, the motion vectors, and the total entropy for each frame. Additionally, we plot for each frame the PSNR. With this we can evaluate which components take up the most space and what characteristics of the videos cause bit-rate spikes or low PSNR. We show an example in Figure 1. It is programmed in `plot_normal.py`, but one can also use `plot_psnr_per_frame.py` to individually show the PSNR per frame.

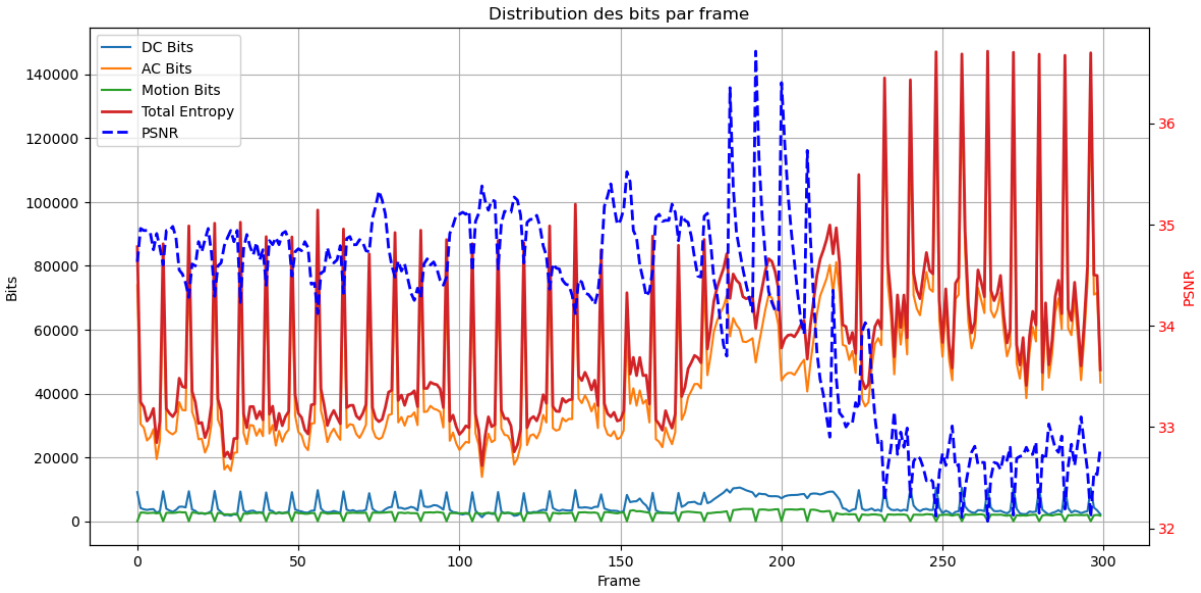

Figure 1: Bits distribution for the `foreman` test video with the final codec (using the CABAC entropy coder and with the motion vector bug mentioned in Section 5 fixed) with intra period of 8 and quantization factor of 16

We should note however one thing about the data shown in Figure 1. There is an irregularity with the PSNR shown with respect to the theoretical expectations. Usually, the PSNR should be highest at the I frames (the peaks of the bit-rate) and then decrease with each following P

frame up until the next I frame. What we have instead is a PSNR dipping at the bit-rate peaks and then increasing. This is counter-intuitive and we have no plausible explanation for it. We have double-checked the formulas and the plotting scripts, and everything seems in order. We suspect that the problem may come from the code that reconstructs the frames, which was pre-existing to our changes.

Regarding motion estimation and motion compensation, we have also implemented a way to see the motion vectors and the error image for each frame. We show an example in Figure 2.

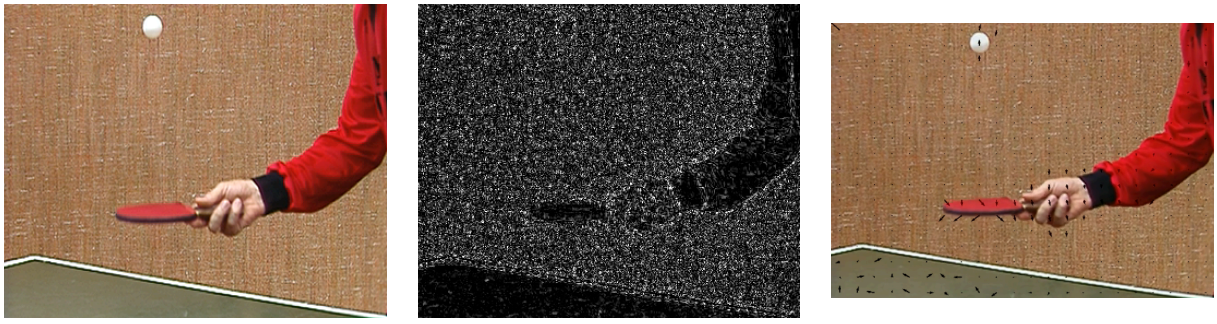

Figure 2: Original frame (left), error image (middle), motion vectors (right)

The error images are computed directly in the encoder, just after the compression, in the function `checkResultFrames` which reconstructs a YUV video from the compressed one. We use this reconstructed video in turn to compute the error images which are then written as a YUV video and can be directly viewed in YUView. We have only recorded the differences in the Y values, and set U and V to 128 in order to have a black-and-white error image. The values in the images we show here are scaled up because the original errors are too low to see anything.

