Virtual Codec for Video4Linux

Thomas Alexander aan de Wiel

November 14, 2016

Contents

Chapter 1

Introduction

1.1 Motivation

Due to the complexity of video and due to applications interfacing with drivers of the linux kernel for the video hardware, it is hard to find where exactly bugs reside in the code. Furthermore, due to a wide range of available video hardware, each supporting different feature sets, it is impractical to test that your application will work on all hardware that is out there . To alleviate this problem, a linux-driver called `vivid` was developed, which is able to emulate all kinds of video-hardware with specific feature sets. This works great for developers to test their applications without in fact owning the hardware they would like to support.

Testing applications however is just one part of the problem. To ensure that the drivers interfacing with certain types of video hardware behave correctly, a compliance tool, `v4l2-compliance` was developed.

With the advance of smartphones, using accelerators to speed up and/or reduce power usages becomes increasingly prevalent in SOC's. This also includes accelerators for video decoding. Testing drivers for this kind of hardware that handles a certain video codec [1] is not yet fully supported by the `v4l2-compliance` tool.

To add support for those kind of codec-drivers, without introducing the problem of a setup with real hardware and getting familiar with it, it is useful to have a virtual kernel driver that implements a particular video codec that is representative for those hardware-codecs. This driver can then serve as a starting point for implementation of other virtual codec-drivers.

[1] A video codec is an electronic circuit or software that compresses or decompresses digital video [1]

1.2 Problem statement & Project goals

The problem statement for this project is as follows:

How to develop a fast codec for use in compliance testing of modern video codecs?

Note that is this problem statement is intended for not just a specialization project, but for extension to a master thesis as well.
Answering this problem statement can be subdivided into three subtasks:

1. Develop a video codec
2. Develop a virtual kernel driver for this video codec
3. Add support for compliance testing of codec drivers to `v4l2-compliance`

The goal of this specialization project involves only the first subtask, that is, to develop a video-codec for use in a virtual kernel driver. This will then serve as a starting point for adding support for codec drivers in the v4l2-compliance test tool and testing those drivers.

1.3 Requirements

The requirements regarding the video-codec to be developed are as follows:

1. The video codec should be *fast*. Fast in this context is meant as the video codec being able to encode/decode 720p video at a 30fps frame rate on a modern laptop.
2. The video codec should include characteristics of modern video codecs. This among other things includes:
    (a) reference frames
    (b) emulation of macro-block behavior
    (c) Variable compressed sizes of macro-blocks
    (d) On-the-fly resolution switching
3. Support for two modes:
    (a) Mode 1, in which the headers for decoding the video frames are *included*
    (b) Mode 2, in which the headers for decoding the video frames are lacking in the output.
4. The video codec should be well suited to be implemented using integer arithmetic only. [2].

[2]Since it will eventually run in kernel mode where floating point operations cannot be used

5. The video codec should provide video with an acceptable quality.
6. The video codec should be 'simple': the virtual codec driver will be integrated into the kernel.
7. The video codec should be open-source (GPLv2)

1.4 Overview

Chapter 2 will cover the background of video codecs. Chapter 3 will give an overview of several video codecs, summarize certain design decisions and cover the developed codec. Results of our developed video codec can be found in Chapter 4 and are reflected upon in Chapter 5. This report will be concluded in Chapter 6.

Chapter 2

Background

In its simplest form, digital video is a set of video frames (still images). A file containing digital video without any compression applied is called raw video. Even though there is no compression applied, several ways exist to represent the video frames. With compression applied, we enter the world of video-codecs which will be our main focus.

In this chapter we will start with explaining some main concepts behind digital video. We then continue by sketching the outline and explaining the concepts behind a general video codec.

2.1 Introduction

2.1.1 Video Frames

A digital video is a set of *video frames*. A video frame is a still image, that itself consists out of *pixels/pels*. The pixels are the sampled points of the scene that was captured on video.
This is schematically illustrated in Figure 2.1.

2.1.2 Color space

Unlike the early days of video, nowadays virtually all videos are recorded in color. The human eye contains three types of receptors (*cones*) that are each sensitive to mainly *one* color. Colors can therefore be approximately represented for us humans, using 3 numbers (each number corresponding to the excitation of a type of receptor)[2].

A *color space* is what defines the link between objective measurements of colors (wavelengths) and our perception of those colors. When using 3 components, then those 3 components specify the location of a certain color in the color space. The *gamut* of a color space defines the range of perceivable colors that can be represented.

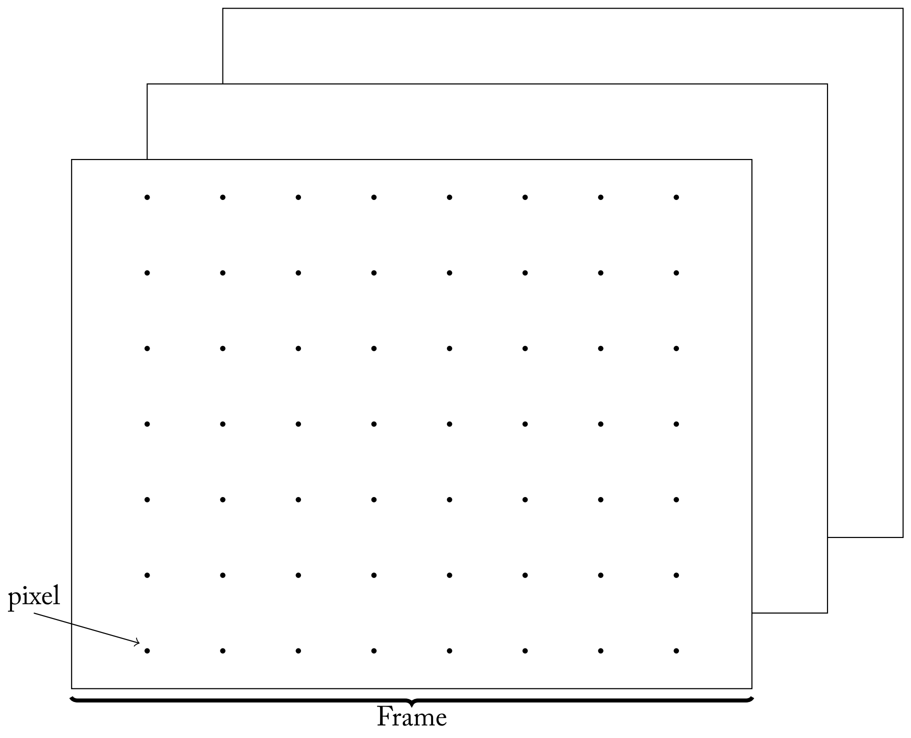


Figure 2.1: Schematic overview of a digital video

There are multiple color spaces available, each with their own purpose. One common family of color spaces is the RGB family, and is used in virtually every computer or TV system. [3].

2.1.2.1 The RGB-color

The family of RGB color spaces uses three components, **R**ed, **G**reen and **B**lue to describe a color. Usually each component is encoded using the same amount of bits. We can distinguish between a linear RGB color space and a non-linear one. Linear here means, that to double the intensity of a certain color component, one simply doubles the value of that component. However doubling the intensity, does not necessarily correspond to a doubled intensity for us, humans. A non-linear RGB color space specifies a non-linear transfer function such that the color space becomes visually linear. Usually we refer to the components of those non-linear RGB color spaces as $R'G'B'$ instead of just RGB for the linear variants.

2.1.2.2 $Y'CbCr$

$Y'CbCr$ is an *encoding* of an $R'G'B'$ color space, and is sometimes also denoted as YCbCr, $Y'UV$ or simply YUV. The three components are:

1. Y' : luminance component
2. Cb: chrominance, the difference between the blue component and Y'
3. Cr: chrominance, the difference between the red component and Y'

The advantage of encoding a color in this way, is that one can take the human visual system into account: the human eye is very sensitive to changes in the luminance component and less sensitive to change in the chrominance components.
Once can therefore use a lower resolution for the chrominance components than for the luminance component, without a significant perceptive loss in quality. This is referred to as *chroma-subsampling* and provides compression based on the characteristics of the human visual system.

2.1.2.2.1 Notation Chroma-subsampling is usually denoted by a string of 3 integers separated by colons: `a:b:c`. In this string the 3 integers a, b, c represent the following:

1. `a` denotes the horizontal luma sampling rate with respect to a reference sampling rate
2. `b` denotes the *horizontal* chroma sampling with respect to a reference sampling rate
3. `c` denotes the *vertical* chroma sampling. Either the same as b (No vertical subsampling) or zero (vertical subsampling by 2).

Some common chroma-subsampling schemes include

1. `4:4:4` (no subsampling)
2. `4:2:2`
3. `4:1:1`
4. `4:2:0`

2.2 Macro-blocks

A macro block refers to a grouping of luma and chroma (sub)blocks that cover a certain area. [4]. Macro blocks commonly refer to an organization of a 4:2:0 subsampled YUV video frame. Since the two chroma planes have half the amount of samples in both the vertical and horizontal direction, an 8×8 block in one of the

chroma planes, corresponds to a *larger* area in the original frame than an 8×8 block in the luma plane.
A macro-block based on a 4:2:0 chroma subsampling usually consists out of the following :

1. four 8×8 blocks of luma samples
2. one 8×8 block of chroma samples of each chroma plane

2.3 Video-Codec

Video codecs are all about *compression*: storing the original video using less space. Compression itself can be subdivided into two categories:

- *lossless*: the video is compressed by removing statistical redundancy. It is possible to exactly recover the original video
- *lossy*: Besides moving redundancy, some information is thrown away in lossy compression. Its compressed result is an approximation of the original video when decompressed.

Due to the high-bandwidth requirements of uncompressed video, and due to lossless compression only giving a moderate decrease in space [5], video codecs usually make use of lossy compression as well.
A video contains 3 dimensions which are represented by:

1. the spatial domain (2D)
2. the temporal domain

Further compression of video can be achieved by reducing redundancy in those domains.
The major video codecs tend to do redundancy removal in those two domains separately[6].
Reduction in the temporal domain is usually done using techniques called *Motion compensation* and *Motion Estimation*, whereas reduction in the spatial domain is usually done using *Transform coding*.
Once Motion compensation and Transform coding have taken place, the obtained representation can be *entropy-coded*, that is, the statistical redundancy is removed (lossless compression).
A general block diagram of a video codec as just described is shown in Figure 2.2 and will serve as a reference for development of our own video-codec.

