# Interactible Video Compression with Motion Compensation

Daniel Borgehammar

Department of Electrical and Information Technology
Lund University

Advisor: Irina Bocharova

November 26, 2011

Printed in Sweden
E-huset, Lund, 2011

# Popular Summary

There is a lack of suitable compression schemes for remote control applications. Current applications make use of still image compression with a compression too low to work well on cellular networks. Better compression can be achieved with a suitable video compression scheme, but current standards are not designed for interactivity.

The importance of supporting user interactivity can be illustrated as follows. A mobile phone with a small screen is used to control a home PC. Since the screen of the mobile phone is small the user needs to move the view around a lot to see different parts of the home PC's screen. This is where user interactivity comes into play. For existing video compression there would be a delay between the user trying to move the view and the view actually moving. With a poor connection the user might have to move the view, wait for it to update, then move, wait and move again to find what she is looking for.

The thesis describes a video compression scheme which allows the view to be moved instantly, without delay. To combat poor connections, the scheme supports progressive updating of the image, that is, allowing a very low quality image to

be sent quickly and improving it over time. Additionally, it supports updating areas not currently in view. This makes it possible to have a low quality image of the whole PC screen, good enough to navigate with, already prepared when the user moves the view. Then, when the user stops moving the view, the part of the screen that is then visible quickly improves to full quality.

A requirement for the scheme to work is efficient motion compensation. The thesis outlines two methods of adapting existing techniques to this end:

**(1)** Block based motion compensation

**(2)** Simple translational global motion compensation

The two methods can be combined and testing indicates that this will result in significantly improved compression as compared to existing remote control applications.

# Abstract

There exist various types of remote control software for mobile devices and smart phones. One problem they have in common is the lack of good image data compression. Typically, image compression standards such as JPEG and PNG are used to compress the image data, it works, but gives very low frame rates. Existing video compression standards are not used for remote control software, likely, in large part, because implementations are too specialized. It is easy enough grabbing image data from part of the screen and moving around the area you capture image data from is possible, but when you try to do other things like zooming in and out you start to run into problems.

The goal of this thesis is to develop a good method for image data compression, targeted specifically towards remote control applications. More specifically, this thesis explores the use of wavelet transformation to compress the image data more efficiently than the commonly used JPEG and PNG standards. In addition, two wavelet based video compression methods are proposed to improve the compression further, while giving the flexibility remote control software requires. The video compression methods shows very competitive performance when compared to H.264.

# Acknowledgment

This work has been carried out at the Department of Electrical and Information Technology at Lund University. I would like to express my gratitude towards my supervisor, Irina Bocharova. Without her guidance this work would not be what it is today.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1
# Introduction

The goal of this master thesis was to develop an efficient compression algorithm for video with special demands, the primary target for the compression scheme being remote control software where a user requires immediate response to sent commands. One example of such a program could be a program for remote controlling a computer using a mobile phone, acting as keyboard, mouse and screen. Another example would be a wireless remote control for some form of robot or vehicle with built-in camera.

The main requirement was that the compression method must allow streaming video with a delay that is hardly noticeable by a human. As a consequence the calculation complexity had to be kept to a minimum since any time spent on encoding and decoding adds to the overall delay.

While current video compression standards can achieve fairly good compression and high frame rates while streaming, the compression in some scenarios are far from optimal. Examples of typically poorly handled situations are translational camera movements over a static background and rendering of a zoomed-out view.

**Requirements** The following is a list of requirements imposed on the compression scheme based on the expected needs of real life implementations. The requirements should not be viewed as features to be included but rather as restrictions. For instance, the first requirement implies that there is little to no buffering of new video frames before they are transmitted.

1. Allow streaming video with a delay that is hardly noticeable by a human
2. Low computational complexity for both encoding and decoding
3. Continually changeable compression level
4. Support for progressive updating of the image
5. Efficient handling of different zoom levels

# Chapter 2
# Video Compression Overview

When talking about compression it is possible to use different definitions. Throughout this report, *compression* simply refers to the ratio *uncompressed / compressed*. So when it is stated that the compression of an image is 10, it simply means that the uncompressed image is 10 times as large as the compressed one.

There are two types of compression, lossless and lossy. While lossless compression gives the highest possible quality, it may be difficult to obtain even a compression of 2. Using lossy compression, much higher compression ratios can be achieved, but at the cost of reduced quality. However, even at compression ratios of 10-20 it can be very difficult to see the difference between the compressed and the original image.

Compression is achieved by removing redundant information from the image. The more apparent redundancies seen in images are repeated shapes and colors. A straight line, for instance, could be represented with the color and a count rather than with the color value repeated for every pixel. More subtle redundancies can take the form of predictable variations, such as the color of a line shifting linearly from black to white over $X$ pixels. Any additional information over the bare minimum required to recreate the original image perfectly is considered redundant. Most existing systems are based on spatial transforms.

While image compression is just concerned with spatial redundancies, video compression also needs to deal with the redundancies between images, often referred to as temporal redundancies. In a video sequence there are typically not a lot of changes between each frame, meaning there is potential for a large compression gain if the redundancies can be removed. The temporal redundancies are generally dealt with using different forms of motion compensation.

Some of the more common standards currently used for lossy and lossless image compression are JPEG and PNG respectively. A more recent but less widely used standard is JPEG 2000 which allows both lossless and lossy compression. For many images JPEG 2000 gives better lossless compression than PNG and higher quality than JPEG at the same compression ratio. JPEG 2000 has still not been widely adopted and is not supported by nearly as much software as JPEG and PNG currently are.

The MPEG-4 standard covers the coding of audio and video and consists of different *Parts*. One of the parts covers the file format .mp4, which is a container for audio and video data, playable by many modern video players. For this thesis however, the main interest is in Part 10, also known as H.264. It is one of the

more commonly used standards for video compression. The proposed compression method in this thesis will be compared both to H.264 as well as comparing still image compression with the above mentioned standards.

An example of a typical video compression scheme can be seen in Figure 2.1.

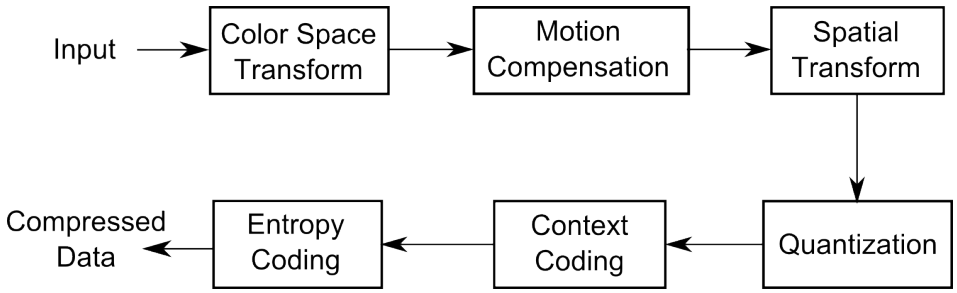


**Figure 2.1:** A typical video compression scheme

## 2.1 Color space transforms

Color images are typically stored as three or four separate color components. Depending on the color space chosen there can be a lot of correlations between the components. A good choice of target color space for lossy compression should not only reduce the correlations between the color components but also concentrate important information to one component. This allows more lossy compression of the remaining components while keeping the quality loss to a minimum.

The standard representation for color images in computers can be considered as using an RGB color space. Although far from all images are represented in RGB, it is practically always possible to directly convert the image to RGB. In this work it's assumed the original image is always represented in RGB.

Although there are many target color spaces that could be considered, only luminance-chrominance based color spaces were considered in this work. It is well known that luminance-chrominance color spaces are well suited to image compression and it was not deemed to be worth the effort to explore alternatives that *might* give slightly better results. The luminance component is a combination of R, G and B, representing the brightness in the image. The chrominance components hold information about color in the image and are typically the blue-difference and red-difference.

## 2.2 Spatial Transforms

When considering different spatial transforms there are a few important properties to consider [1]. First and foremost, only linear transforms will be considered. In addition the transform should have the following properties:

- transform should be invertible, that is, the transform itself should not introduce any errors

- transform should be orthonormal, this preserves the size of the mean squared error (MSE) before and after the transformation. This is important since otherwise even small errors introduced by the quantization step could become very large
- low computational complexity, the transformation should be separable into two one-dimensional transforms
- the correlations should be reduced, a good transform will have little to no correlations between the transform coefficients
- the energy/information should be concentrated to a small number of transform coefficients, this allows the least significant information to be removed first

Some more details of these properties will be given below. If we have an input column vector $\mathbf{x}$ of length $N$ and a transformation matrix $T$ of size $N \times N$ the linear transform can be expressed as

$$\mathbf{y} = T\mathbf{x}$$

and the invertible property for a linear transform as

$$T * T^{-1} = I$$

where $I$ is the identity matrix. Finally the orthonormal property (for a real matrix) can be expressed as

$$T^{-1} = T^T$$

To transform a 2D-input signal $X$ of size $N \times N$ we can in the general case do the following. First the input signal $X$ is rearranged as a vector $\mathbf{x}$ with length $N^2$ and then it is transformed by a transform matrix $T_1$ of size $N^2 \times N^2$. This gives an output vector $\mathbf{y}$ with length $N^2$ which can be rearranged to the output matrix $Y$ of size $N \times N$.

$$\mathbf{y} = T_1\mathbf{x}$$

Applying the transform in this way requires $N^4$ multiplications. A more efficient approach is possible, however, if the transform is separable into two 1D transforms. In this case there exists a transform matrix $T_2$ of size $N \times N$ that can be applied in the following way, giving the exact same result as above

$$Y = T_2 X T_2^T$$

In this case the number of multiplications is reduced to $2N^3$.

When considering the fourth point, it should be mentioned that what is meant by correlation in this thesis is simply the linear dependence. Thus even if all correlations are removed there may still be nonlinear dependencies left. The nonlinear dependencies will for the most part be ignored in this thesis; the expected compression gain compared to computational effort is simply too low.

Let $\mathrm{E}[X] = \mu$ be the expected value of the random variable $X$, then the standard deviation of $X$ can be expressed as

$$\sigma = \sqrt{\mathrm{E}[(X - \mu)^2]}$$

Using these definitions and considering two transform coefficients as random variables $X$ and $Y$, the correlation between these two coefficients can be expressed as

$$\rho_{x,y} = \frac{\mathrm{E}[(X-\mu_x)(Y-\mu_y)]}{\sigma_x \sigma_y}$$

The information compacting property can be seen as follows. Take a vector $\mathbf{y}$ consisting of transformation coefficients sorted in decreasing importance. As increasingly many of the last coefficients are zeroed, the MSE of the reconstructed vector $\mathbf{x}$ should increase as slowly as possible.