As for the motion vectors, these are first collected in `makebitstream`, along with the block to which they relate, and written to a CSV file under a name like `mv_[video_name]_.csv`. This is then used to create frames displaying the motion vectors with `matplotlib`, which are finally compiled into a video and overlaid on top of the original video using `ffmpeg`. This pipeline is contained in the script `motion_vectors.py`.

3.2. Complete video statistics

Regarding complete video statistics, these are used to see how different arguments and encoders compare to one another. As such, they show how the PSNR evolves and the BD-rate. We have two scripts for this task.

The first is meant to compare different intra-frame periods and how efficiently the temporal redundancy is harnessed. As we want to see how for each period the compression evolves for different bit-rates, we pass to the script several quantization parameters. It roughly amounts in comparing the evolution of the PSNR to the bit-rate, but this is a byproduct of changing the QP. We have no method to accurately control the bit-rate (which is to be expected from this "primitive" encoder, as it's a task difficult even for the most advanced ones).

Implementation-wise it is similar to that of showing the PSNR per frame, except that we compute only one PSNR and bit-rate for the whole video. This is done for several sets of

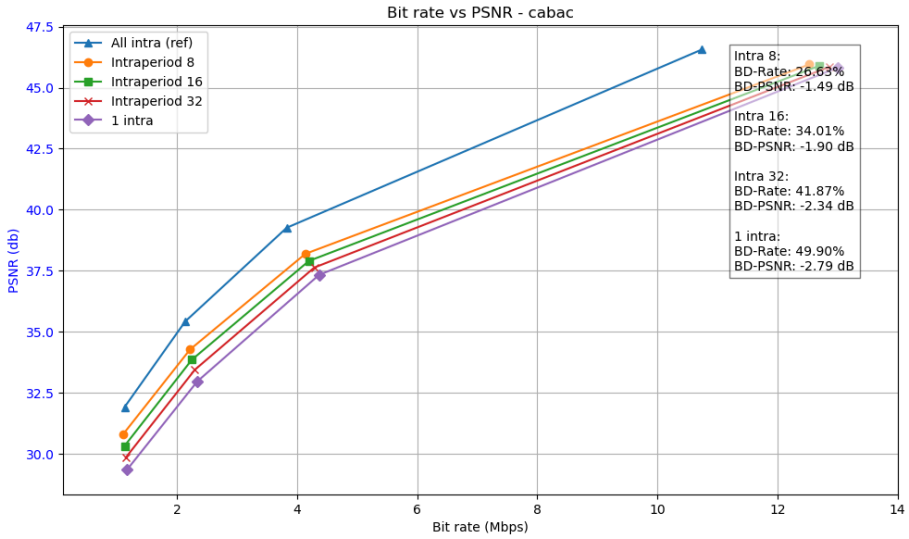

Figure 3: Evolution of the PSNR with increasing bit-rate for different intra-frame periods (latest encoder, using CABAC as the entropy encoder, with the `football` video and QP values of 2, 8, 16, 32)

parameters on one source video and then the results are plotted together as in Figure 3. It is implemented in `plot_psnr.py`.

The second one is an extension of the first one with the addition that now it is done for several entropy coders. It is implemented in `plot_psnr_ec.py` and examples of it can be seen in Section 6.1 and Section 6.2.

4. Entropy coding assessment

Some of the PSNR vs. bit-rate graphs revealed a decrease in compression quality after new features were added to the codec. This outcome was initially counter-intuitive, since it was expected that enhancements would improve compression efficiency. However, upon deeper analysis, this unexpected behavior appeared to be linked to the entropy coding stage. The original entropy coder implemented in the ICSP codec looked like a basic look-up table rather than a true entropy-based method. To investigate this further, histograms representing the frequency distribution of the encoded bit-stream were generated. Results highlighted by these new graphs were again quite unexpected. The distribution of the encoded bit-stream appeared well-structured and aligned with the characteristics typically associated with an efficient encoder. Specifically, the most frequently occurring symbols were encoded using fewer bits, resulting in a frequency concentration around lower bit lengths. This suggested that, at a surface level, the bit allocation strategy followed the principles of entropy coding, where more probable values are assigned shorter codes. However, despite this well-shaped bits distribution, the overall compression performance remained in some cases suboptimal, indicating that the issue might be in the motion estimation methods and/or motion vector generation steps.