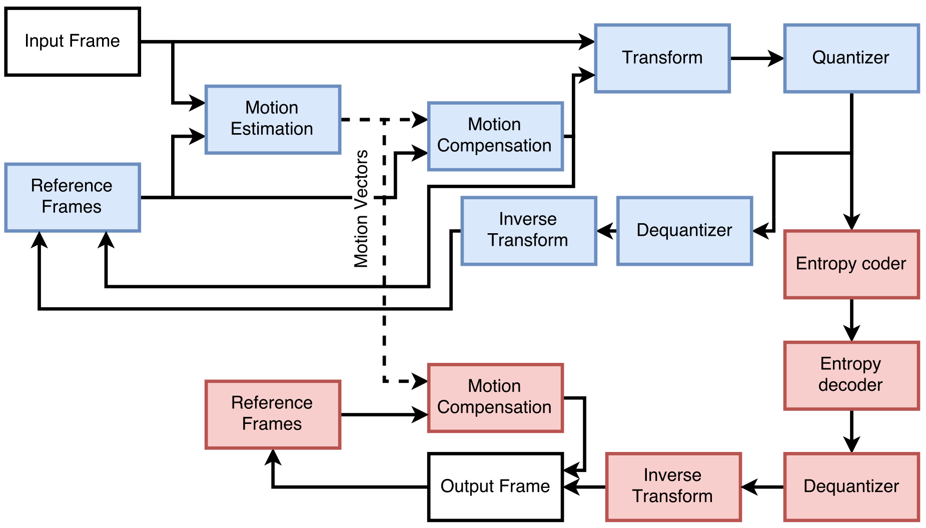
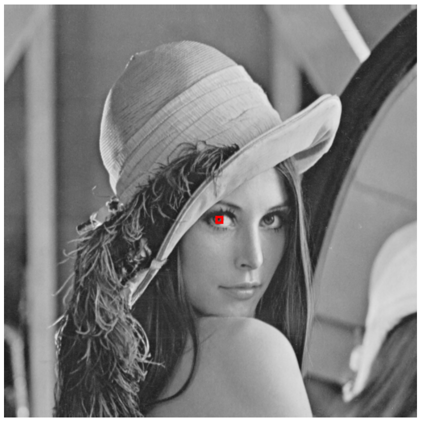


Figure 2.2: A reference codec, [5, 6, 7]

2.4 Transform coding

Transform coding is a compression technique that reduces the spatial redundancy in images/video frames. The idea is based on the observation that pixels in a small section of an image/picture are usually highly correlated [8].
By using suitable transform, the data from the spatial domain can be transformed into another domain, in which the pixel-energy is mostly concentrated in a small number of coefficients for typical input data[4].
Take for example the 8×8 block as indicated in the *Lena* image, see fig. 2.3.


(a) Lena image, with indicated block to be transformed

(b) Enlarged selected block

Figure 2.3: Lena image and selected block

By applying a discrete cosine transform (see section 2.4.2), we obtain the following coefficients [1]

$$\begin{pmatrix} 50 & 40 & 46 & 47 & 75 & 62 & 69 & 94 \\ 45 & 39 & 35 & 40 & 87 & 87 & 65 & 86 \\ 48 & 37 & 35 & 38 & 55 & 90 & 65 & 50 \\ 76 & 47 & 40 & 42 & 68 & 112 & 77 & 56 \\ 90 & 72 & 66 & 66 & 90 & 108 & 74 & 53 \\ 98 & 84 & 84 & 91 & 83 & 72 & 57 & 66 \\ 79 & 86 & 90 & 80 & 76 & 55 & 65 & 113 \\ 57 & 54 & 60 & 57 & 64 & 77 & 107 & 160 \end{pmatrix}$$

Figure 2.4: Block data

$$\begin{pmatrix} 555 & -66 & 21 & 34 & 15 & -20 & 24 & 0 \\ -75 & -23 & -24 & 48 & -9 & -5 & 3 & 4 \\ 2 & -54 & 35 & -66 & 29 & -16 & -8 & 13 \\ 31 & 58 & -21 & -13 & 17 & -3 & -7 & 7 \\ 9 & -33 & 16 & 15 & -7 & 6 & 3 & 2 \\ -10 & 18 & 6 & -13 & 0 & 5 & 0 & 0 \\ -13 & 4 & 1 & -2 & -3 & 14 & -2 & -5 \\ -4 & 7 & -4 & -6 & -7 & 0 & -2 & 2 \end{pmatrix}$$

Figure 2.5: DCT coefficients

[1] Rounded to integer values

We can clearly see that the highest coefficients are concentrated in a small number of coefficients in the upper left corner. These coefficients correspond to the lower spatial frequencies of the image.
Transformation of the data itself does not achieve any compression. However, if only a few coefficients carry most of the energy, most of the other coefficients can be neglected/given a lower priority (which saves bits to encode them), which is referred to as *quantization*.
It turns out that the human eye is less sensitive to information contained in the higher spatial frequencies[7]. To for example lossy compress the data obtained by using the DCT as we did above, we could discard/assign far fewer bits to the higher spatial frequencies. This is for example how JPEG's lossy compression works. Quantizing the transformed coefficients with the the default JPEG quantization matrix results in

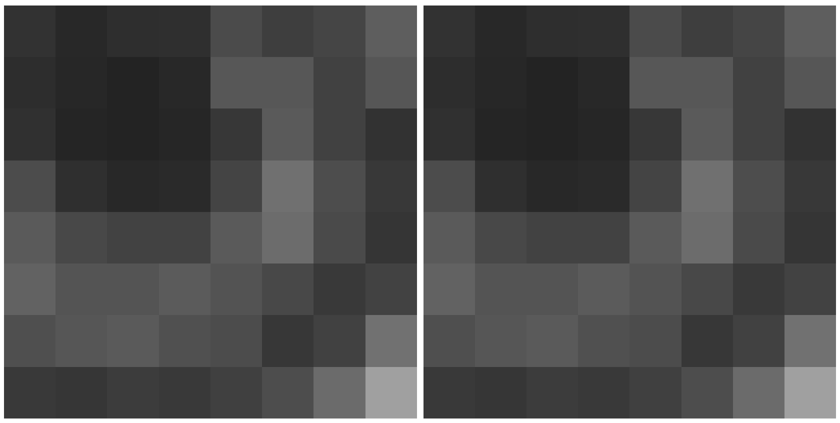
$$\begin{pmatrix}
35 & -6 & 2 & 2 & 1 & -1 & 0 & 0 \\
-6 & -2 & -2 & 3 & 0 & 0 & 0 & 0 \\
0 & -4 & 2 & -3 & 1 & 0 & 0 & 0 \\
2 & 3 & -1 & 0 & 0 & 0 & 0 & 0 \\
1 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

As can be observed, the lower right part of the matrix contains a substantial amount of zeros, which can be efficiently compressed. In order to approximate/reconstruct the original input one should dequantize the coefficients and use apply an inverse transform. If one dequantizes the coefficients in section 2.4 and applies the inverse DCT, one obtains section 2.4. Observe the striking similarity between the two pictures in fig. 2.6.
Transformation can be done on a block-basis (block transform coding), in which a frame is decomposed into smaller blocks and in which each block is transformed independently. Besides block transform coding, there also exist other transform methods based on for example wavelet analysis (works on the whole image) and pyramid schemes[9]. Since our codec should emulate macro-block behavior, we will restrict our focus to block-based transforms.

2.4.1 Some block-transforms in video-codecs

Relevant/Lightweight block-transforms in video codecs are:

1. Discrete Cosine Transform(DCT). This is a very popular block transform and is used in the MPEG compression schemes, in the h26x codec family (h261, h262, h263)[10, 11, 12]. and in the open codec theora as well [13]. Furthermore the open codec "thor" developed by cisco, VP8 and the commerical h264 codec are using an integer transform that approximates the DCT. [14, 15]. An open codec in development, "daala", is based on the DCT as well (Although a lapped variant).



(a) Original block (b) Block after inverse transformation of quantized coefficients

Figure 2.6: Original and decompressed block

2. Discrete Walsh Hadamard Transform (DWHT): The DWHT finds it use for example in intra-coding in H264/AVC [4]. It is furthermore applied in VP8, in which it is applied on the coefficients produced by a DCT transform (secondary transform)[15].

Of those two transforms, the DCT has better energy compaction properties than the DWHT. [16, pp. 419].

2.4.2 Block-Transform coding

In block-transform coding a reversible, linear transformation is used to transform the data[17]. A general 2-dimensional linear transform can be defined as

$$\mathbf{Y} = \mathbf{A}\mathbf{X}\mathbf{A}^T \tag{2.1}$$

with $\mathbf{A}$ the transform matrix, $\mathbf{X}$ the matrix with input data and $\mathbf{Y}$ the matrix containing the transform coefficients.
The ideal linear transform would pack as much image energy into as few coefficients as possible, which is referred to as *energy compaction*. It can be shown that the KLT (Karhunen-Loeve Transform) is optimal in the sense of energy-compaction. Its use in compression however is very limited since the transform is dependent on the data to be transformed. That means that when applying an inverse transformation to retrieve the original data, additional information to perform the inverse transformation is required.[11]
We will therefore restrict our discussion to the already mentioned block transforms in section 2.4.1.

2.4.2.1 The Discrete-Cosine-Transform

The 1D DCT of a sequence $x(n)$ with N samples in its most popular form can be defined as [18, 4]

$$X(k) = \sum_{n=0}^{N-1} \sqrt{2}N\epsilon(n)x(n)\cos\left[\frac{\pi}{N}n\left(k+\frac{1}{2}\right)\right] \tag{2.2}$$

$$\tag{2.3}$$

with

$$\epsilon(n) = \begin{cases} \frac{1}{\sqrt{2}} & n = 0 \\ 1 & \text{elsewhere} \end{cases} \tag{2.4}$$

This can be written in matrix form as:

$$\mathbf{F} = \mathbf{Q}\vec{x} \tag{2.5}$$

Where $\vec{x}$ is an N-dimensional vector and $\mathbf{Q}$ is the $N \times N$ DCT matrix with

$$Q_{ij} = \sqrt{2}N\epsilon(j)\cos\left[\frac{\pi}{N}j\left(i+\frac{1}{2}\right)\right]$$

The matrix $\mathbf{Q}$ is orthonormal, so the inverse DCT transform is obtained by multiplication with $\mathbf{Q}^T$.
Extension to 2D follows eq. (2.1).