There exists a transform called the Karhunen-Loeve transform [1] that both optimally compacts the information as well as removes all correlations. However, the basis functions depend on the input vector. What this means is that not only the transform coefficients, but also the basis vectors are needed for reconstruction. This extra information will generally far outweigh the compression gains for the transform coefficients gained compared to other transforms. Better choices are the discrete cosine transform (DCT) and discrete wavelet transform (DWT), both of which can be applied independently of the input. While neither transform generally removes all correlations and the information compacting isn't perfect, they both perform well and fulfill the other criteria. Some additional properties that each has will be described below.

### 2.2.1 Discrete Cosine Transform

The DCT [2,3] is a Fourier related transform and is similar to the discrete Fourier transform. The idea for both transforms is the same, to obtain a frequency representation of the input signal, the low frequencies giving the general structure and the higher frequencies adding details. A good approximation of the original signal can often be obtained from just the lower frequencies making this representation very suitable for lossy compression. There are a few reasons why the DCT is more suitable to image compression than the more general Fourier transform. Unlike the Fourier transform the DCT only operates on real numbers, which allows some simplifications. The energy compaction is also better, in particular for the DCT variant called DCT-II which uses even boundary conditions. The DCT-II (scaled to obtain an orthonormal basis) is defined as

$$y_k = C_k \sum_{n=0}^{N-1} x_k \cos\frac{(n+1/2)k\pi}{N} \text{ with } k = 0, 1, ..., N-1 \text{ and } \begin{array}{l} C_k = \sqrt{\frac{1}{N}}(k=0) \\ C_k = \sqrt{\frac{2}{N}}(k>0) \end{array}$$

This equation can also be written in matrix form as

$$A_{kn} = C_k \cos\frac{(n+1/2)k\pi}{N}$$

Given an input $\mathbf{x}$ as a column vector the 1D-transform becomes $\mathbf{y} = \mathbf{A}\mathbf{x}$ and given an input matrix $X$ the 2D-transform becomes $\mathbf{Y} = \mathbf{A}\mathbf{X}\mathbf{A}^{\mathrm{T}}$. Thanks to the orthogonality the inverses can simply be written as $\mathbf{x} = \mathbf{A}^{\mathrm{T}}\mathbf{y}$ and $\mathbf{X} = \mathbf{A}^{\mathrm{T}}\mathbf{Y}\mathbf{A}$ respectively.

What the transform gives is the amplitude of each frequency that the signal is made up of. From a compression point of view this is very beneficial when the amplitude for several frequencies are close to zero. By removing those frequencies a lot of compression can be achieved with minimal loss in quality.

As mentioned previously, applying the transform as two 1D-transforms requires $2N^3$ multiplications. Fast DCT algorithms exist which reduces the computational complexity, in particular for the cases where $N$ is a power of two. Some of the fastest algorithms have a complexity of $\frac{3}{2}N\log_2(N) - N + 1$ additions and $\frac{N}{2}\log_2(N)$ multiplications [4][1]. Even with the fastest algorithms it becomes too computationally expensive to apply the DCT to an entire image. Typically a image is divided into square blocks, generally with $N = 4$ or 8. It is possible to approximate the DCT using integer transforms. The decorrelating properties will generally be somewhat worse, but the computational complexity can be reduced and roundoff errors can be avoided, which otherwise is a issue when applying the DCT using limited precision floating point arithmetic.

**Example:** For $N = 4$ the transformation matrix A is

$$\mathbf{A} = \begin{bmatrix} \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) & \frac{1}{2}\cos(0) \\ \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{5\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{7\pi}{8}\right) \\ \sqrt{\frac{1}{2}}\cos\left(\frac{2\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{6\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{10\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{14\pi}{8}\right) \\ \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{9\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{15\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{21\pi}{8}\right) \end{bmatrix}$$

The cosine function is symmetrical and repeats after $2\pi$ radians; after simplification **A** becomes

$$\mathbf{A} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) \end{bmatrix}$$

or

$$\mathbf{A} = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} \text{ where } \begin{aligned} a &= \frac{1}{2} \\ b &= \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) \\ c &= \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) \end{aligned}$$

[1] The paper appears to have mistakenly stated the formula for additions as being the formula for multiplications and vice versa, this has been corrected here.

The transformation matrix in the example above can be further simplified. By doing some approximations a DCT-like transformation can be obtained which only requires integer arithmetic. The approximations used in H.264 for the 4x4 matrix is $a = 1$, $b = 2$ and $c = 1$ giving

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}$$

This transformation matrix is no longer orthogonal and to achieve correct results rescaling is needed after the transformation. The rescaling can essentially be done for free by combining the rescaling with the quantization step.

### 2.2.2 Discrete Wavelet Transform

The discrete wavelet transform (DWT) [5] works in a similar fashion as the DCT, but there are some important differences. The DCT uses the cosine as the basis function and the transform gives the amplitudes of the cosine at different frequencies; when summing up the cosines the original signal is obtained. Rather than using a periodic basis function, the DWT uses basis functions obtained by scaling and shifting one short wave-like function called *mother wavelet*, see Figure 2.2.

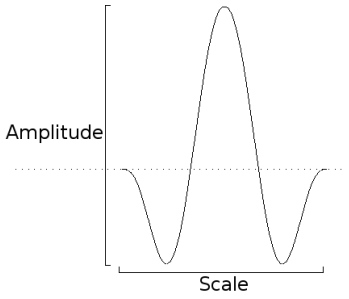


**Figure 2.2:** A wavelet basis function

Instead of representing the signal as a sum of the weighted basis functions at different frequencies, the transform gives the amplitude of the basis function at different points in time and at different scales, see Figure 2.3.

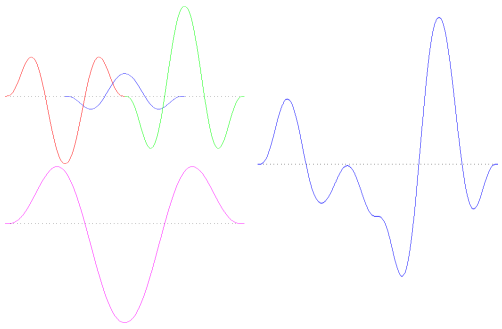

**Figure 2.3:** At the top left there are 3 wavelets at different points in time and different amplitudes but having the same scale, at the bottom left there is one wavelet having a larger scale. To the right is the sum of the wavelets.

The wavelet transform can be interpreted as a decomposition of the original signal over a set of bases with different time-frequency resolutions. This gives good localization of energy, both for long harmonic signals and short signals with sharp changes in time.

One advantage over the DCT is that a good resolution in time is obtained, another is the ability to choose the base wavelet functions. The latter is difficult to take advantage of, but it is potentially possible to analyze an image and pick the most suitable wavelet function for that particular image. In this thesis no attempt is made to find the optimal function for particular images. Instead, only one wavelet function is used which can be implemented very efficiently and gives good compression for a wide array of images, but not optimal for any particular image.

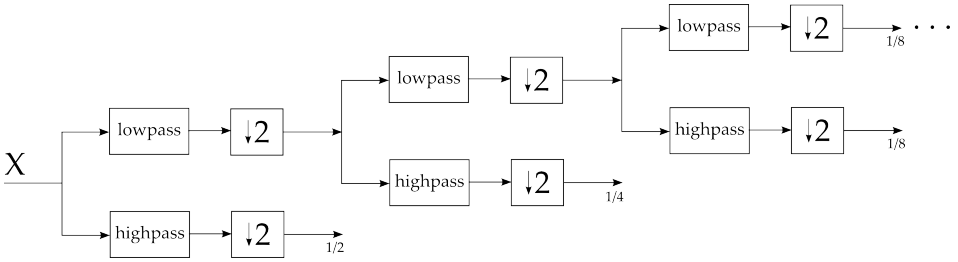


**Figure 2.4:** Hierarchical set of low- and high-pass filters. The $\downarrow 2$ symbolizes decimation. After each set of high- and low-pass filters half of the remaining components are returned.

The implementation of the DWT is most easily described using digital filters. The transform can be described as a set of hierarchical high-pass and low-pass filters [1]. The input is first copied and sent to both a high-pass filter and a low-pass filter. After filtering both parts are decimated; that is, only the even-numbered samples are kept. The exact same procedure is repeated for the decimated low-frequency part while the high-frequency part after each step represents the final output. See Figure 2.4. The decimation is done to remove the redundant information introduced when the input is duplicated. While not immediately clear, no information is lost and the original input can be perfectly reconstructed.

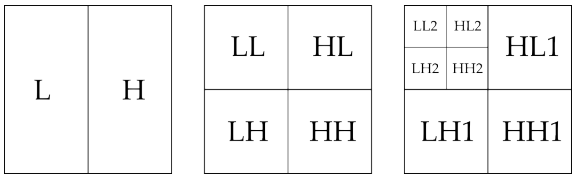


**Figure 2.5:** The wavelet filtering is applied in several steps. First vertically (left), leaving the low and high frequency components in the left and right half respectively. In the middle the transform has been applied both horizontally and vertically and to the right the transform has been applied again to the LL part.

Both the high-pass and the low-pass filters are applied to the entire image. The high-pass filter only keeps the high frequency components, which are generally quite sparse. The low-pass filter keeps the low frequency components, where most of the important information is generally contained. For an image the filtering is applied twice, once horizontally and once vertically, see Figure 2.5.

Once the entire image has been filtered once, the LL part is filtered again the same way. This process is repeated until the LL part is small (anywhere from 1x1 to a few hundred pixels). Since the compression gain becomes lower for each filter stage it is often best to stop the iteration early.

Any discrete-time filter can be determined in time domain by its pulse response h(n). For the so-called filters with finite pulse(impulse) response (FIR filters) h(n) is a finite sequence of filter coefficients. For example, in this thesis two wavelet filters are used given by $h(n) = [-1\ 2\ 6\ 2\ -1]/8$ and $g(n) = [-2\ 4\ -2]/8$, a low-pass and high-pass filter respectively. The same filters can be determined in the z-transform domain by its transfer function

$$H(z) = \sum_{n=0}^{\infty} h(n) z^{-n}$$

where z is a complex variable. The filters h(n) and g(n) rewritten in this form:

$$H(z) = -\frac{1}{8}z^{-2} + \frac{1}{4}z^{-1} + \frac{3}{4} + \frac{1}{4}z - \frac{1}{8}z^{2}$$
$$G(z) = \frac{1}{4}z^{-2} - \frac{1}{2}z^{-1} + \frac{1}{4}$$

For this thesis the only important thing to understand about the z-transform is that $z^x$ symbolizes a delay where $x$ is the amount of the delay. Applying the filters to some input $X$ can be done through convolution. The result of the filtering can be seen in Figure 2.6 where the above filters have been applied to the left image, first on each row and then on each column, producing the middle image. Applying the filters again on the upper left quarter of the middle image produces the rightmost image.