4.1. ICSPCodec Bitstream

The original bit-stream is composed by a header of 14 bytes:

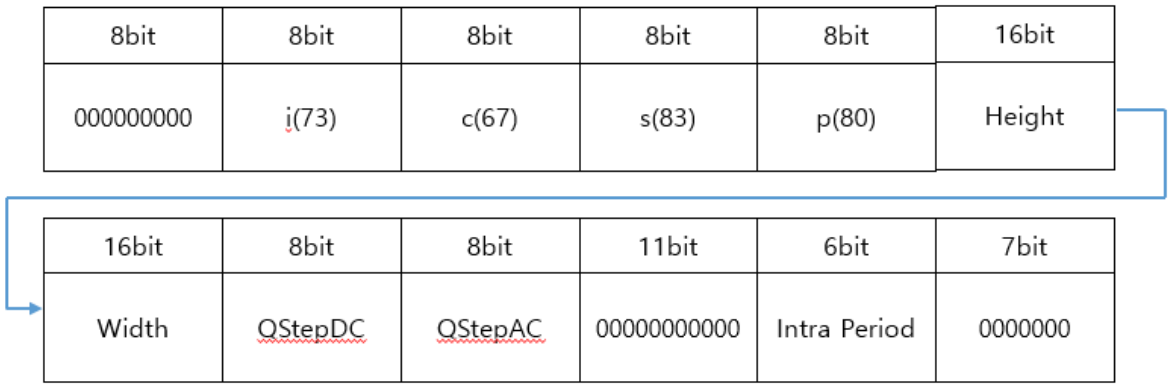

Figure 4: 14 bytes header

followed by intra body if I-frame:

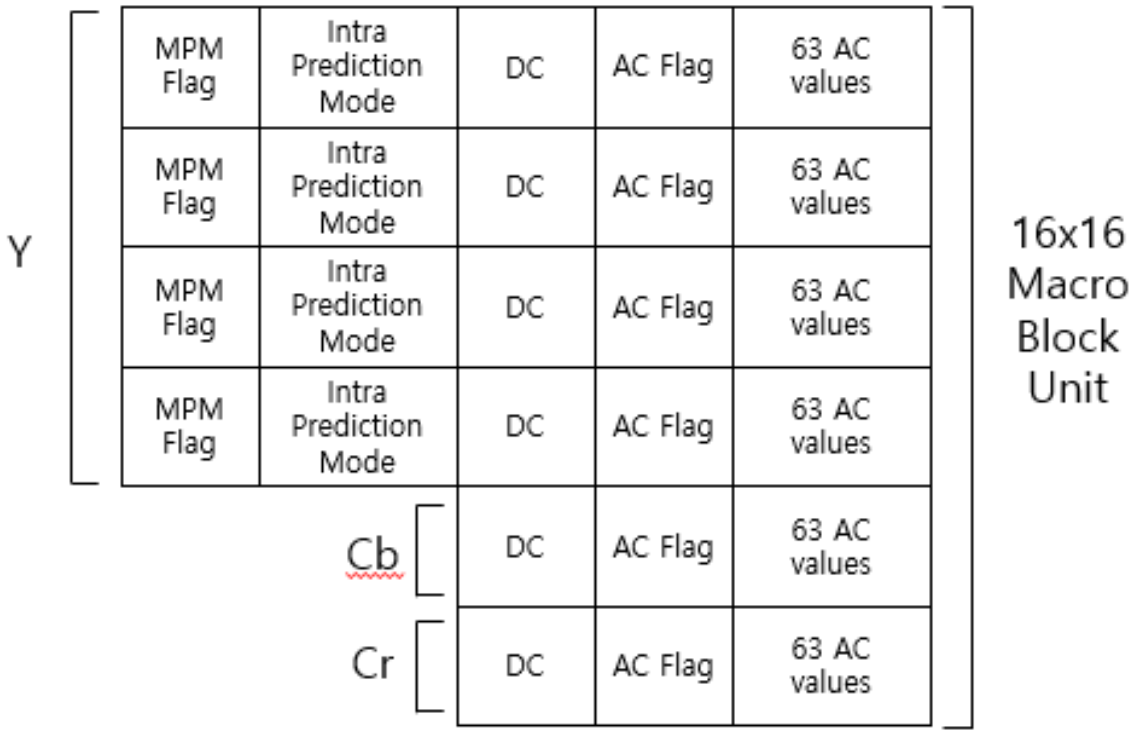

Figure 5: Intrabody

or interbody if P-frame:

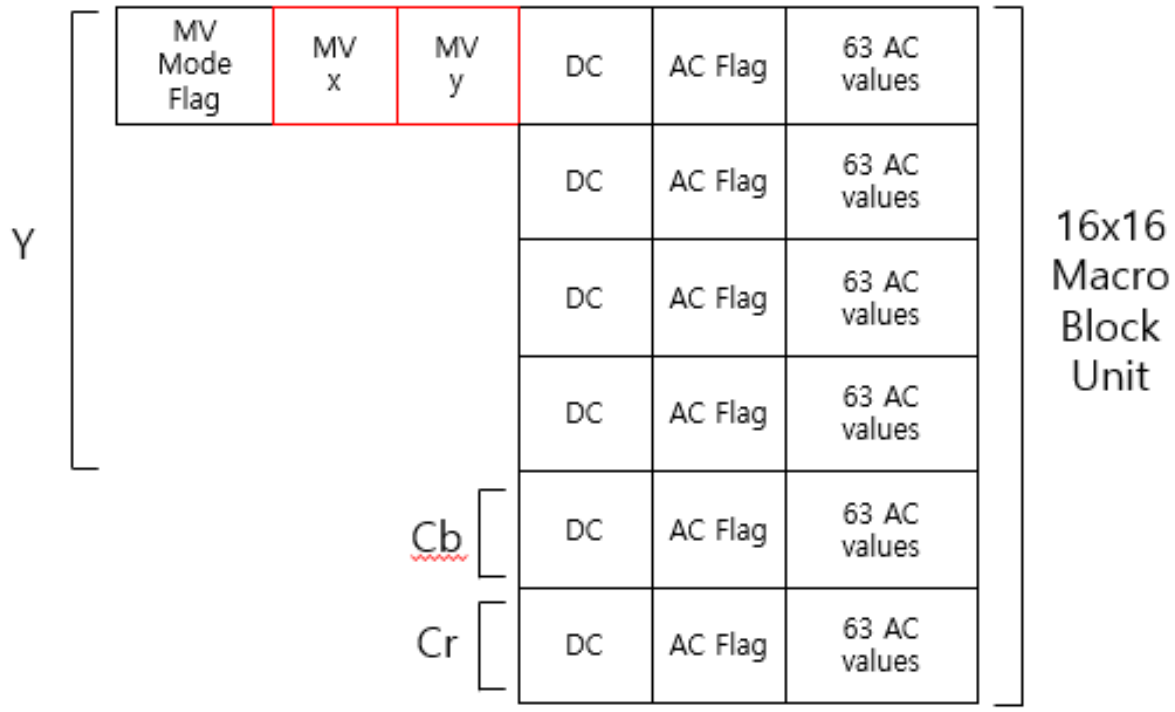

Figure 6: Interbody

5. Motion estimation and motion vectors

5.1. Visualization

To further investigate the suboptimal compression results, particular attention was given to the motion estimation process and the handling of motion vectors within the original ICSP codec. Since motion estimation plays a crucial role in inter-frame compression by reducing temporal redundancy, any inefficiencies in this component can significantly impact both compression efficiency and output quality. Upon analysis, it was suspected that the motion estimation algorithm used by the codec was potentially inaccurate, leading to poorly chosen motion vectors. These inaccurate vectors may have resulted in ineffective inter-frame prediction, causing higher residual errors and reducing the PSNR of the reconstructed frames. This prompted a deeper evaluation of how motion vectors were calculated, encoded, and utilized during the compression process, as they appeared to be a key factor contributing to the lower-than-expected performance despite the presence of typical compression features.

Motion vectors were extracted and overlaid on the base video frames for visual inspection. The resulting visualization did not align with the actual motion observed in the video. Specifically, motion vectors pointed diagonally, regardless of the true direction of movement. This anomaly prompted a debugging phase to identify the root cause.

5.2. Debug

Upon closer inspection, a bug in the spiral search algorithm used for motion estimation was discovered: it was incorrectly sampling only the diagonal pixels of each candidate block, rather than evaluating all positions along the intended spiral path (as in Figure 7). After correcting the algorithm to properly test all pixels along the spiral, the visualizations improved (motion vectors were no longer constrained to diagonal directions, and a more diverse set of motion directions became visible). Despite this improvement, the results were still not fully satisfactory.

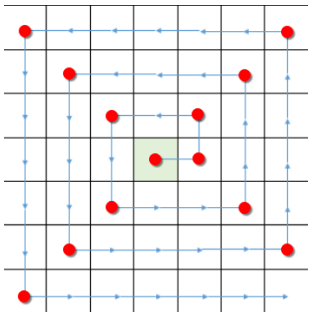

Figure 7: Example spiral search, only positions marked with red dots were tried

Further experiments with larger search windows (longer spirals) produced mixed results. It is likely that the limited resolution of the test videos (CIF format) and inherent visual noise contributed to the inconsistency. We have also found that during the motion estimation phase, the reference image used is not a reconstruction of the frame in question but the original frame. While this certainly could have an impact on the motion vectors, we cannot ascertain that this is the root cause of the inaccurate motion vectors. Fixing this issue is possible but non-trivial and it was discovered too late in the development process to remediate to it.

6. Entropy coders

6.1. Huffman coding

Huffman Coding is a technique of compressing data to reduce its size without losing any of the details and it is usually useful when data to be compressed contain frequently occurring values. To improve the base ICSP Codec, Huffman coding was implemented into the encoder.

The Huffman implementation focused on the AC coefficients of each macro-block, where the absolute values of the coefficients were first analyzed to compute symbol frequencies. Based on this distribution, a Huffman tree was dynamically constructed, and variable-length codes were assigned. During encoding, each AC coefficient was converted to its corresponding Huffman code, with an additional bit added to represent the sign of non-zero values. The resulting bit-stream was then packed into bytes. This integration aimed to improve compression efficiency by better matching the coding strategy to the actual data distribution within the blocks.