2.4.2.2 Walsh-Hadamard Transform

The Walsh-Hadamard transform is based on the Hadamard matrix $\mathbf{H}$, which contains solely 1's and -1's. The *naturally ordered* Hadamard matrix $\mathbf{H}_{2M}$ of size $2M \times 2M$ can be recursively defined in terms of $\mathbf{H}_M$ if M a power of 2:

$$\mathbf{H}_2M = \begin{pmatrix} \mathbf{H}_M & \mathbf{H}_M \\ \mathbf{H}_M & -\mathbf{H}_M \end{pmatrix} \tag{2.6}$$

With $\mathbf{H}_1 = 1$. eqs. (2.7) to (2.9) show the naturally ordered Hadamard matrices for $M \in \{1, 2, 4\}$.

$$\mathbf{H}_1 = (1) \tag{2.7}$$

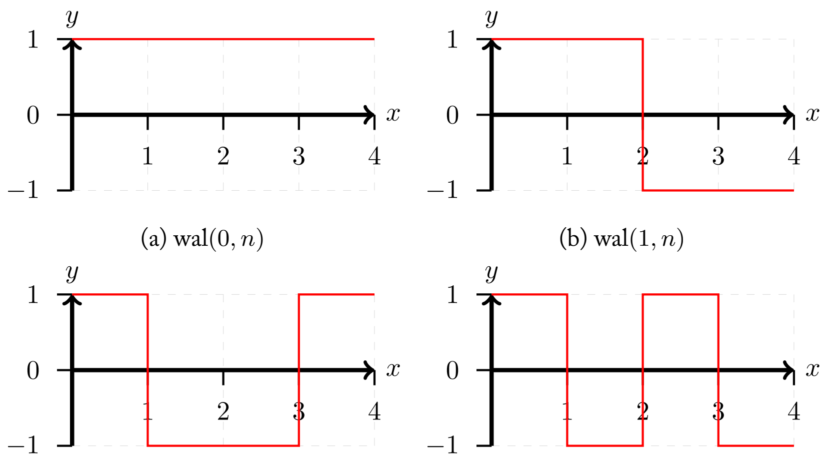
$$\mathbf{H}_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.8}$$

$$\mathbf{H}_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{2.9}$$

2.4.2.2.1 Walsh-functions A complete set of $N = 2^M$ discrete Walsh functions wal(m, n) can be defined as follows [19]:

$$
\begin{aligned}
&\text{wal}(0, n) = 1, && \text{for } n = 1, 2, \ldots, N \\
&\text{wal}(1, n) = \begin{cases} 1 & \text{for } n = 1, 2, \ldots N/2 \\ -1 & \text{for } n = \frac{N}{2} + 1, \frac{N}{2} + 2, \ldots N \end{cases} \\
&\text{wal}(m, n) = \text{wal}(\lfloor \frac{m}{2} \rfloor, 2n) \cdot \text{wal}(m - 2\lfloor \frac{m}{2} \rfloor, n)
\end{aligned}
$$

With $m = 0, 1, \ldots, N - 1$ and $n = 1, 2, \ldots, N$.
Plotting those for $N = 4$ results in fig. 2.7.



(a) wal$(0, n)$ (b) wal$(1, n)$

(c) wal$(2, n)$ (d) wal$(3, n)$

Figure 2.7: Walsh functions for $N = 4$

Observe that with increasing m, we also have an increasing amount zero-crossings wal(m, n). The amount of zero-crossings gives rise to the term *sequency*, the amount of sign changes, which gives a frequency interpretation of those function[20].
If we interpret each row of the Hadamard matrix as a wave form, we get ”pulse wave-forms” similar to Walsh-functions. Observe how for $N = 4$ the set of Walsh functions exactly represent the wave-forms in the $\mathbf{H}_4$ (see eq. (2.9)). If we look at the rows of $\mathbf{H}_4$ we observe that the amount of sign changes $(0, 4, 1, 2)$ is not in increasing order. This is due to the natural ordering of the Hadamard matrix. Commonly used in image processing is the *sequency*-ordered Hadamard matrix in which the sign changes per row is strictly increasing.
The Walsh-Hadamard transform of a vector $\vec{x} \in R^{2^M}$ can be achieved by multiply-

ing it with the sequency ordered Hadamard matrix $\mathbf{H}_M$:

$$\vec{x}' = \mathbf{H}\vec{x} \tag{2.10}$$

In which $\vec{x}'$ denotes the transformed vector. Again, the 2D version of the Walsh Hadamard transform follows eq. (2.1). Since the sequency ordered Hadamard matrix is symmetric [19] and orthogonal:

$$\mathbf{H}_m^T \cdot \mathbf{H}_m = M^2\mathbf{I}$$

Inverse transformation can therefore be achieved by multiplying the transformed vector $\vec{x}'$ with $\frac{1}{M^2}\mathbf{I}_M$.

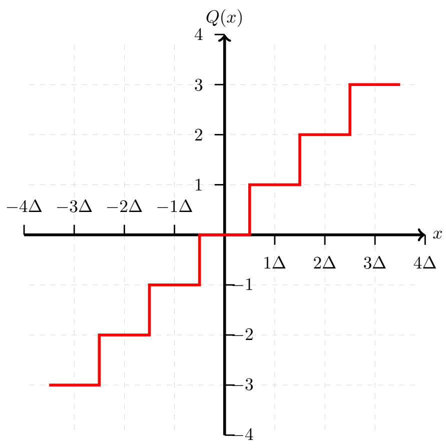
2.5 Quantization

Quantization is the process of mapping a range of input values to a smaller set of output values. Quantization is therefore an *irreversible* process, since several distinct input values might be mapped to the very same output value. At the same time, quantization makes it possible to compress the input data more: after quantization, less bits are required to represent all possible values.
Input values to a quantizer might have a continuous range, as is the case in for example ADC's. In the context of video codecs, however, the quantizer will be fed input values with a discrete amplitude. Hence the quantizer will further restrict the range of possible discrete values.
In image coding the quantization should give less priority/assign fewer bits to coefficients that are visually unimportant, while retaining/slightly assign fewer bits the more significant coefficients. Which transform coefficients are considered important is not only dependent on the desired compression to be obtained, but it is also dependent on the human visual system. Certain coefficients of the transformed data are of more importance for the human eye than others.
We can distinguish two types of quantization:

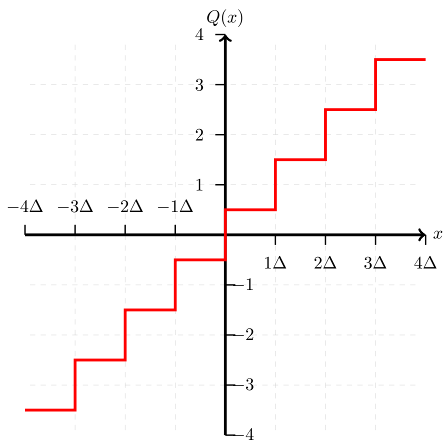
1. *Scalar quantization*: as it name implies, scalar quantization scales every coefficient(scalar) independently. The output of scalar quantization is an index corresponding a certain value of the quantized coefficient.

2. *Vector quantization*: instead of quantizing every scaling seperately, vector quantization quantizes vectors of coefficients. The output of vector quantization are indices as well, however they correspond to vectors instead of scalar values.

Scalar quantization is the quantization type typically used in video codecs [21] and is easier to implement. We will therefore not cover vector quantization. Scalar quantization can be further subdivided into uniform and non-uniform quantizers.



(a) Midtread Quantizer



(b) Midriser Quantizer

Figure 2.8: Two uniform quantizers

2.5.1 Uniform Quantization

An uniform quantizer makes use of constant *Step-size* Δ to map input values to a reduced set of output values. We can distinguish between two types of uniform quantizers: the mid-tread and mid-riser quantizer, see also fig. 2.8.
A mid-riser quantizer can be described with the following formula:

$$Q(x) = \Delta \left(\lfloor \frac{x}{\Delta} \rfloor + \frac{1}{2} \right)$$

A mid-tread quantizer on the other hand can be described using:

$$Q(x) = \Delta \lfloor \frac{x}{\Delta} + \frac{1}{2} \rfloor$$

The difference between the two quantizers lies in how they map values around zero to their respective output values. Effectively, mid-riser quantizers cannot map their inputs to a zero output, whereas mid-tread quantizer can.

2.5.1.1 Non-uniform quantizer

In non-uniform quantizers the step-size Δ is non-constant. One commonly used non-uniform quantizer is the *dead-zone-quantizer*. Unlike the uniform quantizers, a broader range around zero is mapped to zero, see also fig. 2.9.
The deadzone-quantizers finds it use in video codecs, where it is sometimes desired that small values are quantized to zero in order to obtain a better compression ratio. By using a deadzone-quantizer, the encoder will not unnecessarily allocate bits for

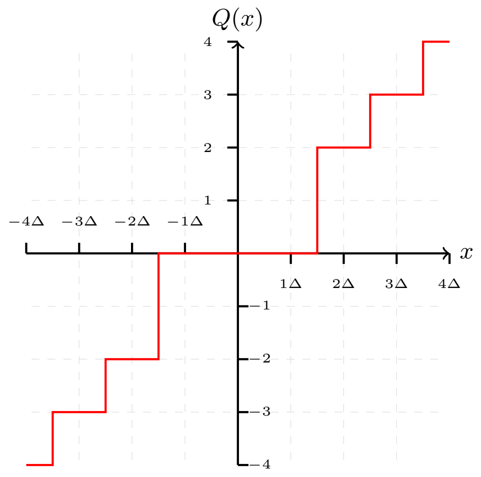


Figure 2.9: Deadzone-quantizer

noisy signals around zero[4, pp. 86] (signals around zero that are not noise however, will be quantized more coarsly with a deadzone quantizer).

2.6 Motion Compensation & Motion Estimation

2.6.1 Introduction

We have introduced transform coding as a method to remove spatial redundancy. In principle one could apply transform coding to every frame of a video, but in practice it turns out that one can obtain more compression by exploiting temporal redundancy as well[4].
The main idea behind removing temporal redundancy is that adjacent frames in a video are usually highly correlated[22]. By *predicting* a frame based on a nearby frame it is possible to achieve a higher coding efficiency.
The simplest method would be to predict a frame by encoded the *difference* between the frame and a certain reference frame. This reference frame, could for example be the previous frame. Since adjacent frames usually are very similar, the difference between them should usually be small and be easily compressable.
Most video codecs, however use more advanced techniques called *Motion Estimation* (ME) followed by *Motion Compensation*(MC) to reduce the temporal redundancy. ME tries to find out how parts in the reference frame(s) have moved with respect the current frame. *Motion vectors* describe how certain parts of an (or even the whole) image have moved with respect to a reference frame (see fig. 2.2). Motion vectors can be as simple as describing only translational movement, up to specifying rotation.

MC uses the motion vectors from the motion estimation to compensate for movement between the reference frame(s) and the frame being encoded. If for example the current frame has moved to the right (due to camera panning) with respect to a reference frame, this could be described by a motion vector specifying the translation for the the whole frame. The difference of the reference frame and the motion compensated current frame (translate it back as much as the motion vector specifies) will be smaller then taking the direct difference between the reference frame and current frame: in this way ME + MC can be used to achieve a better compression ratio.