**Figure 2.6:** The original is to the left. The middle and right image have been transformed by one and two 2D filterings respectively.

## 2.3 Motion Compensation

Motion compensation [3] is a way to deal with the temporal redundancies. The main idea is to predict the displacement of pixels from their position in a previous frame. The displaced pixels in the current frame can then be subtracted from the pixels in the previous. The obtained residual will contain less information if the match is good, allowing the residual to be compressed a lot more. For the residual to be of any use, the information about the displacement is also needed. This additional information is represented by motion vectors. For motion compensation to give any improvement in compression the information for the motion vectors needs to be less than the gain from compressing the residual. This is one of the main reasons why motion compensation is usually performed on blocks of pixels: it is a simple scheme that allows one motion vector to represent the displacement of many pixels.

One of the simplest approaches is to split the new image into rectangular blocks of fixed size and for each of these blocks do a linear search for the best match in the previous frame. Different matching criteria can be used when determining the best match. A popular criterion is the sum of the squared difference

$$\min_{\alpha,\beta} \left\{ \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} [x_c(m,n) - x_p(m+\alpha, n+\beta)]^2 \right\}$$

where $x_c(m,n)$ and $x_p(m,n)$ are pixels of the current block and of the co-sited block of the previous frame, $\alpha$ and $\beta$ are shifts of the pixels along the coordinate axes. The motion vectors are vectors of the shifts $\alpha$ and $\beta$, that is, $A = (\alpha_1, ..., \alpha_m)$ and $B = (\beta_1, ..., \beta_n)$.


**Figure 2.7:** Block matching is performed on the red square in the new image. Its corresponding location in the previous frame is also marked in red. A search for best match is performed in the dark region and the best match is marked by a green square.

Another popular criterion is taking the sum of the absolute values instead of the squared values. An example of this block matching approach is shown in Figure 2.7 the new image has been divided into a number of same-sized blocks. Matching

is performed on the block marked in red in the new image, its location in the previous image is marked in red as well. The dark region is searched for a match and the best match in the previous frame has been marked in green.

When dealing with lossy compression it is important to avoid accumulating errors from compressing several successive frames. A common approach [6] is to reconstruct the compressed image at the sender in order to get an image with the exact same errors as the receiver will have. Then the reconstructed image is used as a base when constructing the next residual to be sent, thereby correcting the errors introduced by the lossy compression of the previous residual. An illustration of how the motion compensation fits into the overall compression scheme is illustrated in Figure 2.8.

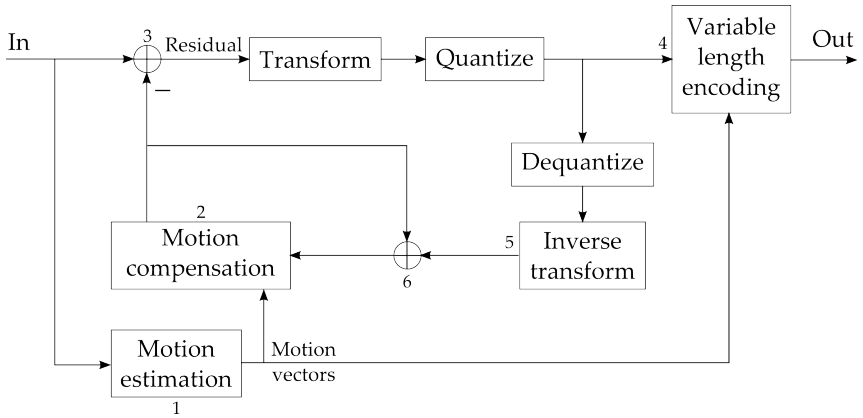


**Figure 2.8:** Overview over the video encoder and how the motion compensation fits into the overall compression scheme.

1. Motion estimation is performed on the new image, producing motion vectors (motion vectors are zero for the first image, since there is no previous image to compare with)
2. Motion compensation is performed on a local copy of the previous image, identical to the image the receiver has, including any compression artefacts (for the first image, no previous image has been sent or stored and the output is zero)
3. Output from motion compensation is subtracted from the new image
4. Residual is transformed and quantized and then compressed using variable length encoding
5. Residual is also dequantized and reconstructed to its original state, but with the same quantization errors as the receiver will have
6. Reconstructed residual is added to the motion compensated image used to construct the previous residual, thereby creating a local copy of the same image the receiver gets, including any compression artefacts

## 2.4 Quantization

This is the only step which removes information. By removing information an arbitrarily high compression rate can be achieved, but naturally at the cost of reduced quality. The simplest form of quantization is uniform scalar quantization. It can be implemented simply as a division and rounding operation. Let $x$ be the input and $\delta$ be the quantization step. The quantized value $y$ is then given by

$$y = \left\lfloor \frac{x}{\delta} \right\rceil$$

and the original value can then be approximated as

$$\hat{x} = y \cdot \delta$$

One modification that can be made is to extend the zero zone. That is, let all values $x$ below a set threshold be quantized to zero. This can be beneficial when quantizing the higher frequencies of the DCT and DWT. Small values centered around zero in these transforms will give very minor contributions to the image.

There are more sophisticated quantization methods that in general give lower quantization errors. However, for a source that has very low correlations, which will be the case after the spatial transformation, the reduction in quantization errors can be expected to be very minor. For this reason, other more computationally complex methods were never considered.

## 2.5 Entropy Coding

The final step in the compression scheme is to represent the information in a more compact form by applying entropy coding [7]. Entropy is a measure of the best possible lossless compression that can be achieved when treating data as a sequence of independent and identically-distributed random variables. The formula for measuring the best possible compression in bits

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

where $p(x_i)$ denotes the probability of $x_i$. When the data to be measured is known, $x_i$ can simply be calculated as the number of occurrences of this symbol divided by the total number of symbols.

The general idea is to spend few bits on symbols with high probabilities and more bits on symbols with low probabilities. Entropy coding can be applied symbol-by-symbol or generalized to be applied on blocks of input symbols. The symbol-by-symbol approach is simpler and allows faster implementations, but generally results in lower compression. The optimal compression for a symbol may be a fractional number of bits, but using symbol-by-symbol compression a whole number of bits (rounded up) needs to be spent to represent it. The problem is greatest when there are symbols with very high probabilities. Say the data to be

encoded contains 97% zeros, 2% ones and 1% twos. The number of bits needed on average to compress this data would be

$$H(X) = -[\, 0.98 \log_2(0.98) + 0.02 \log_2(0.02) + 0.01 \log_2(0.01) \,] \approx 0.222 \text{ bits}$$

Using a good block-based approach, an average 200-bit sequence could be compressed using as few as $200 * 0.222 \approx 44$ bits. An optimal symbol-by-symbol code on the other hand would need 1 bit per zero and 2 bits for each of the other two symbols. The coding table could look like the following:

| Symbol | Bit pattern |
|---|---|
| 0 | 0 |
| 1 | 10 |
| 2 | 11 |

The bits needed to encode an average 200-bit sequence with the above coding table would be

$$200 \times 0.97 \times 1 + 200 \times 0.02 \times 2 + 200 \times 0.01 \times 2 = 206 \text{ bits}$$

Notice that 1 cannot simply be mapped to 1 since when decoding, 11 could mean both a two or two ones. The above code is a prefix code, also sometimes called prefix-free code, which means that no code word is the prefix (start) of any other code word.

There are many entropy coding methods, but some of the most popular and likely suitable for the compression in this thesis are: Huffman coding, exponential Golomb coding and arithmetic coding.

### 2.5.1 Huffman coding

Huffman coding [8] is a popular symbol-by-symbol compression scheme and is optimal for symbol-by-symbol coding. The computational complexity is also very reasonable. A downside with Huffman coding is that the frequencies of the input data need to be known by both the encoder and decoder. A common solution is to create a table with the frequencies of all symbols which is then used when encoding and sent together with the encoded data to the decoder.

Huffman coding can be generalized to work on blocks of symbols. The basic idea is to group individual symbols together and represents the group with a new symbol. However, the complexity increases quickly with the block length and arithmetic coding is generally a better choice for block coding.

### 2.5.2 Exponential Golomb Coding

This is based on Golomb coding [9], and just like Golomb coding it is a prefix code which does not require knowledge about the probability distribution. Exponential Golomb coding encodes symbols using a fixed bit pattern that is determined according to a single parameter $k$. The following algorithm can be used to encode any non-negative integer:

1. $b = \lfloor X/2^k \rfloor + 1$
2. Write $\lceil \log_2(b+1) \rceil - 1$ zeroes as prefix followed by $b$ in binary
3. Write $X - 2^k \lfloor X/2^k \rfloor$ in binary as suffix

To encode negative values the algorithm needs to be modified slightly. An easy solution is to add an extra step before the encoding that changes the input in the following way: First take the absolute value of $X$, multiply by 2 and then subtract one if $X$ was negative. This makes all positive values even and all negative values odd, allowing for easy decoding. In the table below some outputs for $k = 0, 1, 2$ have been given as an example. As Table 2.1 hints at, with a small $k$, small values

| X | *k=0* | *k=1* | *k=2* |
|---|---|---|---|
| 0 | 1 | 10 | 100 |
| 1 | 010 | 11 | 101 |
| 2 | 011 | 0100 | 110 |
| 3 | 00100 | 0101 | 111 |
| 4 | 00101 | 0110 | 01000 |
| 5 | 00110 | 0111 | 01001 |
| 6 | 00111 | 001000 | 01010 |
| 7 | 0001000 | 001001 | 01011 |
| 8 | 0001001 | 001010 | 01100 |

**Table 2.1:** Bit patterns produced by exponential Golomb coding.

require few bits to encode while larger values quickly require a lot of bits to encode. For a larger $k$ small values requires more bits to encode, but the number of bits required to encode large values does not increase nearly as quickly.

**Example:** The sequence to be encoded is $\mathbf{X} = [0, 0, 2, 0]$ with k=0. Using Table 2.1 the encoded sequence becomes $[1, 1, 011, 1]$, or simply 110111.