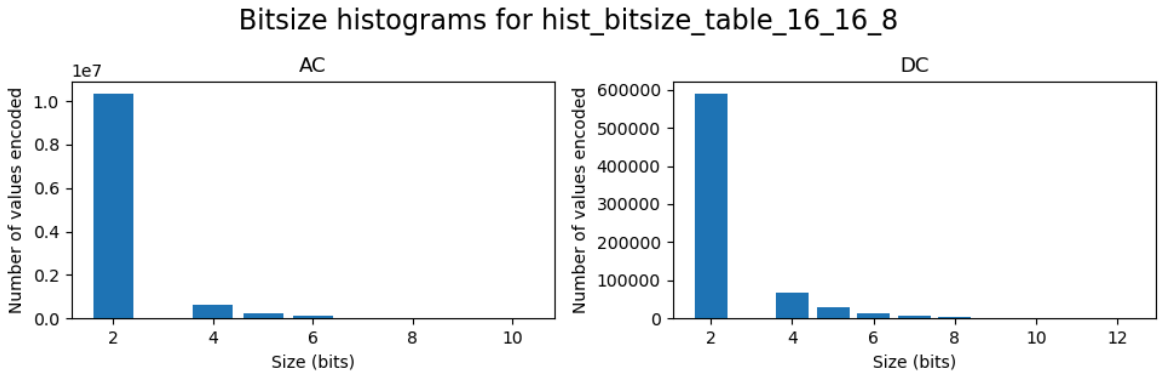

Figure 8: Original Decoder Histogram

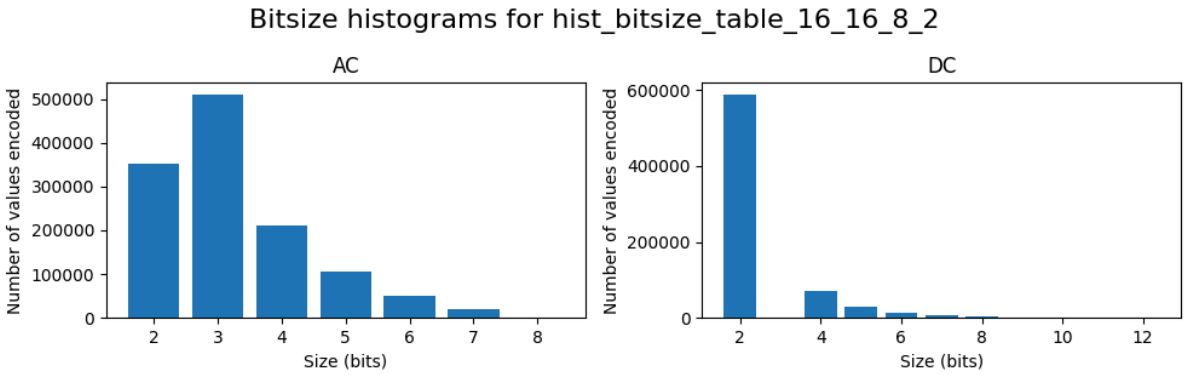

Figure 9: Huffman Decoder Histogram

As expected, the DC histograms remain identical, since no modifications were applied to the DC coefficients. In contrast, the AC histogram after Huffman coding exhibits a more balanced distribution across symbol values. This redistribution suggests a more efficient entropy representation, which is expected to reduce the overall bit-stream size.