A reference frame should be a frame that is *decoded* before the frame that is currently considered. This does not imply that the frame is also displayed *before* the current frame in the decoding process. This motivates the categorization of frames into:

1. I-frame: intra-frame. Encodes a whole frame, without the use of reference frames
2. P-frame: inter frame. Encoded using a reference frame that is displayed in the past.
3. B-frame: inter frame. Encoding using references frames in the past and future.

It turns out that motion estimation is computationally the most expensive part of the video encoding process, and can take up to 90% of the computation time when implemented in software [23, 24].

2.7 Entropy Coding

To store the coefficients obtained after quantization video codecs typically perform entropy coding to reduce the statistical redundancy.
Popular entropy-coding methods include:

1. Run-level coding
2. Arithmetic-coding
3. Huffman coding

2.7.1 Run-length encoding & Run level coding

Run length encoding is a lossless method to compress data based on replacing repeating data symbols with the the amount of repetitions and the symbol that is repeated. For example, consider the sequence

$$\{5, 5, 5, 4, 4, 4, 4, 4, 4, 4\}$$

This sequence could be represented as

$$\{\{3, 5\}, \{7, 4\}\}$$

Where each tuple $\{a, b\}$ describes the amount of repetitions (a) and the value (b) being repeated. This compression method is particularly effective for data with long runs of the same symbol.
As mentioned before, transform coding is often targeted at producing zero coefficients. It would therefore be useful to not encode those zeros. If we consider the following sequence with a substantial amount of zeros:

$$\{5, 0, 0, 0, 0, 3, 4, 0, 0, 0, 0, 5\}$$

This sequence could be compressed by only stating where the *non-zero* components are located using tuples like ({offset,value}). In this case that results in:

$$\{\{0, 5\}, \{5, 3\}, \{6, 4\}, \{11, 5\}$$

This form of lossless compression is called run-level coding.

2.7.2 Huffman coding

Named after David Huffman, Huffman coding tries to compress data by replacing symbols that are likely to occur with shorter codes and less-frequent occurring symbols with longer codes. Huffman coding is a form of *prefix-codes* [25]
A prefix code for a set of symbols is a collection of code words (one per symbol) in which no code-word forms a prefix of another code-word.

Consider an alphabet of symbols, S. In order to use Huffman coding, one needs statistics about the frequency of occurrence of the letters of this alphabet to be encoded, F. Using this information, a *Huffman* tree is created using the following method [26]:

Algorithm 1 Huffman Tree

```
1: procedure HuffmanTree
2:     if $|S| = 2$ then
3:         Encode one letter with 0, the other with 1
4:     else
5:         Create a forest with a single node tree for each symbol in $S$
6:         label each tree with the frequency of the corresponding symbol
7:         while More than one tree in the forest do
8:             select the two trees (a,b) labeled with the lowest frequencies
9:             remove those trees from the forest
10:            Create a new symbol $\omega \leftarrow$ letter with $f_\omega = f_a + f_b$
11:            create a node corresponding to $\omega$ with $a, b$ as its children
12:            Label the resulting tree with $f_\omega$
13:            Insert the tree in the forest
14:        end while
15:    end ifreturn The only tree left in the forest.
16: end procedure
```

The prefix code for a symbol can then be found by traversing the tree to the node corresponding to that symbol, recording a '0' every time going to a left node, and a '1' for a right node.
This will then result in an optimal prefix code for the symbol [25].

2.7.2.1 Video codecs

In for example the H261 (but also theora) video codec, the quantized transform coefficients are run-level coded. Common occurring run-level symbols are assigned a Huffman code, whereas symbols that are not considered common are then separately encoded using an escape sequence. This combination of run-level-coding and Huffman coding is a form of Variable-Length-Coding (VLC).

2.7.3 Arithmetic coding

Unlike Huffman coding, which assigns codes on a symbol bases, arithmetic coding assigns *one* fractional number (between 0 and 1) to multiple symbols at once[27]. This makes it possible that a symbol can be encoded using less than 1 bit [28], which is not possible in Huffman Coding, as the smallest prefix code is 1-bit in size. Implementation-wise, arithmetic coding is more complex than Huffman coding [29], and explaining the mathematical concepts behind it, is out of scope of this report.

2.8 Video-quality

With lossy compression it is possible to trade video quality for compression ratio. In order to make sure the quality of a video meets the demands, several statistics exist. Common objective statistics include the Mean Squared Error(MSR) or Peak-Signal-to-Noise-ratio (PSNR). Do note that those metrics do not always correspond well to the quality that we as humans perceive [30].
Since the video-quality of our codec is to be *acceptable*, we will restrict ourselves to the most popular objective statistic, the PSNR.

2.8.1 PSNR

The PSNR of a video frame is defined as follows[30]:

$$PSNR = 10 \cdot \log_{10} \frac{L^2}{MSE} \tag{2.11}$$

with L the dynamic range of the pixel values and MSE being the mean squared error:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (x_i - y_i)^2 \tag{2.12}$$

In which N is the total amount of pixels, x_i the ith pixel in the decoded frame and y_i the pixel in the original frame.
In tests it has been shown that PSNR values higher than 20dB correspond to an 'acceptable' video quality [31].

Chapter 3

Design of a new video codec

This chapter will give an overview of the design-process of our video codec. It will do so by starting with a selection and comparison of several techniques that are candidates for being used in our codec, followed by a more practical description of how those techniques were applied.

3.1 Method

In order to develop a fast, for modern video codecs representative, video codec several tasks need to be completed

1. Produce an overview of popular modern video codecs and their main characteristics
2. Comparison and selection of techniques regarding following:
    (a) Spatial compression (Transform coding)
    (b) Temporal compression (Reference frames)
    (c) Entropy coding
3. Combine the selected techniques into a *custom* video codec.

3.2 Overview of codecs

In this section we will look at a selection of (modern) codecs. This selection is partly based on popularity, but also on the patents/"openness" of a codec.

3.2.1 Patents/Open codecs

It is worth to pay some attention to the aspect of licenses, patents and the conditions of using a certain codec. Since the codec to be developed is to be part of the Linux kernel, it is important that the ideas and techniques used do not infringe any patents.

For example, US patents expire after 20 years [32], meaning that many specific techniques as in current video-codecs are likely to be covered by patents.
Furthermore if a codec is open source, this does not mean that one can copy and use this code in the Linux kernel as this might result in conflicts between the licensing of the Linux kernel (GPLv2) and that of the codec.

3.2.1.1 Open codecs

With the increase of devices connected to the internet, the availability of a codec that can be used freely increases (motivate). Several open codecs are out there/in development. Two examples are:

1. Theora
2. Daala

Both codecs are/have been developed by the Xiph.Org Foundation.

3.2.1.2 Commercial codecs

Popular commerical codecs include:

1. H26x family of codecs, developed by ITU-T.
2. MPEG family of codecs developed by the Moving Picture Experts Group.
3. VPx family of codecs: first developed by ON2, from VP8 onwards owned and developed by google.

3.2.1.3 Overview codec characteristics

In this overview we consider several open (source) codecs, popular codecs as well as old codecs (interesting due to the expiration of their patents).

Table 3.1: Overview codec characteristics

Codec	**Transform**	**B-Frames**	**P-Frames**	**Entropy coder**
Theora	DCT II (8x8)	×	✓	RLC + huffman
Daala	Lapped DCT	✓	✓	Arithmetic coding
VP8	DCT (4x4) followed by WHT(4x4)	×	✓	Arithmetic Coding
h264/MPEG4 AVC	Integer approximation DCT (4x4)	✓	✓	Arithmetic Coding
h261	DCT (8x8)	×	✓	RLC + huffman
MPEG1	DCT (8x8)	✓	✓	RLC + huffman
MPEG2	DCT	✓	✓	RLC + huffman

We observe how the codecs, not from the MPEG family do not include B-frames. B-frame coding tools have heavily been patented [1], which might explain their absence in those codecs. Furthermore we observe the fast majority uses a variant of the

[1]urlhttp://www.avs.org.cn/avsdoc/2003-7-30/Cliff.pdf

DCT as their main transform. Whereas modern codecs use arithmetic coding for their entropy coders, older codecs like theora and h261 are based on variable length coded huffman data.

3.3 Transform

As indicated in section 2.4.1, the most popular block-transform used in video coding is the DCT. This, however does not mean that we should solely focus on this transform.
As indicated in section 1.3, the codec to be developed cannot use *floating-point* operations. The DCT, which uses multiplications/division with irrational numbers, would have to be approximated using fixed-point arithmetic. Although it is not impossible, this makes it more complex though. In general, transformations that are not integer based lead to calculation errors which can cause differences between the encoder and the decoder leading to a phenomenon *drift*. With an integer based transform, this will not happen as long as all calculations can be carried out without underflow/overflow.
Integer based transforms include the 4x4 integer approximation of the DCT as introduced in H264 as well as the Walsh Hadamard Transform. Both transforms can be implemented without multiplications and divisions. The Walsh Hadamard transform can even be implemented without the use of bit-shifts.
Besides implementability using integer arithmetic only, the complexity of the transform is of utmost importance as well. It turns out that for both the DCT family of transforms as well as for the Walsh-Hadamard Transform a fast implementation of the transform exists.
For the development of our codec the DWHT was chosen:

- Due to its definition, it can be directly implemented using integer arithmetic
- Mathematically and computation wise, the transform is simple

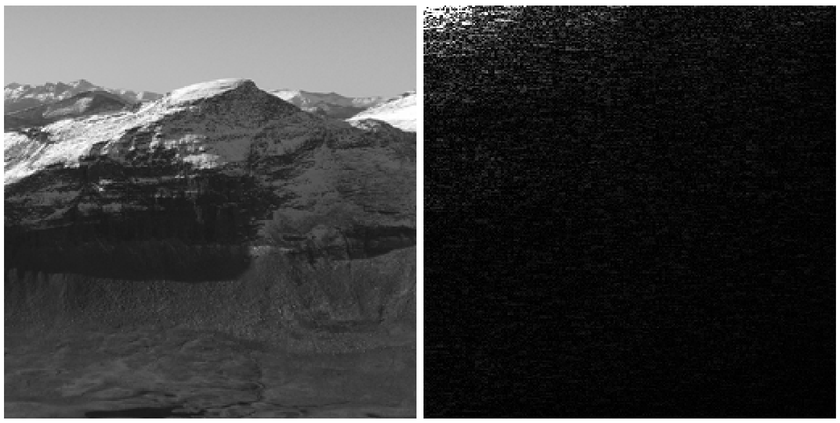
In principle one could have argued the same for the 4x4 DCT transform as in H264. However as the codec needs to be able to be included in the Linux kernel, use of technology as introduced by H264 might lead to patent-infringement.

3.3.1 Walsh-Hadamard Transform

In our codec we make use of the analogy between sequency and frequency. [20] shows that magnitude of the transformed coefficients decreases with increasing sequency, which is visualized and indeed confirmed by the example in fig. 3.1.
Hence the *sequency-ordered* Walsh-Hadamard Transform has been chosen for use in our codec.