### 2.5.3 Arithmetic coding

Arithmetic coding [10] encodes the entire input sequence as a single number, a fraction between zero and one. It effectively has a block length the size of the input. Arithmetic coding quickly approaches the entropy for the data source as the length of the input data increases, and can thus generally be viewed as optimal.

The idea of arithmetic coding is to divide the interval zero to one into subintervals according to the symbol's probability distribution. Each subinterval represents one symbol. Then each subinterval is again divided into subintervals, with each new subinterval representing a combination of two symbols. The new subintervals will represent all combinations of two symbols, with a size according to the probability of the pairs. To encode any message the interval zero to one can simply be subdivided as many times as there are symbols to encode. When the subinterval is found that represents the combination of symbols in the message, any number that falls within the subinterval can be picked as codeword. When decoding, the

same procedure can be repeated. The codeword is then used to pick the right subinterval and thus obtain the original message.

It is important to realize that the shorter the interval is, the more bits are needed on average to represent it. For instance [0.4, 0.9) can be represented with 1 bit; 0.1 in binary ($2^{-1} = 0.5$ in decimal). While [0.3, 0.4) requires 3 bits; 0.011 in binary ($2^{-2} + 2^{-3} = 0.375$ in decimal).

Another important note is that the decoder does not implicitly know when to stop decoding. In theory, all codewords represent an infinite sequence of symbols. Without any additional information the decoder will not know to stop once the message has been decoded, but will instead continue, producing garbage symbols. One way to let the decoder know when to stop is to simply send the length of the message together with the codeword, another is to use a special end of message symbol.

Each symbol is assigned a subinterval according to its probability, beginning at the cumulative sum of the given symbol and ending at the cumulative sum of the next. So, if the data to be compressed contains the symbols 0, 2 and E with probabilities 0.6, 0.2 and 0.2 respectively, then the subintervals would be $[0, 0.6)$, $[0.6, 0.8)$ and $[0.8, 1)$.

The compressed length (in bits) of a sequence $x_1, x_2, \ldots, x_n$, compressed using arithmetic coding, can very accurately be calculated by

$$l = \left\lceil -\log_2 \prod_{t=1}^{n} p_t \right\rceil + 1 = \left\lceil -\sum_{t=1}^{n} \log(p_t) \right\rceil + 1$$

where $p_t$ is the probability assigned to symbol $x_t$. Notice that $p_t$ is an assigned value used for encoding and decoding and that it does not have to be close to the true probability of the symbol $x_t$. The length $l$ is minimized and coincides with the entropy $\times\, n$ when all $p_t$ are equal to the frequency of $x_t$ in the given sequence.

**Example:** The sequence to be encoded is 0020E, where E marks the end of the stream. The probabilities for the symbols are given in the table below together with the initial subintervals.

| Symbol | Probability | Cumulative sum | Subintervals |
|---|---|---|---|
| 0 | 0.6 | 0 | $[0, 0.6)$ |
| 2 | 0.2 | 0.6 | $[0.6, 0.8)$ |
| E | 0.2 | 0.8 | $[0.8, 1)$ |

Here "[" means inclusive and ")" exclusive, so 0.8 does not belong to the second interval, only the final interval. The beginning of each interval is picked as the cumulative sum of probabilities for that symbol. The coding procedure is recurrent and at each step the beginning of the interval is recomputed as F=F+G*q, where F is the previous beginning, G is the probability of the processed sequence (size of the interval) and q is the cumulative sum of probabilities for the given symbol. This process is illustrated in Figure 2.9.
The final interval is 0.25056 – 0.2592. To uniquely decode the sequence, any

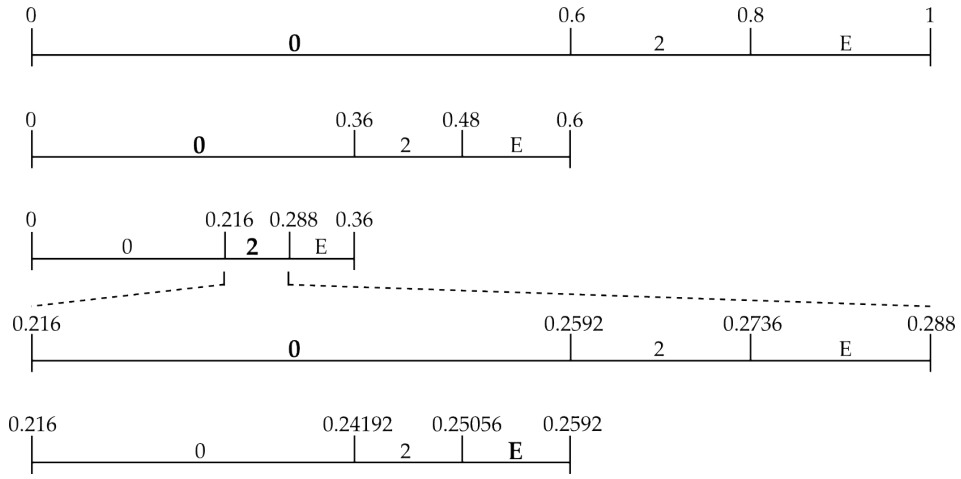


**Figure 2.9:** Encoding a short sequence using arithmetic coding.

fractional number can be picked within this interval. The shortest fractional binary number is 0.0100001 ($2^{-2} + 2^{-7} = 0.2578125$ in decimal). Since it is given that the value is between zero and one only the 7 fractional bits need to be saved. This is close to the expected value of $l = -\lceil 3\log_2(0.6) + 2\log_2(0.2) \rceil + 1 = 8$ bits. The inclusion of the escape symbol is quite costly for this very short sequence. As seen in the example from 2.5.2 exponential Golomb coding can compress the same sequence using only 6 bits (no extra bits are spent on the length of the sequence). Arithmetic coding is most suitable for somewhat longer to very long sequences.

---

One notable issue with this compression scheme is the need for the probabilities to be known by the decoder. This requires the probabilities to be communicated which results in extra information sent; this extra information should also preferably be compressed. The solution is to use adaptive arithmetic coding.

### 2.5.4 Adaptive Arithmetic Coding

Essentially, adaptive arithmetic coding is an improved form of arithmetic coding. Rather than using a fixed frequency table that needs to be available before encoding/decoding, the initial frequency table will just be a guess. The initial guess can be as simple as giving the same probability to all symbols, or some educated guess based on the compression settings or other known parameters. The symbols encoded are counted and the frequency table updated continuously according to already encoded symbols. As more symbols are encoded, the frequency table will converge towards the true frequencies of the symbols.

Using this method the compression for the first symbols may be poor if the initial guess is bad. But the frequency table converges fairly quickly and the symbols after the first few will receive a compression very close to the entropy

(less then 1% larger). Two notable benefits of this adaptive approach is that streamed data can be compressed on the fly, there is no need to wait for all data to be available before compression is started. The other advantage is that the frequency table is essentially compressed together with the data.

It is important to note that the frequency table of the decoder must be in sync with the encoder for each symbol. That is, the decoder must use the exact same frequency table when decoding a symbol as the encoder used. However, the encoder can then change its frequency table before encoding the next symbol as long as the decoder makes the exact same update. What this means is that both encoder and decoder must start out with the same guess of the frequency table. Then when encoding, the initial frequency table must be used for the first symbol, after which the frequency table can be updated before encoding the next symbol. This way the decoder can decode the first symbol and make the same update to its own frequency table, mirroring the encoder.

There are different ways to estimate the probability of future symbols based on already decoded symbols. Assume that $x_1, x_2, \ldots, x_t$ is a sequence observed at the encoder. The estimate that the next symbol $x_{t+1}$ will equal $a$ can then be written as $\hat{P}(x_{t+1} = a)$. A first attempt to estimate $x_{t+1}$ might be

$$\hat{P}(x_{t+1} = a) = \frac{N_t(a)}{t} \tag{1}$$

where $N_t(a)$ is the number of occurrences of the value $a$ in the sequence of length $t$. However, the first occurrence of the value $a$ will always result in the probability being estimated as zero. A zero probability means the symbol cannot be encoded, resulting in the encoding process failing. Instead a biased symbol probability estimate can be used. For instance, all symbols can be given an initial probability of $1/M$, where $M$ denotes the alphabet size of the source. The estimate can then be written as

$$\hat{P}(x_{t+1} = a) = \frac{N_t(a) + 1}{t + M},\ t = 0, 1, 2, \ldots \tag{2}$$

Zero probabilities are avoided and for large $t$ this estimate is quite close to (1). However, the convergence towards the true symbol probabilities of the source is not ideal. A better estimate suggested by [11] is

$$\hat{P}(x_{t+1} = a) = \frac{N_t(a) + 1/2}{t + M/2},\ t = 0, 1, 2, \ldots \tag{3}$$

This is essentially the same as (2) but with a smaller initial weight for each symbol, resulting in a faster convergence.

Other approaches [12] used for estimating the probability distribution make use of an additional symbol called escape-symbol. If the next symbol $x_{t+1}$ is equal to a previously encoded symbol it is assigned the probability

$$\hat{P}(x_{t+1} = a) = \frac{N_t(a)}{t + 1}$$

Otherwise it is encoded as the escape symbol followed by the new symbol. The escape symbol is assigned the probability $\frac{1}{t+1}$ and the new symbol is assigned the

probability $\frac{1}{M-M_t}$, where $M_t$ denotes the number of different alphabet symbols which already occurred. The probability when $x_{t+1}$ is not equal to a previously encoded symbol is then the combined probability of the above, that is

$$\hat{P}(x_{t+1} = a) = \frac{1}{(t+1)(M - M_t)}$$

The escape symbol can be seen both as a symbol and as a container for other symbols. The already encoded symbols cover one range of the interval 0 to 1 and the escape symbol covers the remaining range. Then the escape symbol interval is divided into equally large intervals, one for each of the remaining symbols in the alphabet not yet encoded.