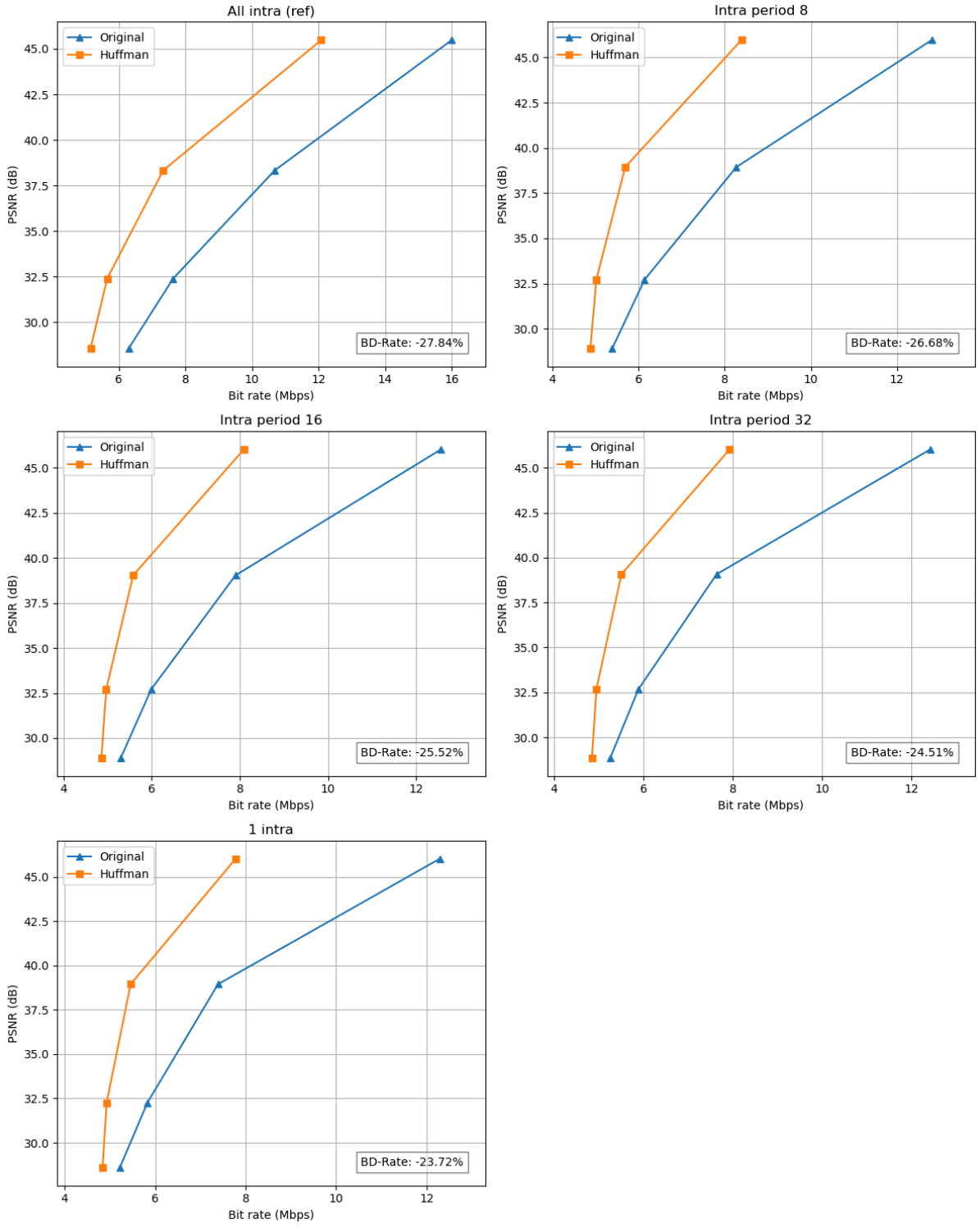

Figure 10: BD-Rate between Huffman and Original encoder for different intra-frame periods.

As anticipated, applying Huffman coding exclusively to the AC coefficients already yields a noticeable improvement in reconstruction quality, with a PSNR gain corresponding to a BD-Rate reduction of approximately 25%.

6.2. CABAC: Context Adaptive Binary

Context Adaptative Binary Coder is a variable length, lossless entropy coding method used to compress sequences of bits by encoding the most probable symbol with less bits depending on the context bins.

To improve the ICSPCodec, we've used VLC's x264 cabac encoder and modified the original `makebitstream` function which generate the header (Figure 4) then the body of the bit-stream. An alternative intra-body (Figure 5) / inter-body (Figure 6) and all intra-body (if there are only I-frames) function is called instead of the codec base one where flags and values are encoded with the cabac encoder, each one using a different context bin. Each value is encoded first, with a flag that signals if it is equal to zero then if not we encode the unsigned number and its sign.