3.3.1.1 Fast Walsh Hadamard Transform

Naive implementation of the Walsh-Hadamard Transform as in eq. (2.10) results in an $\mathcal{O}(N^2)$ algorithm. One can do better, by deriving a fast Walsh-Hadamard



(a) Original image (b) Magnitude of the sequency-ordered Walsh-Transform coefficients of fig. 3.1a

Figure 3.1: Example Image and its Walsh-Hadamard transform

transform analogously to famous FFT (Fast Fourier Transform). This, however results in an output that is not sequency ordered [19].
Several attempts have been made to produce a fast transform that produces sequency ordered output data [33, 34, 19]. The first two attempts however, require that one bit-reveres the output. [19] derives a sequency ordered algorithm by decomposing the sequency ordered Hadamard Matrice of size $2^N \times 2^N$ into N sparse matrices. The complete transform is then performed in N stages, in which each stage corresponds with multiplication by one of the sparse matrices.
For an 8 input size, the decomposition of the 8×8 sequency-ordered Hadamard Matrix is as follows (-1 is replaced by simply a - for formatting reasons):

$$\mathbf{H}_{sequency} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & - & - & - & - \\ 1 & 1 & - & - & - & - & 1 & 1 \\ 1 & 1 & - & - & 1 & 1 & - & - \\ 1 & - & - & 1 & 1 & - & - & 1 \\ 1 & - & - & 1 & - & 1 & 1 & - \\ 1 & - & 1 & - & - & 1 & - & 1 \\ 1 & - & 1 & - & 1 & - & 1 & - \end{pmatrix} = \mathbf{H}_1 \cdot \mathbf{H}_2 \cdot \mathbf{H}_3 \qquad (3.1)$$

With

$$\mathbf{H}_1 = \begin{pmatrix} 1 & 1 & & & & & & \\ 1 & - & & & & & & \\ & & 1 & - & & & & \\ & & 1 & 1 & & & & \\ & & & & 1 & 1 & & \\ & & & & 1 & - & & \\ & & & & & & 1 & - \\ & & & & & & 1 & 1 \end{pmatrix} \mathbf{H}_2 = \begin{pmatrix} 1 & 1 & & & & & & \\ & & 1 & 1 & & & & \\ 1 & - & & & & & & \\ & & 1 & - & & & & \\ & & & & 1 & - & & \\ & & & & & & 1 & - \\ & & & & 1 & 1 & & \\ & & & & & & 1 & 1 \end{pmatrix} \mathbf{H}_3 = \begin{pmatrix} 1 & 1 & & & & & & \\ & & 1 & 1 & & & & \\ & & & & 1 & 1 & & \\ & & & & & & 1 & 1 \\ 1 & - & & & & & & \\ & & 1 & - & & & & \\ & & & & 1 & - & & \\ & & & & & & 1 & - \end{pmatrix}$$

Transformation of a vector $\vec{x}$ can then be achieved by

$$\vec{X} = \mathbf{H}_1\mathbf{H}_2\mathbf{H}_3\vec{x} \tag{3.2}$$

This leads to the flow diagram as in fig. 3.2 for an 8 input Walsh-Hadamard Transform[2]. in which a dotted line respectively a solid line mean that the corresponding

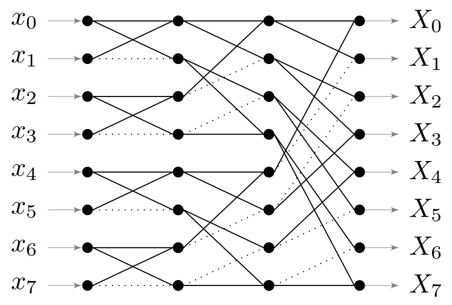


Figure 3.2: Fast sequency ordered Walsh-Hadamard Transform

value is multiplied by -1 and 1. Every node except the input/output nodes represent an addition. Note that only $\mathcal{O}(N\log(N))$ operations have to be performed. Transformation of a 2D-block of data is achieved by applying the fast walsh Hadamard transform row-wise, followed by a column-wise transformation. This corresponds to eq. (2.1). Inverse transformation is obtained by using the same transform algorithm, altered with a scaling factor.

3.3.1.2 Block-size

As [17] indicates, with increasing block size, one obtains a higher level of compression. Increasing the block size however comes at a higher cost as well (more calculations are necessary per pixel). The most popular block sizes are 8×8 and 16×16. In our codec we have chosen for a block-size of 8×8, to have a block-size that is representable but not too computationally intensive. Transformation of an 8×8 block is then performed transforming each row, followed by transforming each column.

[2]Note that this flow diagram was already present in [20], only reversed

3.4 Video frames

Frames in our codec consist out of two planes, similar to `Y'UV420`. `Y'UV420p`, which has a chroma subsampling of 4:2:0 (see section 2.1.2.2), organizes the Y', U and V coefficients all separately in three planes (no interleaving). In our case, The layout of a frame can be described as:

Table 3.2: Layout of a frame

Frame Header
Run-level coded Y' coefficients
Run-level coded U coefficients
Run-level coded V coefficients

Each plane contains the coefficients of all 8×8 blocks. The first 64 coefficients that correspond to the the coefficients of the first block in the frame (upper-left corner). The second sequence of 64 coefficients correspond to the 2nd block (right of the first block) and so on.

3.4.1 Frame types

Our codec has two types of frames: I-frames and P-frames. In order to introduce references frames, our codec is limited to such called "P-frames". Note that this is not unnecessary an unrealistic feature, since several modern codecs do not feature bidirectional frames either.
The type of blocks that make up a frame are

1. I-block: a block containing 8x8 samples of the current frame to be compressed
2. P-block: a block containing 8x8 samples. To keep the codec lightweight [3] it was decided to restrict the search area for motion estimation to the corresponding block in the reference frame. A p-block is therefore obtained by taking the difference of the samples from the current block with samples of the corresponding block in the previous frame.

The first frame in a video is always an I-frame, since there is no reference frame available yet. I-Frames solely consist of I-blocks. A P-frame is defined as a frame containing a mix of I and P-blocks. How many blocks are P-blocks depends on the frame to be compressed, see section 3.4.1.1. Our codec is forced to insert an I-frame after every N P-frames, in which N can be configured. In this way a frame can get lost without making it impossible to decode the rest of the video.
Categorization into I-frame and P-frames is not visible at the bit-stream level: since it is necessary for the decoder to know whether the current block to be decoder is a P or I block, this information needs to be present at a block-level. Replication

[3] see section 2.6.1, motion estimation can take up to 90% of the encoding time

this information in the frame-header is therefore unnecessary. This might change if more complex features are added, which require information about whether the current frame needs a reference frame to be decoded.

3.4.1.1 Decision making

In order to decide whether a block is coded as an I-block/P-block, one should have be able to predict the size of the resulting I/P-block. Simply encoding the current block in both variants, comparing their sizes imposes a lot of additional overhead. Our encoder therefore, analogously to H261 [6], calculates a *heuristic* that is based on the variance. The variance is calculated for both the block to be encoded and the *difference-block* (contains the difference between the current block to be encoded and the block of the reference frame corresponding to the current block). The reasoning behind using the variance is that it measures the spread of the data. Roughly, data that has a higher spread has a higher frequency/sequency content and therefore has more non-zero coefficients than data with a lower spread .

3.4.1.2 Transformation of I- & P-blocks

In case of an I-Block, the coefficients that are being transformed (8 bit) can range from 0 to 255. The maximum value obtainable through Hadamard Transformation can be found from eqs. (3.1) and (3.2). Note that the following situation results in the maximum value for the first coefficient of the transformed vector

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & - & - & - & - \\ 1 & 1 & - & - & - & - & 1 & 1 \\ 1 & 1 & - & - & 1 & 1 & - & - \\ 1 & - & - & 1 & 1 & - & - & 1 \\ 1 & - & - & 1 & - & 1 & 1 & - \\ 1 & - & 1 & - & - & 1 & - & 1 \\ 1 & - & 1 & - & 1 & - & 1 & - \end{pmatrix} \cdot \left(255, 255, 255, 255, 255, 255, 255, 255\right)^T \tag{3.3}$$

Hence, in general, the maximum obtainable value with a Hadamard Matrix of $N \times N$ is $N \cdot 255$. The minimum value obtainable is $-\frac{N}{2}255$: we cannot get any smaller, since each row contains at least 4 '1's and 4 '-1''s. This means that transforming them as is, leads to asymmetry between the smallest and biggest value obtainable. By adding a bias of -128, we obtain more symmetry which makes it easier to encode it using two's complement.
In case of a P-Block, the coefficients that are being transformed can range from -255 to 255: that is, the coefficients have twice the range of the coefficients that are being coded in an I block. Symmetry is already implied by the range, hence adding a bias is not necessary when transforming P-block coefficients.

3.5 Entropy coding

In order to actually profit transforming our data, we need to apply some form of entropy coding. Arithmetic coding has not been considered to be a viable option: not only does it use of fractional numbers make it hard to avoid floating point arithmetic, but the performance (speed-wise) is also lower than that of a huffman coder.
The entropy coding as used in our codec is designed to be simple, and only makes use of a run-level coder. Since the target is not necessarily to achieve a good compression ratio, but to achieve variable compressed block sizes, it has been decided to omit a huffman coder which as applied in for example h261 and theora. Due to the use of a walsh Hadamard transform, instead of the DCT, it is not possible to reuse quantization tables/huffman dictionaries from for example theora. Proper huffman coding would therefore require analysis of a large amount of video data to detect common occurring symbol sequences. Furthermore during testing of the codec, it was found that a deadzone-quantizer resulted in a reasonable compression ratio (see section 3.5.1.1)

3.5.1 Dead-zone quantization & RLC

3.5.1.1 Quantization

Quantization in our codec happens on a block-basis (8×8). For every coefficient, a seperate step-size can be defined. This gives rise to an 8×8 quantization matrix $\mathbf{Q}$, in which element represents the stepsize to be used by the quantizer for the corresponding coefficient in the 8×8 transformed block.
The step-size as used by the quantizer for a certain coefficient is always a power of 2, which makes it possible to quantize the coefficients using (right) bit-shifts instead of usually expensive integer division. Elements of $\mathbf{Q}$ are therefore representing the amount of bits that the quantizer should shift the coefficient to obtain the desired divison by the power of 2.
The quantizer as used in our codec is a deadzone quantizer: it was found during testing, that when no deadzone quantizer is used, the size of the compressed video is around the same size as the original input. This was caused by runs of zero being broken by a small coefficient.
By including a deadzone quantizer, the runs of zero were less frequently broken by non-zero coefficients, resulting in a significant coding efficiency (see section 5.1).
n.