Once all symbols in the alphabet have been encoded at least once, formula (1) will be used. Thus we have

$$\hat{P}(x_{t+1} = a) = \begin{cases} \frac{N_t(a)}{t+1} & \text{if } N_t(a) > 0 \text{ and } M < M_t \\ \frac{1}{(t+1)(M-M_t)} & \text{if } N_t(a) = 0 \text{ and } M < M_t \\ \frac{N_t(a)}{t} & \text{if } M = M_t \end{cases} \tag{4}$$

Another encoding method similar to (4) is based on the assumption that is better to have a larger probability of the escape-symbol at the first steps of encoding since almost all symbols are associated with the escape-symbol in the beginning.

$$\hat{P}(x_{t+1} = a) = \begin{cases} \frac{N_t(a)-1/2}{t} & \text{if } N_t(a) > 0 \text{ and } M < M_t \\ \frac{1}{2t(M-M_t)} & \text{if } N_t(a) = 0 \text{ and } M < M_t \\ \frac{N_t(a)}{t} & \text{if } M = M_t \end{cases} \tag{5}$$

Notice that for $t = 0$ we define

$$\frac{1}{2t(M - M_t)} = \frac{1}{M}$$

## 2.6 Context Coding

Ideally the spatial transform will remove all correlations in the image, but this is almost never the case and some correlations will remain. Context coding [1] is a way to efficiently reduce the correlations further by taking advantage of known properties of the spatial transform.

**Example:** Let's say a known property of the transform is that zeroes are most often encountered in groups. Whenever one zero is encountered the chance of the next value being zero is very high until another value than zero is encountered. In this case context coding could be used to sort the values into two separate streams. All values are put into *stream 1* until and including the encountered zero. All following zeroes are put into *stream 2* up until and including the first

nonzero value. The process is repeated until the end of the stream. Both streams are then compressed using arithmetic coding separately. The compression will be improved because the probability and therefore compression of zeroes will go up in *stream 2*, and the same goes for the nonzeroes in *stream 1*. When decoding the streams the decoder can follow the same procedure. That is, all symbols are taken from *stream 1* until a zero is encountered, then from *stream 2* until a nonzero is encountered, and so on.

# Chapter 3

# Implementation Issues for Application

This section will cover some of the issues that where encountered and some of the choices made during this thesis.

## 3.1 Color Space Transform

Three different luminance-chrominance transforms where considered. The first requires floating point precision and is used in the JPEG standard: [13]

$$\begin{array}{lcllll} Y & = & 0.299R & +0.587G & +0.114B & \\ C_b & = -0.16874R & -0.33126G & +0.5B & +128 \\ C_r & = & 0.5R & -0.41869G & -0.08131B & +128 \end{array}$$

The second is used in the JPEG2000 standard and can be computed using integer arithmetic: [14]

$$\begin{aligned} Y &= \frac{R + 2G + B}{4} \\ C_b &= B - G \\ C_r &= R - G \end{aligned}$$

The third is comes from a paper [15] using a alternative transform which can also be done using integer arithmetic:

$$\begin{aligned} Y &= R + G + B \\ C_b &= R - 2G + B \\ C_r &= R - B \end{aligned}$$

From a computational perspective the first is considerably more expensive since it requires floating point multiplication. This conversion also introduces some errors. Some initial testing suggested that this transform did not decorrelate the color components significantly better than the other two. Due to this and the fact that it is slower and introduces errors it was discarded early in development.

The second and third differ slightly in how the components are calculated and which of them has the best decorrelating properties will differ depending on the

image. Since they both use integer arithmetic neither will introduce any rounding errors and both can therefore be applied losslessly. It should be mentioned that the third requires a larger numeric range which is likely to negatively impact lossless compression. For lossy compression it is not a problem since the numeric range is again reduced after the quantization step.

Since it was not obvious which of the later two would give the best results they where both tested.

## 3.2 Wavelet Transform using Lifting

As mentioned previously, the wavelet transform can be performed iteratively using a filter bank. Taking a closer look at one stage of the filter bank, Figure 3.1, each element is filtered twice and then half of the calculated values are thrown away.

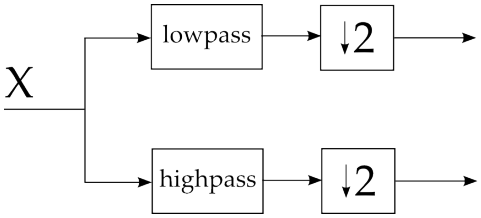


**Figure 3.1:** One stage of an iterated filter bank.

The filters can however be combined to avoid the wasted calculations and thus reducing the calculations by half. Using a technique called lifting [16, 17] the calculations can be reduced even further. This technique allows the input stream to be split into even and odd indices. Each lifting stage is very simple, but *lifts* the complexity of the overall filter. An $N$ stage lifting filter can be seen in Figure 3.2. This filter is equivalent to that shown in Figure 3.1, but uses much fewer

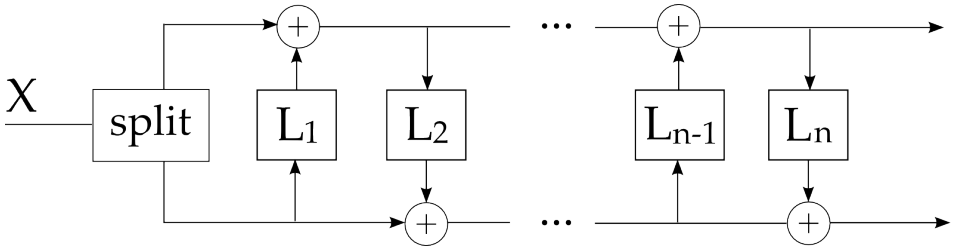


**Figure 3.2:** One stage of an iterated filter bank using N lifting steps.

calculations.

To give an idea of how to design a filter using lifting, the filter introduced in chapter 1 will be used as an example.

$$H(z) = -\frac{1}{8}z^{-2} + \frac{1}{4}z^{-1} + \frac{3}{4} + \frac{1}{4}z - \frac{1}{8}z^{2}$$
$$G(z) = \frac{1}{4}z^{-2} - \frac{1}{2}z^{-1} + \frac{1}{4}$$

The lifting scheme shown in figure 3.2 inputs two symbols each time unit and no decimation is performed. To account for this the filter components with an odd time delay are grouped together and shifted in time

$$H(z) = -\frac{1}{8}z^{-2} + \frac{3}{4} - \frac{1}{8}z^{2} + z^{-1}\left(\frac{1}{4} + \frac{1}{4}z^{2}\right)$$
$$G(z) = \frac{1}{4}z^{-2} + \frac{1}{4} + z^{-1}\left(-\frac{1}{2}\right)$$

The two filters can now be rewritten in matrix form as a polyphase matrix.

$$P(z) = \begin{pmatrix} -\frac{1}{8}z^{-1} + \frac{3}{4} - \frac{1}{8}z & \frac{1}{4} + \frac{1}{4}z \\ \frac{1}{4}z^{-1} + \frac{1}{4} & -\frac{1}{2} \end{pmatrix}$$

Notice that the time delay has been reduced to half and that the right column has been shifted in time compared to the original filters. Finally the matrix can be factorized into several simple lifting steps.

$$P(z) = \begin{pmatrix} 1 & 0 \\ 0 & -\frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{4} + \frac{1}{4}z \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\frac{1}{2}z^{-1} - \frac{1}{2} & 1 \end{pmatrix}$$

The corresponding filter using lifting can be seen in figure 3.3.

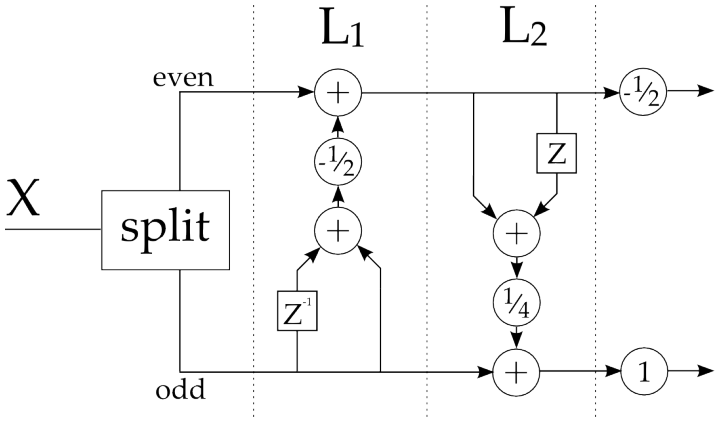


**Figure 3.3:** Lifting filter used in the thesis.

It is important to note that when implemented using integer arithmetic the final scaling by $-\frac{1}{2}$ is a lossy operation. The scaling doesn't improve the decorrelation, however, so it is best to simply ignore the scaling when performing the transformation. It is just a matter of keeping in mind that the scaling is not correct if any operation is performed later that requires correct scaling.

However, the division in the L1 and L2 stages is not an issue as long as the rounding is performed in a consistent way (for instance, always rounding down). While there will be "rounding errors" during the transformation, the inversed transformation will reverse the errors and still give a perfect reconstruction.

### Computational complexity

As can be seen in figure 3.3, 2 additions and one shift operation are needed for the odd input symbols, plus an additional 2 additions and a shift for the even. So, applying the filter once to the entire image requires only 3 operations per pixel. One iteration of the transform requires the filter to be applied twice however, both in the horizontal and vertical direction. Each additional iteration requires the filter to be applied again on $\frac{1}{4}$ of the image, then $\frac{1}{4^2}, \frac{1}{4^3} \cdots$. This sum quickly approaches $\frac{4}{3}$. So the total number of operations per pixel for the lifting transform using this filter will be close to $3{*}2{*}\frac{4}{3} = 8$ integer operations per pixel. This can be compared to a fast DCT implementation using a block size of $N = 8$ which requires $\frac{3}{2}N\log_2(N) - N + 1 = 29$ additions and $\frac{N}{2}\log_2(N) = 12$ multiplications, a total of 41 operations per pixel [4]. Faster algorithms using an integer approximation of the DCT exist; one such algorithm is described in [18] and requires 16 integer operations per pixel.

## 3.3 Adaptive Arithmetic Coding

Since arithmetic coding compresses data very close to the entropy, it is possible to make very accurate estimates of the compression and no actual encoder was implemented. Instead the compression was estimated by doing detailed simulation of the adaptive arithmetic coding process.

The implemented method was based on equation (3) from section 2.5.4, but taking inspiration from the usage of escape-symbols used in equations (4) and (5). The problem with all three is that the size of the alphabet needs to be known. In the case it is not known it would need to be guessed. Making an initial frequency table which includes all 32 bit integers, just to be sure, would be extremely inefficient. The way this was dealt with was to include an escape-symbol in the frequency table. Whenever the encoder encounters a new symbol not already present in the frequency table, the new symbol is instead encoded as the escape symbol followed by the new symbol encoded using exponential Golomb coding. Once the new symbol has been encoded, it is then added to the frequency table, after which it can be encoded normally whenever encountered again. When in turn the decoder encounters the escape symbol, it adds the following Golomb coded symbol to its own frequency table, keeping the encoder and decoder in sync.