6.3. Cabac results

6.3.1. table_cif(352x288)_300f.yuv

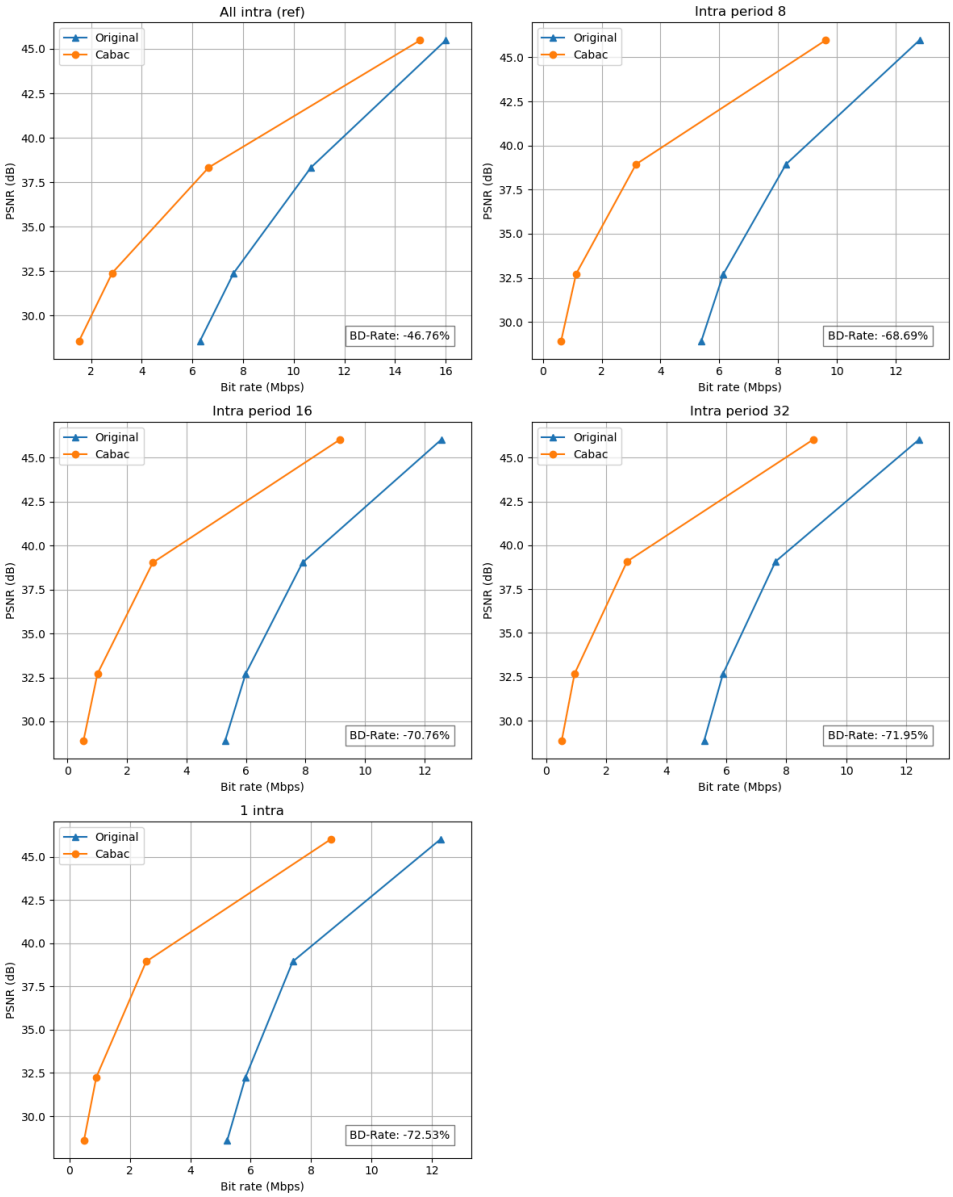

Figure 11: Table CIF Cabac vs original Decoder Histogram

6.3.2. formean_cif(352x288)_300f.yuv

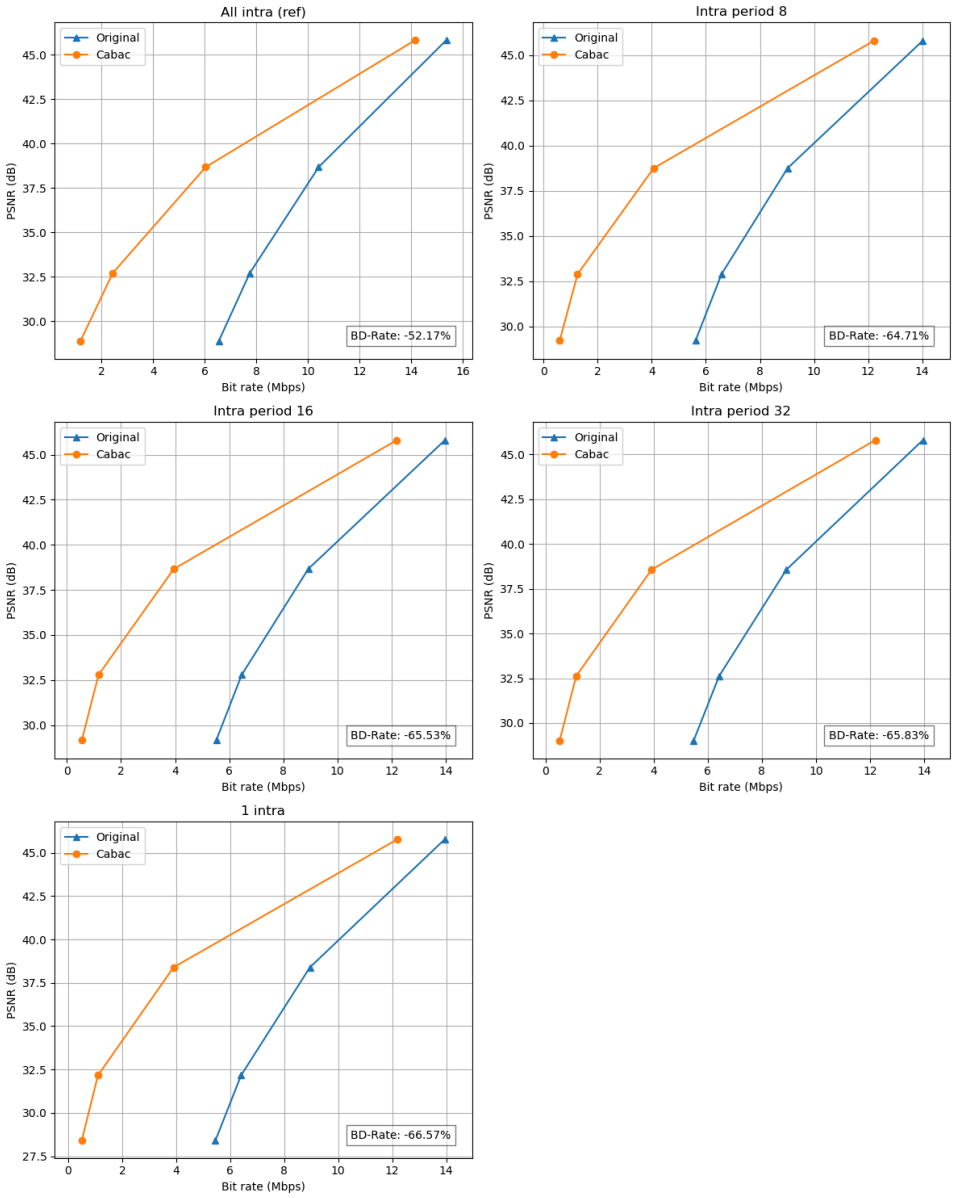

Figure 12: Foreman CIF Cabac vs original Decoder Histogram

We can see in both sequences that we get a negative BD-rate, meaning that our new way of encoding is more efficient as we get a lower bit-rate for similar PSNR. We can also observe the consequence of encoding the zero flag of each value as we, increase in QP we have a bigger gap between curves as we don't encode the whole value and its sign and the flags are encoded via Cabac.