3.5.1.2 RLC

The Run-Level Coding as used in our codec codes every non-zero coefficient in a 16 bit integer. 4 bits of this integer are reserved for indicating the length of zeros and 12 bits for the quantized coefficient (As can be seen from section 3.4.1.2, the maximum/minimum value will be $127 \cdot 8 / -128 \cdot 8$ respectively, taking 12 bits to encode)
The maximum `run` is 14: the value 15 is reserved for zero sequences *longer* than 14. If for example, a block to be transformed is constant (only one unique value), then

15-12	11-0
`run`	`value`

Table 3.3: Encoding of RLC tuple

the Walsh-Hadamard transform of that block results only in the very first coefficient being non-zero. For an 8×8 block size, that means 63 trailing zeros, which would normally have to be encoded using 4 tuples, instead of one.

3.5.2 Zig-Zag Scanning

As discussed in section 3.3.1, the magnitude of the transformed coefficients in general decreases with increasing sequency. Simply scanning the transformed coefficients left to right, top to bottom will therefore not result in scanning the high-frequency components successively.
In a similar fashion to JPEG and other DCT based video codecs, we scan the transform coefficients in a block in a zig-zag fashion (See Fig. 3.3). In this way, the coefficients are ordered in order of increasing horizontal and vertical sequencies. When appropriate quantization is applied to the high-sequency components, scanning in zig-zag fashion is therefore likely to result in a sequence with a substantial amount of trailing zeros/small coefficients. This can then effectively be compressed using Run-Level-Coding.

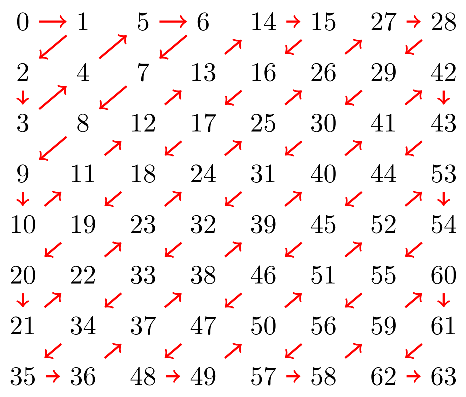


Figure 3.3: Zig-zag scanning for a 8×8 block

Chapter 4

Results

4.1 Method

To evaluate the performance of our codec we focus at four aspects:

1. *Video quality*: video quality will be assessed by calculating the PSNR for the Y'-plane of every frame. The Y' plane is chosen since it is visually most important for us humans (see section 2.1.2.2).

2. *Compression ratio*: the compression ratio, defined as

$$\text{Compression ratio} = \frac{\text{compressed size}}{\text{Original size}} \cdot 100\% \tag{4.1}$$

3. *Compression/Decompression speed*: the time required by the encoder and decoder to compress respectively decompress a single video frame.

4. *Reference frames*: to evaluate the effectiveness of reference frames, we the percentage of P-blocks in every frame will be recorded.

4.1.1 Scenes

In order to produce meaningful results, *raw* videos are used, such that our encoder will not have any advantage/disadvantage due to coding artefacts of the video codec that is used to compress the videofile.
The video samples are two 720p yuv 4:2:0 files from derf's collection [1]:

1. **ParkJoy**: a scene having a difficult coding difficulty [35]

2. **OldTown**: a scene having an easy coding difficulty [35]

Both scenes 500 frames long.

[1]`https://media.xiph.org/video/derf/`

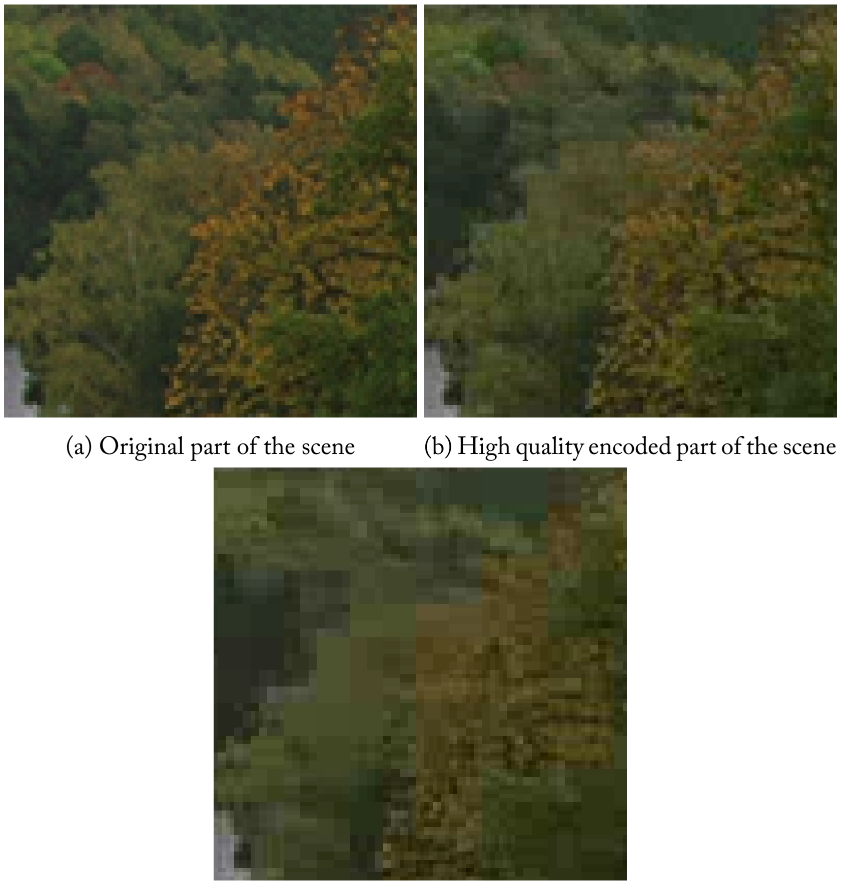
4.1.2 Quantizer

The same quantization matrix (defining the step-size) is to be used for both I- and P-blocks. In order to test the ability of the encoder to compress more at the cost of reduced video quality, a "high-quality" (eq. (4.2)) and "acceptable-quality" quantization matrix (eq. (4.3)) are to be used (note how those matrices follow the zig-zag pattern as in section 3.5.2).
As video-quality is not objective, a subjective trial-and-error method has been used by the author to determine those matrices. High-quality was considered to be a video in which small details/colors *within* macroblocks were not significantly degraded. For the acceptable matrix, some significant loss of this small detail within macroblocks was allowed, but the overall quality should not be severly degraded with respect to the original video. Note that we focus on detail *within* a macroblock, since blocking artefacts at the edges of macroblocks are inevitable when lossy compression is applied. Due to transforming blockwise, quantization is also performed blockwise. This means that the sequency components between blocks are not treated in exactly the same way. This leads to discontinuities at the boundaries. Usually, codecs apply a deblocking-filter, which smooths out those discontinuities. Our codec however, does not include such a filter (due to speed-reasons) An example illustrating the quality degradation related to the the high and acceptable encoding of a frame can be found in fig. 4.1.

$$\begin{pmatrix} 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 3 \\ 2 & 2 & 2 & 2 & 2 & 2 & 3 & 4 \\ 2 & 2 & 2 & 2 & 2 & 3 & 4 & 5 \\ 2 & 2 & 2 & 2 & 3 & 4 & 5 & 5 \\ 2 & 2 & 2 & 3 & 4 & 5 & 5 & 5 \\ 2 & 2 & 3 & 4 & 5 & 5 & 5 & 5 \end{pmatrix} \tag{4.2}$$

$$\begin{pmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 4 \\ 3 & 3 & 3 & 3 & 3 & 3 & 4 & 5 \\ 3 & 3 & 3 & 3 & 3 & 4 & 5 & 6 \\ 3 & 3 & 3 & 3 & 4 & 5 & 6 & 7 \\ 2 & 3 & 3 & 4 & 5 & 6 & 7 & 8 \\ 3 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 4 & 5 & 6 & 7 & 8 & 9 & 9 \\ 4 & 5 & 6 & 7 & 8 & 9 & 9 & 9 \end{pmatrix} \tag{4.3}$$

The values in those quantization matrix represent the amount of bits that the corresponding transform coefficients should be shifted to the right to quantize the coefficient. Dequantization is achieved by left-shifting with the same amount of bits.
The deadzone width of the quantizer is 20: all transform coefficients that fall into the range $[-20, 20]$ are therefore mapped to zero. The amount of P-frames after



(a) Original part of the scene (b) High quality encoded part of the scene

(c) Acceptable quality encoded part of the scene

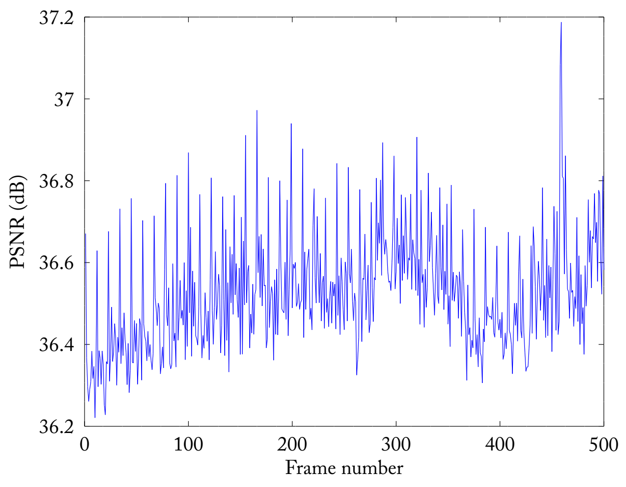
Figure 4.1: Selection of a frame - Encoded with different qualities

which an I-frame must be inserted (see section 3.4) was set to 10.

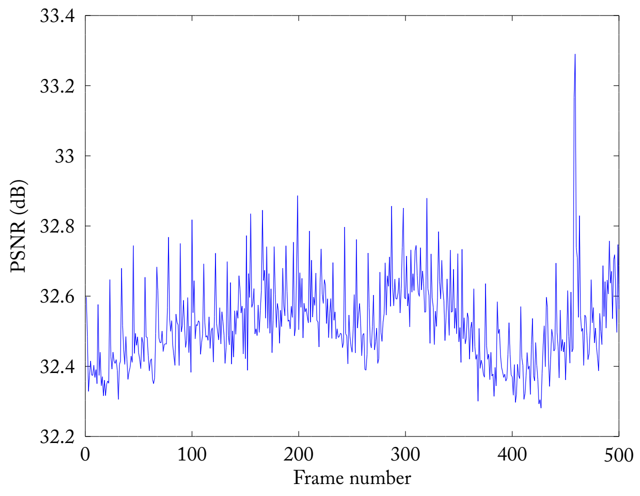
The laptop running the codec was a dell xps 13 including 8GB of RAM and an i5-5200 CPU.