One additional feature was added to the encoding procedure. The initial alphabet size could be set before encoding started through a parameter T. All values smaller than T plus the escape symbol are given an equal probability and only values T and larger are encoded using exponential Golomb coding. This way fewer symbols need to be compressed using exponential Golomb coding. Also, it is known that all symbols being encoded using exponential Golomb coding will be T or larger. By taking advantage of this knowledge these values can first be subtracted by T, reducing the average number of bits needed to encode them. Setting the alphabet size correctly means better overall compression, while setting the alphabet size too large reduces compression by giving probabilities to symbols that may never be used.

The algorithm is given in detail on the next page.

**Input:**
- Data to be encoded $\mathbf{x} = (x_0, x_1, ..., x_n)$
- Threshold parameter T
- Golomb constant $k$

**Initialize:**
- Create a list $N_t$ with the escape symbol **E** and all integers up to but not including the threshold T (initial count of all symbols set to zero)
- Set M to T+1 (+1 accounts for the escape symbol)
- Set the number of compressed bits B to $G(k, 0) + G(T, k)$
- Set s to 0 (counter for additional symbols)

**for** $t = 0$ **to** n **do**
- Convert input value $x_t$ to a positive integer: remove sign, multiply by two and subtract by 1 if it was negative

**if** $x_t < T$ **or** $N_t(x_t) > 0$ **do**
- Set $p$ to $\frac{N_t(x_t)+1/2}{t+s+M/2}$
- Set B to B$-\log_2(p)$

**else**
- Set $p$ to $\frac{N_t(\mathbf{E})+1/2}{t+s+M/2}$
- Set B to B$-\log_2(p)$
- Set B to B$+G(x_t - T, k)$
- Set $N_t(\mathbf{E})$ to $N_t(\mathbf{E}) + 1$
- Set $s$ to $s + 1$
- Initialize $N_t(x_t)$ to 0
- Set M to M + 1

**end**
- Set $N_t(x_t)$ to $N_t(x_t) + 1$

**end**

**function** G(x, k)
**return** $2 * \lfloor \log_2(\lfloor x/2^k \rfloor + 1) \rfloor + 1 + k$
**end**

**Figure 3.4:** Pseudo code for adaptive arithmetic encoder

## 3.4 Context Coding

The implementation of the context coding was done based on the choice of using the wavelet transform. After each iteration of the wavelet transform some correlations tend to remain. In particular, it was observed that the value of coefficients after each wavelet transformation was related to their parent coefficient obtained after the following transformation. Figure 3.5 shows how the coefficients relate.

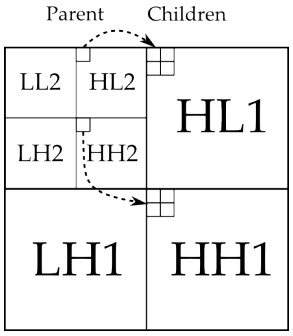


**Figure 3.5:** Parent-child relation of coefficients in a wavelet decomposition.

The contexts were split based on the absolute value of the parent. Two parameters determined how the splitting was performed; *numberOfStreams* and *stepSize*. The *numberOfStreams* determined the number of streams the data was split into. All children belonging to a parent were put into a given stream according to

$$streamNumber = \left\lfloor \frac{|parentValue|}{stepSize} \right\rfloor + 1 \tag{3.1}$$

The *streamNumber* was then limited by the *numberOfStreams* parameter

$$streamNumber = \min(streamNumber, numberOfStreams) \tag{3.2}$$

This limitation on the *streamNumber* means that only smaller values are sorted while all larger ones are put into the last stream.

The algorithm is described below, for simplicity only 2 transformation iterations are assumed as in Figure 3.5.

1. Store a copy of LL2 as it is in an extra stream; *stream 0*
2. Take the LL2 part and transform it one more time. Keep the HL3, LH3 and HH3 parts (the LL3 part is discarded).
3. Iterate over all the (parent) elements in HL3, LH3 and HH3 in turn. Put the corresponding children found in HL2, LH2 and HH2 into the stream according to (3.1) and (3.2).
4. Repeat the above, i.e. iterate over all the (parent) elements in HL2, LH2 and HH2 this time.
5. Encode the streams individually using adaptive arithmetic coding

## 3.5 Motion Compensation

Two different motion compensation techniques were developed. One method denoted *method 1* investigated how well it works to do motion compensation in the transformation domain. The other method denoted *method 2* investigated how well wavelet based compression compares to H.264 when ignoring most of the difficulties related to motion compensation.

### Method 1

One big issue with block-based compensation in the image domain is that it results in many small residual blocks, which need to be transformed individually. For a DCT approach it makes little to no difference since the transformation is block-based as well. However, for a wavelet-based approach many of the advantages over DCT are lost when transforming only small blocks, such as better decorrelation at block borders. By doing the motion compensation in the transformation domain this problem is avoided.

Another advantage gained is that the residual is applied directly in the transformation domain. This means the decoder can maintain a transformed copy of the image, which gives some added flexibility, for instance if the decoder is only interested in a zoomed-out version of a very large image area. In this case the decoder could opt to process only the data needed to obtain the smallest scale image, rather then first obtaining the full scale image and than scaling it down.

#### Block-based motion compensation in transformation domain

Below the full compression process is described very briefly in just five steps, after which point three to five will be described in more detail.

1. Convert the RGB color space to YUV (see section 3.1).
2. Perform wavelet transformation and quantization on each of the Y, U and V components (see sections 3.2 and 2.4).
3. Perform motion estimation on the transformed Y component only.
4. Use the motion vectors obtained to form residuals for all of the Y, U and V components.
5. Compress residuals and motion vectors using adaptive arithmetic coding (see section 3.3).

**Step 3** For efficiency the motion compensation is performed hierarchically [19]. First a full search is performed on the LL part of the Y component. This gives a rough approximation of the movements since the LL part is a miniature of the full image. Then for each of the HL, LH and HH parts, starting with the lowest frequencies, a search is performed using a small window size. The search done for each block is centered around the location pointed to by the motion vectors of the previous search. The new motion vectors obtained refine the approximation of the first search and are used as prediction for the next. This is illustrated in Figure

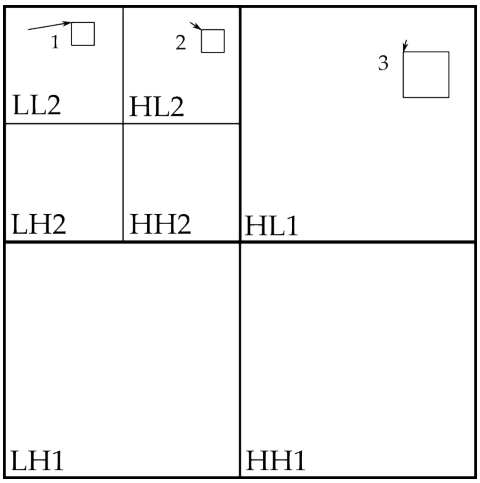


**Figure 3.6:** Hierarchical motion estimation. A full search is performed in LL obtaining 1. The search continues in 2 and 3. Each search is based on the result of the previous search.

3.6.
The following search criterion was used to find a best match

$$\min_{\alpha,\beta} \left\{ \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} |x_c(m,n) - x_p(m+\alpha, n+\beta)| \right\}$$

where $x_c(m,n)$ and $x_p(m,n)$ are pixels of the current block and of the co-sited block of the previous frame, $\alpha$ and $\beta$ are shifts of the pixels along the coordinate axes. For the full search all possible shifts of $\alpha$ and $\beta$ are tested, while for further searches only a few shifts around the location given by the previous search are tested.

**Step 4** Once all motion vectors are obtained, the residuals can be formed. The same motion vectors for the Y component are used for the U and V components as well. The residuals are formed simply by subtracting the block indicated by the motion vector of the previous frame from the block of the current frame.

**Step 5** The method used for context coding discussed in section 3.4 does not work here. The reason is simply that the same correlations cannot be found now that we have blocks of residuals. Instead all residual blocks from each of the LL, LH, HL and HH are encoded together as a separate stream using adaptive arithmetic coding. Taking Figure 3.6 as example, 7 separate streams would be encoded. The assumption is that the numeric range of the residual blocks within each LL, LH, HL and HH part will be more similar than between blocks of different parts. In the same way, the motion vectors for each part are encoded as a separate stream.

### Method 2

One scenario that is likely to be common in the applications where this compression method will be used is panning over still images. As an example, a remote control application is run on a mobile phone with a small screen and is used to control a computer with a significantly larger screen. The computer is running a web browser with mostly static content such as menus, images and text. In this scenario motion compensation will not be particularly useful; instead it is sufficient to compensate for the global translational movements.

#### Global translational motion compensation in image domain

Just as for *method 1*, the full process is described very briefly below, after which point two and three are described in more detail.

1. Convert the RGB color space to YUV (see section 3.1).
2. Global motion estimation performed.
3. Use the motion vector obtained to form one residual of the matching parts of the previous and new frame. Also append the parts of the new frame that did not match the previous frame.
4. Perform wavelet transformation and quantization (see sections 3.2 and 2.4).
5. Use of context coding and adaptive arithmetic coding was done as discussed in section 3.4, the motion vector being compressed using exponential Golomb coding.

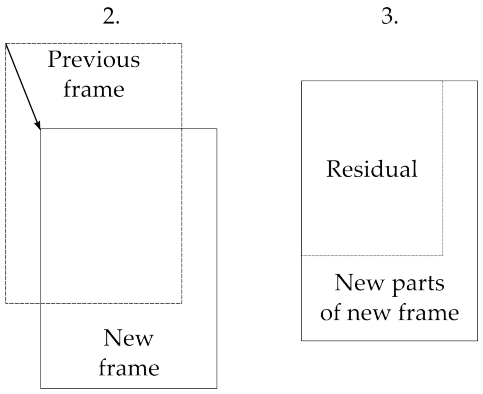


**Figure 3.7:** Steps two and three are illustrated for motion compensation *method 2*. First the motion estimation is performed, then the residual is formed together with the non-matching parts of the new frame.