7. Scalable coding

When a server streams a video to a client, he may propose several resolutions the client can choose from. This is usually to be able to accommodate the many different internet speeds. This system may be implemented in may ways, for example by having a different video for each stream, but an efficient solution is that of *scalable coding*.

Its basic idea is that instead of having multiple videos that greatly overlap over each other (it's the same content after all, just at different resolutions), we have one on the lowest resolution, the *base*, and several *enhancement layers* that can be utilized to improve it.

The scalable coding pipeline implemented in this work separates the video into two layers: a base layer at QCIF/CIF resolution and an enhancement layer carrying residual information. The original CIF/4CIF video is first downscaled using FFmpeg and then encoded with our encoder. This encoded base layer is then decoded and upscaled back. The enhancement layer is constructed by computing the residual between the original frame and the upscaled decoded version. Since residuals can contain negative values, they are shifted into the $[0, 255]$ range and saved in YUV420p format with neutral chroma components. The residuals are then encoded using the same encoder. At the decoder side, the base layer is upscaled, the enhancement residuals are decoded and unshifted, and the final reconstruction is obtained by adding the residuals back to the upscaled base. PSNR is computed between the original and both the upscaled and reconstructed frames to evaluate quality gain from the enhancement layer. The PSNR between the base layer "input" and "output" is computed too.

For the results, we show first the evolution of the PSNR over time and then a table with the file sizes of the different results. Ideally, we would want the downscaled video plus the enhancement layer to be lower than the downscaled video and the original video (otherwise we could just send the two videos). We would also want the quality of the video after the enhancement procedure to be good enough compared to the original. We show the results for `table` in Figure 13 and Table 1, and for `foreman` in Figure 14 and Table 1.

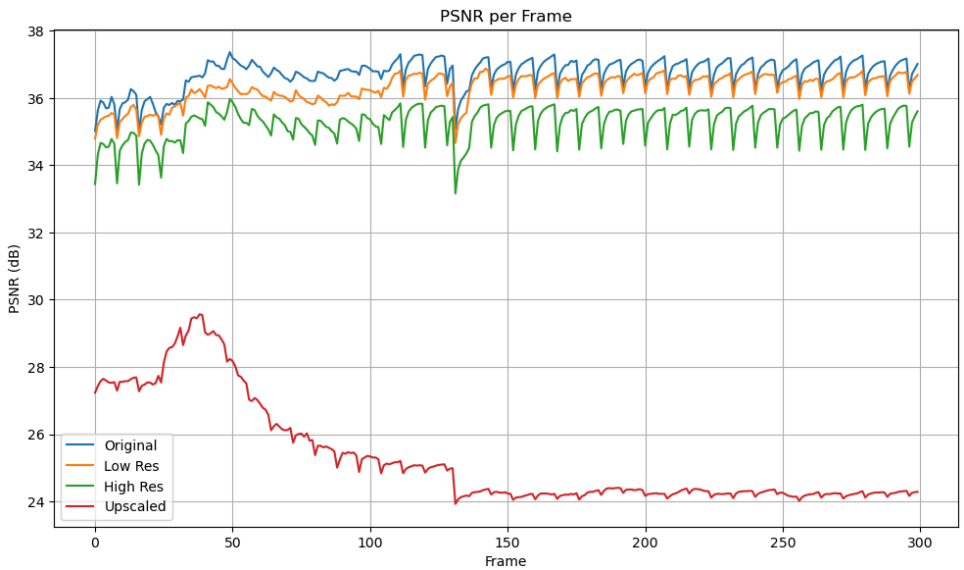

Figure 13: PSNR over time for the different steps of the scalable pipeline with `table` (CABAC, QP of 10 and intra-period of 8)

Figure 14: PSNR foreman

Video File(s) Encoded	Size `table`	Size `foreman`
Original	3.3 MiB	2.7 MiB
Downscaled	1.0 MiB	0.9 MiB
Upscaled	1.7 MiB	1.4 MiB
Enhancement Layer	2.8 MiB	2.3 MiB
Downscaled + Original	4.3 MiB	3.6 MiB
Downscaled + Enhancement Layer	3.8 MiB	3.2 MiB

Table 1: File sizes for scalable pipeline with `foreman` and `table` videos (CABAC, QP of 10 and intra-period of 8)

As we can see, the upscaled-only PSNR is the lowest, as expected due to the spatial down-scaling and up-sampling artifacts. The low-resolution decoded base layer maintains a slightly higher PSNR because it is directly compared at QCIF resolution, and not the original video. The reconstructed high-resolution output, which combines the upscaled decoded base and decoded residuals, shows slightly worse PSNR than the original. This is to be expected, but the good news is that the loss in quality is not that significant and the PSNR remains at or above 35 dB, which is good quality. This loss in quality however is accompanied with also a decrease in the file size, as we can see in Table 1. For both videos, the decrease is of around 10 %. It may seem insignificant here, but with bigger videos this could easily be a deal-breaker.