4.2 PSNR

The PSNR was calculated for every frame by using the decoded output from our video codec and the original corresponding frame.

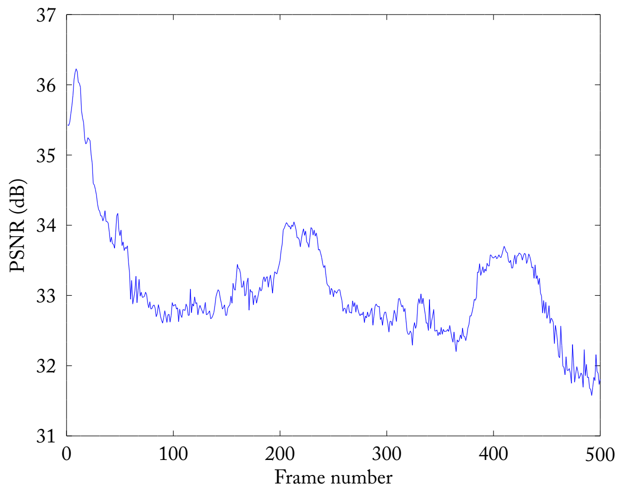


(a) `Old Town` - high quality

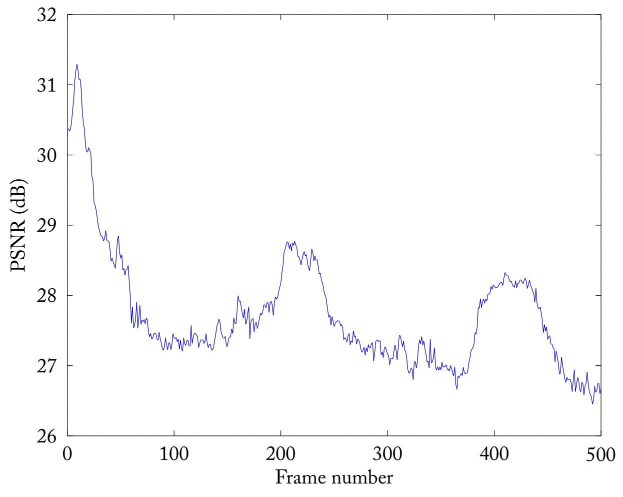


(b) `Old Town` - acceptable quality

Figure 4.2: PSNR of acceptable and high-quality encoding `Old Town`



(a) `ParkJoy` - high quality

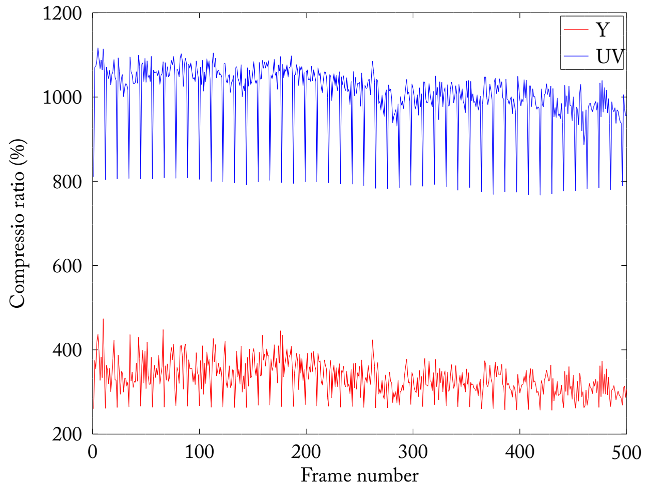


(b) `ParkJoy` - acceptable quality

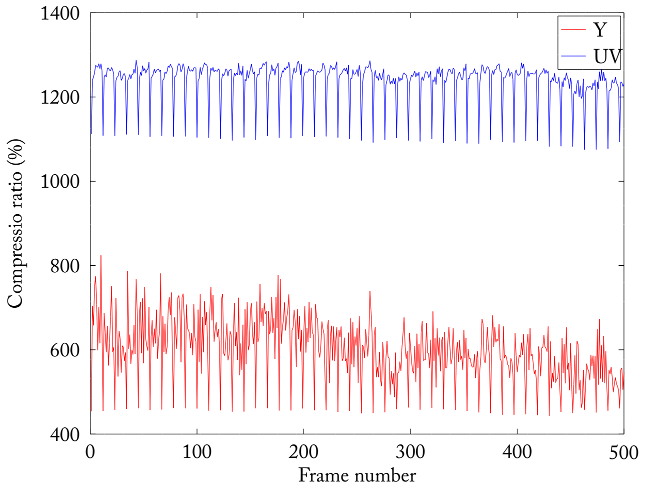
Figure 4.3: PSNR of acceptable and high-quality encoding `ParkJoy`

4.3 Compression-ratio

The compression ratio was calculated for every frame according to eq. (4.1).

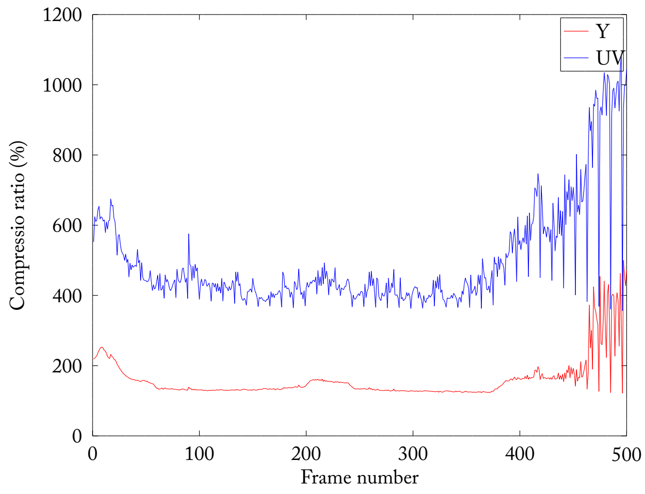


(a) `Old Town` - high quality

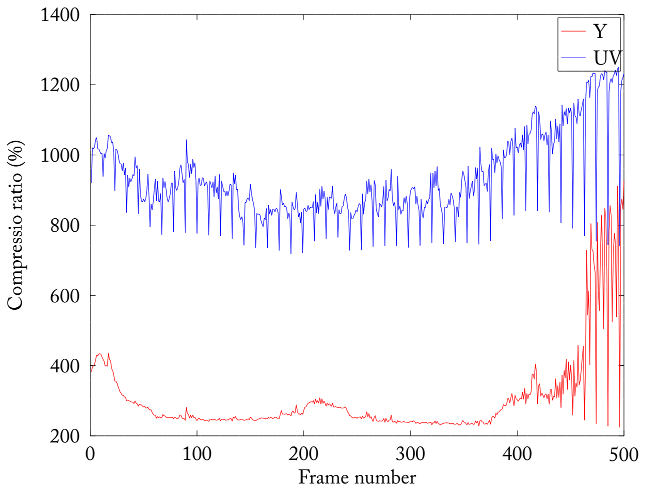


(b) `Old Town` - acceptable quality

Figure 4.4: Compression ratio of high-quality and acceptable encoding of `Old Town`



(a) `ParkJoy` - high quality



(b) `ParkJoy` - acceptable quality

Figure 4.5: Compression ratio of high-quality and acceptable encoding of `ParkJoy`

4.4 Speed

The speedresults are obtained by timing the encoder/decoder for a whole frame, during the 500 frames of both `ParkJoy` and `Old Town`

Encoder times

min (ms)	mean (ms)	max (ms)	total(ms)
14.44	18.04	22.80	9021.04

Decoder times

min (ms)	mean (ms)	max (ms)	total(ms)
8.14	10.97	15.62	5486.64

Table 4.1: Codec time statistics - `ParkJoy` - high quality

Encoder times

min (ms)	mean (ms)	max (ms)	total(ms)
12.61	16.89	28.22	8444.39

Decoder times

min (ms)	mean (ms)	max (ms)	total(ms)
7.10	9.89	17.74	4945.65

Table 4.2: Codec time statistics - `ParkJoy` - acceptable quality

Encoder times

min (ms)	mean (ms)	max (ms)	total(ms)
13.03	15.66	23.76	7830.69

Decoder times

min (ms)	mean (ms)	max (ms)	total(ms)
7.41	9.54	15.08	4772.48

Table 4.3: Codec time statistics - `old town` - high quality

Encoder times

min (ms)	mean (ms)	max (ms)	total(ms)
10.95	14.18	19.32	7091.39

Decoder times

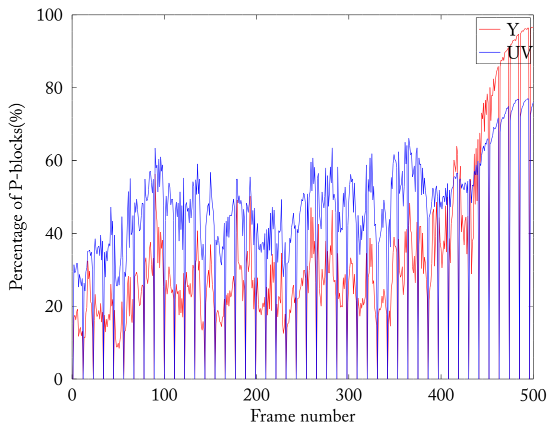
min (ms)	mean (ms)	max (ms)	total(ms)
6.72	8.10	14.28	4050.59

Table 4.4: Codec time statistics - `old town` - acceptable quality

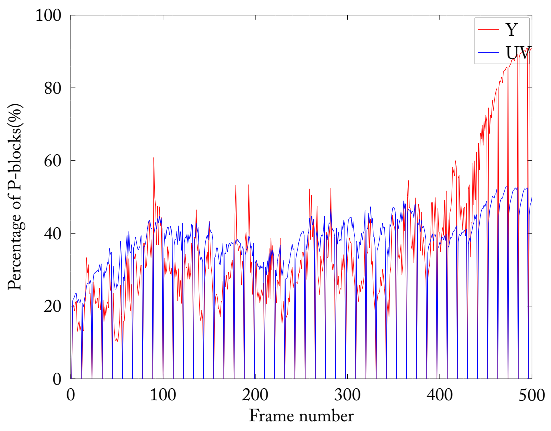
4.5 Percentage P-blocks

The percentage of P-blocks was calculated for every P-block using

$$\text{Percentage}_{P-blocks} = 100\% \frac{\text{\#P-blocks}}{\text{\#blocks in a frame}} \tag{4.4}$$