**Step 2** The global motion estimation implemented was done very simply. The entire frame was considered one block and a full search was performed to find the best match between the new and the previous frames. The search criterion was the same as for *method 1*, but testing only every 32nd pixel to reduce the computational complexity.

$$\min_{\alpha,\beta} \left\{ \sum_{m=0}^{\frac{M}{32}-1} \sum_{n=0}^{\frac{N}{32}-1} |x_c(32m, 32n) - x_p(32m+\alpha, 32n+\beta)| \right\}$$

**Step 3** The motion vector obtained from Step 2 was used to form the residual between the two frames, by subtracting the overlapping part of the previous frame from the new. The new parts of the new frame were kept. The newly formed image, residual plus new parts, was then wavelet transformed and quantized (step 4).

# Chapter 4
# Results

## 4.1 Methodology

The image and video compression algorithms developed in this thesis were implemented and run in Matlab. When generating the compressed images all steps were performed except the entropy coding, since this step is lossless and has no impact on the quality of the reconstructed image. The sizes of the images were instead estimated using the algorithm described in section 3.3, but without generating any compressed data. While the estimate should be very close to the amount of data that a real implementation would produce, it does not account for any header data, such as the resolution of the image. However, this extra header data should be very small and have little impact on the results. The sizes of all other images and video files were measured as the actual file size on disk. For the video files the size of the mpeg container was excluded from the file size to give a more fair size comparison.

The quality measure used when comparing different methods was PSNR (Peak Signal to Noise Ratio), defined as

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAXI^2}{MSE}\right)$$

with $MAXI$ being the maximum value of the measured data and $MSE$ being the mean square error. For all measurements $MAXI$ was 255. All PSNR comparisons and calculations were performed in Matlab. The PSNR depends on the color space used and to increase the chance of allowing comparisons with other works, all images where first transformed to the color space used in JPEG (see section 3.1).

All JPEG images were compressed with GIMP using the optimize option, 4:2:2 subsampling, floating-point as the DCT method and adjusting the compression through the quality parameter. All PNG images where also compressed using GIMP with the compression level set to 9 and saving no additional information. For JPEG2000 OpenJPEG was used and the only option set was the rate (compression) with all other options at their defaults. The H.264 video compression was done using x264. In a attempt to make the comparison as fair as possible some of the H.264 options were disabled. Only previous frames were allowed to be used and then only a single frame used as a reference frame. The deblocking filter was

disabled and visual optimizations that reduce PSNR were disabled. The following settings were used: profile=baseline, level=3, preset=ultrafast, bframes=0, ref=1, no-deblock=true, no-psy=true and the quality parameter crf adjusted to obtain the desired PSNR.

## 4.2 Finding good compression parameters

To achieve good results for both compression and quality, good compression parameters need to be found, parameters such as iterations of the wavelet transform, quantization and number of contexts. The number of iterations was chosen according to

$$iterations = \lfloor \log_2(\min(M, N)) \rfloor - 5$$

where $M$ and $N$ are the dimensions of the image to be compressed. The formula guarantees that the smallest dimension of the LL part is between $2^5$ and $2^6$. While more iterations certainly are possible it was discovered through trial and error that the compression benefit was very minor. It was also harder to find the right compression parameters with too many iterations. Quantization errors in the LL part has a larger impact on the image quality with more iterations. This makes it harder to find the right balance between quantization/compression and good quality.

The quantization of the wavelet coefficients needs to be different for each iteration level for optimal results. That is, the highest frequency coefficients should be quantized the most while the lowest frequency coefficients should be quantized the least. Some research went into finding good quantization parameters $\mathbf{Q} = Q_1, Q_2, \ldots, Q_n$ and linking them to a single compression parameter C. Here $Q_x$ denotes the quantization of wavelet coefficients of the same number of iterations, for instance, $Q_1$ is the quantization parameter for LH1, HL1 and HH1. One more quantization parameter is used then the number of iterations and the last $Q_n$ is used for the LL coefficients. The quantization was done as follows

$$y = \left\lfloor \frac{x}{Q_i} \right\rceil$$

were $x$ is the original wavelet coefficient and $y$ the quantized coefficient. A simulation was run for several images of size $512 \times 512$. For each increment of $C$, the $Q_x$ was incremented that maximized the ratio PSNR/file-size. The graphs in 4.1 show the results for the image Baboon.
The simulation was run for just the Y coefficients and for the U and V coefficients both being compressed using the same set of $\mathbf{Q}$. While the results for the Y coefficients looked fairly similar between different images, the results for the U and V coefficients varied more, especially for the blue (first) and green (second) lines which did not have very linear increases. A linear approximation was done for the Y coefficients of the images. The same approximation turned out to look decent for the U and V coefficients and gave good compression results. While perhaps not optimal, the same linear approximation was used for all of the Y, U and V coefficients. The array of $Q$ values for a given compression C is given by

$$\mathbf{Q} = [0.58 \;\; 0.36 \;\; 0.16 \;\; 0.06 \;\; 0.03] * C + 1$$

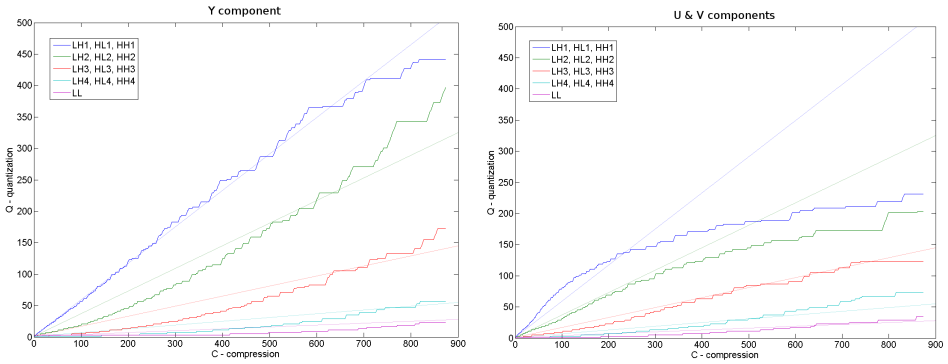


**Figure 4.1:** The graphs show the optimal quantization Q at different compression levels C - optimal in the sense that the PSNR/file-size ratio is maximized. The thin straight lines show the linear approximation made.

It was found that finding optimal values for the number of contexts used and for the adaptive arithmetic coding was more difficult. The choice of values for these parameters also seemed to have a fairly small impact on compression as long as they were reasonable. Because of this these values were simply set once and were not made dependent on the compression C. The values used for still image compression and for motion compensation *method 2*, can be found in Table 4.1.

| | Context coding | | Adap-arithmetic | Exp-Golomb |
|---|---|---|---|---|
| | nbrOfStreams | stepSize | threshold | k |
| Y coeff | 6 | 1 | 8 | 6 |
| UV coeff | 3 | 1 | 3 | 5 |

**Table 4.1:** Compression parameters used for still image compression.

Motion compensation *method 1* did not use any context coding. The values used for *threshold* and $k$ were 2 and 3 respectively and the same values were used for all of the Y, U and V coefficients. The reason for the lower numbers for *method 1* is that residuals are compressed, which typically are close to zero. For *method 1* the block size and window size (search area) also needed to be chosen. No attempt was made to automate the choice, instead different block sizes and window sizes were tried manually for the Foreman movie. 3 iterations of the wavelet transform were done and the optimal block sizes were found to be the following; $4 \times 4$ for the LL and LH3, HL3, HH3 components and $8 \times 8$ for the 6 higher frequency components. It was found to be sufficient to have a window size of $[-2, 2]$. That is, testing the block's current position given by the previous motion vector and testing $\pm 2$ pixels along the x- and y-axis.

## 4.3 Comparison of Still Images

A comparison was done using the proposed image compression method using three different color spaces: RBG, YUV and YUV2k (see section 3.1). Included in the comparison is JPEG2000, using the lossless compression mode, and PNG. Five images were compressed and the result can be seen in Table A.1.

Using the YUV color space, five test images were compressed at five different compression ratios. The same test images were also compressed using JPEG2000 and JPEG using the same compression ratios. The PSNR for the compressed images were also computed and the results can be seen in Tables A.2 to A.6.

## 4.4 Comparison of Movies

The Foreman video was compressed using H.264 and the proposed *method 1* using three different frame rates and the min, max and average PSNR values were computed. The results can be seen in tables A.7 and A.8. Note that the raw format was saved in I420 which uses $2 \times 2$ subsampled U and V components, that is, only 1/4 of those components are stored. In a sense the raw format is already compressed which leads to seemingly lower compression. The average compression is 21.1 when using still image compression only (image quality is identical). As can be seen in the table A.8, the improvement in compression from the motion compensation is roughly 42% for all three frame rates.

A video made by doing a screen capture while browsing the LTH website was compressed using both H.264 and the proposed *method 2*. The type of motions in the movie consisted solely of global translational movements where the view area was moved around over the much larger web-browser window. The result can be seen in Table A.9. The sequence was recorded and compressed at 15 frames per second and at a 320x460 resolution. The sequence was 280 frames long and had an uncompressed size of 123 MB.

## 4.5 Discussion

The method for still image compression developed in this thesis is consistently performing better then JPEG, but slightly worse then JPEG2000. That JPEG2000 performs better should not be a surprise, after all the method in this thesis has received considerably less development time in comparison. There are several areas that could be improved. The quantization could be optimized further. The adaptive arithmetic coding parameters would be closer to optimal if set according to the quantization level of the wavelet coefficients being encoded. Furthermore the context coding has only received limited development and is likely far from optimal. Considering how close the results are to JPEG2000 already and the room for improvement, there is a good chance that the developed method in this thesis could surpass JPEG2000 with further work.

*Method 1* shows surprisingly good results considering that the translation variance of the wavelet transform limits the compression gain of the motion compensation. Something to note is that *Method 1* is largely unaffected by the frame

rate. Since method 1 has very low complexity the result suggests it could be very suitable for the type of applications it is aimed for. While it certainly is possible to stream H.264 at a higher quality by using more costly compression options, it is also not desirable that the streaming process takes up too much resources. There are also a lot of improvements that could be made to *method 1*. One such improvement is good global motion compensation. If the previous image is made to match the new image by translating, rotating and stretching it before transformation, then the translation variance will be less of an issue and the compression can be expected to improve a lot.

*Method 2* appears to be superior to H.264 for the particular content compressed, despite the very simple motion compensation method used. The compression could also easily be improved further. Putting the residual together with the new image data reduces compression since there will most often be a sharp edge between the two that is difficult to compress . Compressing the two parts individually instead should give noticeably better results. A larger improvement in compression could be gained by taking advantage of the fact that the view area often moves over a fairly static background. That which has been viewed and which leaves the view area can be buffered until the view area returns to the same location. If nothing has changed the buffer is used and no new data needs to be sent.

The two motion compensation methods each tests a fairly extreme case, both of which are very likely scenarios. For instance, a text document could be viewed, or a movie. For the best results an implementation should incorporate both methods. A topic for further research could be how *method 1* and *method 2* could be realized in an actual implementation, as well as finding ways to refine and optimize the methods.

Several requirements (see chapter 1) were imposed on the compression stream. The first can be considered fulfilled since no buffering/delay of new images is needed at all. The only delay the compression scheme adds is the computational time needed to perform the compression. Since the computational complexity is very low the second requirement is also fulfilled. Changing the compression between frames can easily be done by simply changing the compression parameter fulfilling requirement three. The compression is done in such a way that the lowest frequency part is encoded/decoded first and the highest frequency part last. Since the wavelet transform naturally does a progressive updating of the image, starting with the lowest frequency, progressive updating of the image is easy to support. Efficient handling of different zoom levels can be done by only performing as many wavelet transformations as needed to get close to the desired zoom level. Once close the small scale image can be scaled by traditional methods. The amount of data that needs to be accessed and the computational cost will be roughly the same regardless of zoom level fulfilling requirement five.

# Chapter 5
# Conclusions

In this thesis two video compression techniques have been developed, denoted *method 1* and *method 2*. *Method 1* focuses on block based motion compensation and *method 2* focuses on simple translational global motion compensation. The first method is special in that it performs the motion compensation in the transformation domain and encodes the resulting residuals losslessly. The improvement in compression compared to still image compression is somewhat modest, around 42%, but the method is computationally cheap and can be combined with *method 2*.

*Method 2* deals with the translational movements that are expected to be very common for the target applications. The test results favor *method 2* over H.264 for the content compressed.

The goal of this master thesis was to develop an efficient compression algorithm for video with special demands. It can be concluded that this goal has been accomplished as the algorithm indeed is efficient and the requirements in section 1 have all been fulfilled. Furthermore it is the belief of the author of this thesis that *method 1* and *method 2* combined will be superior to any other compression method available for remote control applications currently on the market. Several such applications have been tested and none have appeared to provide image compression even close to what is provided with *method 2*.

# References


[1] I. Bocharova. Compression for Multimedia. *Cambridge Univ. Press, Cambridge, US* (2010): 91–108, 121–132, 249–251.

[2] S.A. Khayam. The discrete cosine transforms (DCT): theory and application. *Technical Report, DCT Tutorial* (2003).

[3] I. E. G. Richardson. H.264 and MPEG-4 Video Compression. *John Wiley & Sons* (2003): 30–42, 180–183.

[4] W. Yuan, P. Hao and C. Xu. Matrix Factorization for Fast DCT Algorithms. *IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, **3** (2006): 948–951.

[5] C. Valens. A Really Friendly Guide To Wavelets. *http://www.polyvalens.com/blog/?page_id=15* (17 Oct, 2011).

[6] S. Peter and W. Booth. A New Fast Motion Search Algorithm for Block Based Video Encoders. *Waterloo, Ontario, Canada* (2003). 6–7.

[7] C. E. Shannon. A mathematical theory of communications. *Bell Syst. Tech. J.*, **27**(July) (1948): 379–423.

[8] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, **40**(9) (1952): 1098–1101.

[9] S. W. Golomb. Run-length encodings. *IEEE Trans. Infn. Theory*, **12**(3) (1966): 399–401.

[10] A. Said. Introduction to Arithmetic Coding Theory and Practice. *Hewlett-Packard Laboratories Report*, **HPL-2004-76**(April) (2004). 1–22.

[11] R. E. Krichevsky and V. K. Trofimov. The Performance of Universal Encoding, *IEEE Trans. Infn. Theory*, **IT-27**(2) (1981): 199–207.

[12] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, **30**(6) (1987): 530–540.

[13] T. Acharya and P. Tsai. JPEG2000 standard for image compression: concepts, algorithms and VLSI. *John Wiley & Sons* (2005): 60–60.

[14] Official site of the Joint Photographic Experts Group. *http://www.jpeg.org/.demo/FAQJpeg2k/functionalities.htm* (17 Oct, 2011).

[15] G. S. Gupta and D. Bailey. Discrete YUV Look-up Tables for Fast Colour Segmentation for Robotic Applications. *Canadian Conference on Electrical and Computer Engineering* (2008): 963–968.

[16] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl.*, **4**(3) (1998): 245–267.

[17] C. Valens. The Fast Lifting Wavelet Transform. *http://www.polyvalens.com/blog/?page_id=11* (17 Oct, 2011).

[18] Y. A. Reznik, *et al.* Efficient fixed-point approximations of the 8x8 inverse discrete cosine transform. *Proc. SPIE*, **6696**, 669617 (2007): 1–17.

[19] W. Cai and M. Adjouadi. An Efficient Approach of Fast Motion Estimation and Compensation in Wavelet Domain Video Compression. *Proc. IEEE Int. Conf. Proc. Acoustics Speech, and Signal Processing*, **2**(May) (2004): II-277–II-280.

Appendix A

# Appendix

| Lossless compression | | | | | |
|---|---|---|---|---|---|
| | wavelet RGB | wavelet YUV | wavelet YUV2k | JPEG2000 | PNG |
| House | 1.81 | 1.55 | 1.87 | 1.90 | 1.77 |
| Baboon | 1.27 | 1.16 | 1.33 | 1.33 | 1.25 |
| Lena | 1.75 | 1.50 | 1.79 | 1.77 | 1.65 |
| Big Tree | 1.85 | 1.49 | 1.86 | 1.86 | 1.72 |
| Desktop | 2.96 | 3.15 | 3.74 | 4.41 | 4.34 |

**Table A.1:** Comparison of different methods for lossless compression

| House | | | | | |
|---|---|---|---|---|---|
| Compression | | | PSNR | | |
| wavelet | JPEG2000 | JPEG | wavelet | JPEG2000 | JPEG |
| 5.03 | 5.03 | 4.31 | 45.30 | 45.38 | 44.31 |
| 10.31 | 10.04 | 9.82 | 41.45 | 42.02 | 40.92 |
| 20.15 | 20.19 | 19.59 | 38.79 | 39.36 | 38.19 |
| 40.07 | 40.08 | 39.65 | 36.28 | 36.94 | 34.96 |
| 80.02 | 80.54 | 77.01 | 33.63 | 34.22 | 31.76 |

**Table A.2:** PSNR comparison for image House

| **Baboon** | | | | | |
|---|---|---|---|---|---|
| Compression | | | PSNR | | |
| wavelet | JPEG2000 | JPEG | wavelet | JPEG2000 | JPEG |
| 4.98 | 5.00 | 4.89 | 35.50 | 36.08 | 34.03 |
| 9.96 | 10.01 | 9.90 | 31.42 | 32.27 | 30.95 |
| 20.00 | 20.04 | 19.52 | 28.96 | 29.68 | 28.66 |
| 39.96 | 40.02 | 38.49 | 27.01 | 27.63 | 26.89 |
| 79.91 | 80.06 | 72.50 | 25.52 | 25.97 | 25.32 |

**Table A.3:** PSNR comparison for image Baboon

| **Lena** | | | | | |
|---|---|---|---|---|---|
| Compression | | | PSNR | | |
| wavelet | JPEG2000 | JPEG | wavelet | JPEG2000 | JPEG |
| 5.00 | 5.01 | 4.86 | 43.99 | 43.74 | 42.11 |
| 9.99 | 10.01 | 9.68 | 40.51 | 40.98 | 40.02 |
| 20.00 | 20.02 | 19.97 | 38.23 | 39.05 | 37.96 |
| 39.85 | 40.02 | 39.93 | 36.12 | 37.06 | 35.93 |
| 79.91 | 80.85 | 77.40 | 33.98 | 34.89 | 33.18 |

**Table A.4:** PSNR comparison for image Lena

| **Big Tree** | | | | | |
|---|---|---|---|---|---|
| Compression | | | PSNR | | |
| wavelet | JPEG2000 | JPEG | wavelet | JPEG2000 | JPEG |
| 4.91 | 5.00 | 4.86 | 39.79 | 44.41 | 37.27 |
| 9.98 | 10.00 | 9.68 | 35.57 | 40.86 | 34.48 |
| 19.98 | 20.00 | 19.97 | 32.69 | 38.33 | 32.17 |
| 40.48 | 40.00 | 39.93 | 30.71 | 36.45 | 30.62 |
| 80.08 | 80.00 | 77.40 | 29.35 | 34.87 | 29.12 |

**Table A.5:** PSNR comparison for image Big Tree

| **Desktop** | | | | | |
|---|---|---|---|---|---|
| Compression | | | PSNR | | |
| wavelet | JPEG2000 | JPEG | wavelet | JPEG2000 | JPEG |
| 5.08 | 5.00 | 4.86 | 52.80 | 58.47 | 37.32 |
| 10.04 | 10.01 | 9.68 | 46.81 | 48.86 | 36.98 |
| 20.02 | 20.02 | 19.97 | 39.41 | 40.89 | 34.81 |
| 40.01 | 40.13 | 39.93 | 33.34 | 34.94 | 31.45 |
| 79.98 | 80.00 | 77.40 | 29.23 | 30.73 | 28.23 |

**Table A.6:** PSNR comparison for image Desktop

| Foreman H.264 | | | | |
|---|---|---|---|---|
| FPS | Compression | PSNR (min) | PSNR (avg) | PSNR (max) |
| 30 | 44.62 | 33.67 | 35.94 | 38.46 |
| 15 | 33.98 | 34.74 | 36.36 | 37.96 |
| 10 | 29.78 | 34.80 | 36.05 | 38.61 |

**Table A.7:** Foreman video compressed using H.264

| Foreman Wavelet | | | | |
|---|---|---|---|---|
| FPS | Compression | PSNR (min) | PSNR (avg) | PSNR (max) |
| 30 | 30.60 | 33.04 | 35.51 | 38.39 |
| 15 | 30.56 | 33.06 | 35.51 | 38.39 |
| 10 | 30.14 | 33.06 | 35.52 | 38.24 |

**Table A.8:** Foreman video compressed using proposed *method 1*

| Desktop | | | | |
|---|---|---|---|---|
| Method | Compression | PSNR (min) | PSNR (avg) | PSNR (max) |
| H.264 15 fps | 179.1 | 27.24 | 30.61 | 36.22 |
| Wavelet 15 fps | 188.2 | 37.85 | 39.01 | 40.65 |

**Table A.9:** Screen capture video compressed using H.264 and proposed *method 2*