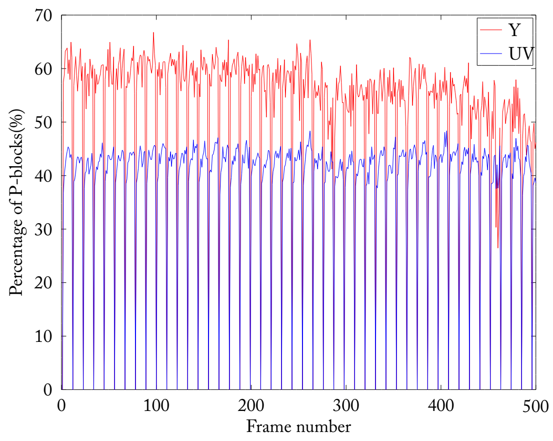


(a) `ParkJoy` - high quality

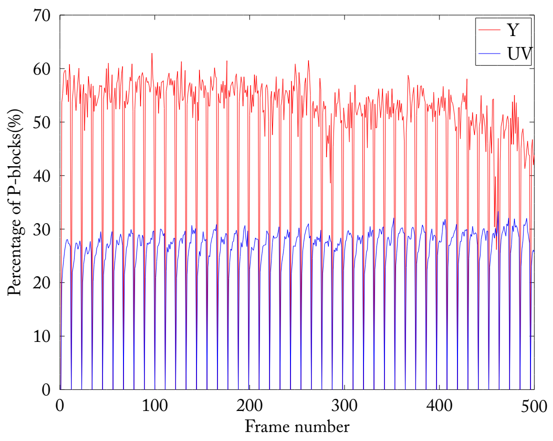


(b) `ParkJoy` - acceptable quality

Figure 4.6: Percentage of P-blocks per frame of high-quality and acceptable encoding of `ParkJoy`



(a) `Old Town` - high quality



(b) `Old Town` - acceptable quality

Figure 4.7: Percentage of P-blocks per frame of high-quality and acceptable encoding of `Old Town`

Chapter 5

Discussion

In this chapter we will reflect on the results as presented in ??

5.1 Compression ratio

If we look at fig. 4.4 we observe that the compression ratio of the UV planes is substantially higher than that of the Y plane. This is also present in fig. 4.5. This is most likely a result of (even subsampled) the chroma plane having lower frequency content than the luma plane [28].
We furthermore observe that after every 10 frames, there is a substantial drop in the compression ratio. This is result of insertion of an I-frame after 10 frames. This effect is less visible in fig. 4.5. Recall that this scene is considered difficult. The more-or-less constant compression ratio of the Y-plane can be explained due to the relatively quick movement in the scene: quick movement causes the difference between to subsequent frames to be large enough to make the encoder decide to use an I-block instead of a P-block. Note that from frame 450 onwards we observe a significant increase in compression ratio in fig. 4.5. This corresponds to the part of the scene in which the movement stops, hence P-blocks become more attractive for the encoder. This behaviour is confirmed by fig. 4.6, in which we see an increasing percentage of P-blocks present in the encoded frames.
Interesting is that when we switch to the acceptable quality matrix, the percentage of P-blocks decreases in both fig. 4.6 and fig. 4.7. This is not surprising, as coarser quantization results in a lower PSNR: hence when the encoder takes the difference of the reference frame (which is the decompressed version of the previous encoded frame) with the current *original* frame of the video, the difference is likely to be higher, than if both frames were taken from the original video.
Furthermore the percentage of P-blocks in the Y and UV plane seems to be strongly dependent on the scene to be encoded. fig. 4.6 shows an on average slightly higer percentage of P-blocks in the UV plane, whereas the percentage of P-blocks in fig. 4.7 is higher for the YV plane.

5.2 Quality

Referring to fig. 4.2 we observe a similar pattern as in the compression ratio: after every ten frames we observe a relatively small increase in the PSNR. This can be explained due to the use of a deadzone quantizer: since P-blocks are usually employed when the the coefficients are relatively small, this will result in transform coefficients that are relatively small as well. A deadzone quantizer will remove most, if not all of the coefficients if they are small enough.
Observe that this effect in fig. 4.3 is far less noticable, which can again be explained by `ParkJoy` containing a lot of movement, making the encoder encode frames using primarily I-blocks for the Y-plane(see also fig. 4.6).
Furthermore observe that in both figs. 4.2 and 4.3 switching from the high-quality quantization matrix to the acceptable quantization matrix results in a drop of the PSNR of approximately 4dB. At the same time, artefacts within macroblocks start to become significant. This can be seen from fig. 4.1, which shows part of a frame encoded&decoded with the acceptable quality matrix. Do note the difference with downsampling: although we see blocks, a block still contains detail.

5.3 Speed

We observe from section 4.4 that the encoder takes around 1.5x more time. This is due to the encoder having a higher complexity than the decoder (it features a partial decoder as well, including an inverse transform & dequantizer in the feedback loop, adding extra complexity (see fig. 2.2).
Note that in order to achieve 30fps, one would need to encode/decode one frame in $\frac{1}{30} \approx 33.3$ms. Observe that in all cases, the maximum encoding and decoding time never exceeds the 33.3ms, hence the the decoding/encoding rate always higher than 30fps.

Chapter 6

Conclusion

We have observed that the compression ratio of videos encoded by our codec varies per scene, and varies between frames as well. We therefore fulfill the requirement of having a variable compression ratio. Important to note as well is that even though transform coding results in coefficients that use more bits than the untransformed coefficients, we obtain compression ratios as high as 700% for the Y'-plane and close to 1300% for the UV planes.
As a result of using a block-transform in our codec, we automatically fulfill the requirement of emulation of macro-block behaviour. Our results also confirm that our codec exhibits the coding artefacts associated with macroblocking (??).
Furthermore, our results in section 4.2 confirm that the obtained quality is acceptable (> 20dB) and that our encoder and decoder manage to run at 30 fps seperately. Hence we can claim fulfillment of acceptable quality of a 720p at 30 fps.
As of now, the codec however does not support on the fly resolution changing, nor does it support stateless codec emulation. It has been decided not to include this into the codec yet, due to the limited-time available. Therefore, all except these requirements as stated in chapter 1 are met.

6.1 Future Work

1. Treat the first walsh-hadamard coefficient differently from the others. As [20] points out, the range of the other coefficients is only halve that of the first coefficient (hence we could save a bit per encoded coefficient).
2. Integrate the codec into the linux kernel: at the time of writing this report the codec is not yet integrated into the Linux kernel. However, this involves a few minor adjustments to the code and adding support for
   (a) State- and stateless-codecs.
   (b) On-the-fly resolution changing.

Which should not be very time-consuming tasks. As of the moment that this report was written, the author has already planned on doing so, outside the scope of this project.

Bibliography

[1] Wikipedia, "Video codec — wikipedia, the free encyclopedia," 2016. [Online; accessed 14-October-2016].

[2] E. Reinhard, E. A. Khan, A. O. Akyuz, and G. Johnson, *Color imaging: fundamentals and applications*. CRC Press, 2008.

[3] A. Ford and A. Roberts, "Colour space conversions," *Westminster University, London*, vol. 1998, pp. 1–31, 1998.

[4] D. R. Bull, *Communicating pictures: A course in Image and Video Coding*. Academic Press, 2014.

[5] I. E. Richardson, *H. 264 and MPEG-4 video compression: video coding for next-generation multimedia*. John Wiley & Sons, 2004.

[6] A. C. Bovik, *Handbook of image and video processing*. Academic press, 2010.

[7] C. Poynton, *Digital video and HD: Algorithms and Interfaces*. Elsevier, 2012.

[8] M. C. Rost, *Data compression using adaptive transform coding*. PhD thesis, NASA, 1988.

[9] L. Torres and M. Kunt, *Video coding: the second generation approach*. Springer Science & Business Media, 2012.

[10] J. F. Blinn, "What's that deal with the dct?," *IEEE Computer Graphics and Applications*, vol. 13, pp. 78–83, July 1993.

[11] I. E. Richardson, *Video codec design: developing image and video compression systems*. John Wiley & Sons, 2002.

[12] M. Ghanbari, *Standard codecs: Image compression to advanced video coding*. No. 49, Iet, 2003.

[13] X. Foundation, "Theora specification," Mar. 2011.

[14] I. E. T. Force, "Thor: High efficiency, moderate complexity video codec using only rf ipr," July 2015.

[15] J. Bankoski, P. Wilkins, and Y. Xu, "Technical overview of vp8, an open source video codec for the web.,"

[16] B. G. Haskell and A. N. Netravali, *Digital pictures: representation, compression, and standards*. Perseus Publishing, 1997.

[17] R. C. Gonzalez and R. E. Woods, "Digital image processing publishing house of electronics industry," *Beijing, China*, p. 262, 2002.

[18] K. R. Rao and P. C. Yip, *The transform and data compression handbook*, vol. 1. CRC press, 2000.

[19] R. D. Brown, "A recursive algorithm for sequency-ordered fast walsh transforms," *IEEE Transactions on Computers*, vol. C-26, pp. 819–822, Aug 1977.

[20] W. K. Pratt, J. Kane, and H. C. Andrews, "Hadamard transform image coding," *Proceedings of the IEEE*, vol. 57, pp. 58–68, Jan 1969.

[21] J.-M. Valin and T. B. Terriberry, "Perceptual vector quantization for video coding," in *SPIE/IS&T Electronic Imaging*, pp. 941009–941009, International Society for Optics and Photonics, 2015.

[22] A. M. Bock, *Video Compression Systems: From first principles to concatenated codecs*. IET Digital Library, 2009.

[23] I. Chakrabarti, K. N. S. Batta, and S. K. Chatterjee, *Motion Estimation for Video Coding*. Springer, 2015.

[24] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 24–35, ACM, 2013.

[25] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education India, 2006.

[26] P. Kube, "Huffman's algorithm pseudocode." Accessed: 2016-11-03.

[27] G. G. Langdon, "An introduction to arithmetic coding," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 135–149, 1984.

[28] B. Waggoner, *Compression for great video and audio: master tips and common sense*. Taylor & Francis, 2010.

[29] A. Shahbahrami, R. Bahrampour, M. S. Rostami, and M. A. Mobarhan, "Evaluation of huffman and arithmetic algorithms for multimedia compression standards," *arXiv preprint arXiv:1109.0216*, 2011.

[30] Z. Wang, H. R. Sheikh, and A. C. Bovik, "Objective video quality assessment," *The handbook of video databases: design and applications*, pp. 1041–1078, 2003.

[31] A. Kondoz, *Visual media coding and transmission*. John Wiley & Sons, 2009.

[32] "35 u.s. code § 154 - contents and term of patent; provisional rights." Accessed: 06-11-16.

[33] J. Manz, "A sequency-ordered fast walsh transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 20, no. 3, pp. 204–205, 1972.

[34] H. Larsen, "An algorithm to compute the sequency ordered walsh transform," 1976.

[35] L. Haglund, "The svt high definition multi format test set," *Swedish Television Stockholm*, 2006.