Video Coding with Adaptive Vector Quantization and Rate Distortion Optimization

DISSERTATION

zur Erlangung des Doktorgrades der Fakultät für Angewandte Wissenschaften der Albert-Ludwigs-Universität Freiburg im Breisgau

von

Marcel Wagner

Dekan: Prof. Dr. Gerald Urban

Referenten: Prof. Dr. Dietmar Saupe, Prof. Dr. James Fowler

Datum der Promotion: 2.10.2000

Für meine Eltern

Acknowledgment

In the first place I would like to express my deep gratitude to my advisor Prof. Dietmar Saupe. He was a constant source of support to me and it is his advice and encouragement which made this dissertation possible.

I am also indebted to Prof. James Fowler who sacrificed his valuable time to review this dissertation. He traveled a long way from Mississippi, USA, to Freiburg, Germany, to take part in the examination committee.

I would like to thank Dr. Raouf Hamzaoui who influenced the initial direction of my research and introduced me quickly to scientific work.

I wish to express my deepest gratitude to Dr. Hannes Hartenstein who encouraged me during many fruitful discussions, guided my work with many helpful suggestions and provided an excellent working atmosphere.

Moreover, I would like to thank many friends and colleagues for their helpful comments and suggestions concerning various aspects of this dissertation. Matthias Zachmann helped me make my English more readable. Evi Malik accompanied this dissertation unrestingly from the first draft to the final version with many valuable suggestions. Björn Grohman, Lucia Vences and Kirstin Wiginton provided numerous helpful comments.

I am also indebted to Prof. Hans Burkhardt who generously integrated me into his work group during my final year at the University of Freiburg. I really appreciated the stimulating working atmosphere in his group as well as the encouragement by all the members of his staff. Especially, I would like to thank Eva Grünwald for her help.

I very much appreciated the support of Micronas GmbH, Freiburg. My special thanks in this respect go to Prof. Miodrag Temerinac.

Abstract

The object of this dissertation is to investigate rate-distortion optimization and to evaluate the prospects of adaptive vector quantization for digital video compression.

Rate-distortion optimization aims to improve compression performance using discrete optimization algorithms. We first describe and classify algorithms that have been developed in the literature to date. One algorithms is extended in order to make it generally applicable; the correctness of this new procedure is proven. Moreover, we compare the complexity of the aforesaid algorithms, first implementation-independent and then by run-time experiments. Finally, we propose a technique to speed up one of the aforementioned algorithms.

Adaptive vector quantization enables adaption to sources with unknown or non-stationary statistics. This feature is important for digital video data since the statistics of two subsequent frames is usually similar, but in the long run the general statistics of frames may change even if scene changes are neglected. We examine combinations of adaptive vector quantization with various state-of-the-art video compression techniques. First we present an adaptive vector quantization based codec that is able to encode and decode in real-time using current PC technology. This codec is rate-distortion optimized and adaptive vector quantization is applied in the wavelet transform domain. The organization of the wavelet coefficients is then made more efficient using adaptive partition techniques. Moreover, the main adaptability mechanism of adaptive vector quantization, the so-called codebook update, is studied. Finally, a combination of adaptive vector quantization and motion compensation is taken into consideration. We show that for very low bitrates adaptive vector quantization performs on prediction residual frames better or at least as well as discrete cosine transform coding.

Zusammenfassung

In dieser Dissertation werden “rate-distortion” Optimierungsverfahren betrachtet und die adaptive Vektorquantisierung hinsichtlich der Einsatzmöglichkeit in der Videokompression untersucht.

Das Ziel der “rate-distortion”-Optimierung ist die Verbesserung von Kompressionsergebnissen unter der Verwendung von diskreten Optimierungsalgorithmen. Zunächst werden die in der Literatur behandelten Algorithmen beschrieben und klassifiziert. Einer dieser Algorithmen wird erweitert um eine allgemeine Verwendung zu ermöglichen. Die Korrektheit dieses neuen Verfahrens wird bewiesen. Zusätzlich wird die Komplexität der “rate-distortion” Optimierungsalgorithmen verglichen, sowohl Implementierungsunabhängig als auch mit Laufzeit-Experimenten. Schließlich wird ein Beschleunigungsverfahren für einen dieser Algorithmen vorgestellt.

Die adaptive Vektorquantisierung ermöglicht es, sich an Quellen mit unbekannter oder nicht-stationärer Statistik anzupassen. Gerade für digitale Videodaten ist diese Eigenschaft wichtig, da aufeinanderfolgende Bilder in einer Bildsequenz zwar eine ähnliche Statistik aufweisen, langfristig jedoch Änderungen in der Statistik zu erwarten sind, selbst wenn Szenenwechsel unberücksichtigt bleiben. In dieser Arbeit untersuchen wir die Kombination der adaptiven Vektorquantisierung mit verschiedenen “state-of-the-art” Videokompressions-Techniken. Zunächst stellen wir einen auf der adaptiven Vektorquantisierung basierenden Codec vor mit der Fähigkeit auf einem PC mit derzeitiger Standardtechnologie in Echtzeit zu kodieren und zu dekodieren. Dieser Codec wird “rate-distortion”-optimiert und die adaptive Vektorquantisierung wird auf das Bild der Wavelet-Transformation angewendet. Die Wavelet-Koeffizienten werden anschließend mit Hilfe einer adaptiven Partitionierungstechnik effizienter organisiert. Darüber hinaus untersuchen wir den wesentlichen Adaptivitätsmechanismus der adaptiven Vektorquantisierung, die sogenannte Kodebuch-Aktualisierung. Abschließend betrachten wir die Kombination der adaptiven Vektorquantisierung mit Bewegungskompensation.

Wir zeigen, daß die adaptive Vektorquantisierung für sehr niedrige Bitraten auf Prädiktionsfehlerbildern bessere oder mindestens genauso gute Ergebnisse erzielt wie eine Transformationskodierung mit der diskreten Kosinus-Transformation.

Contents

Acronyms and Notation

AVQ	Adaptive Vector Quantization
DCT	Discrete Cosine Transform
DPCM	Differential Pulse Code Modulation
ECVQ	Entropy Constrained Vector Quantization
FLC	Fixed Length Code
GLA	Generalized Lloyd Algorithm
ICCH	Incremental Computation of the Convex Hull
LCH	Lower part of the Convex Hull
LM	Lagrangian Multiplier
MC	Motion Compensation
MRVQ	Mean-Removed Vector Quantization
MSE	Mean-Squared-Error
PSNR	Peak-Signal-to-Noise Ratio
PTSVQ	Pruned Tree-Structured Vector Quantization
QCIF	Quarter Common Intermediate Format
QT	Quad-Tree
RD	Rate-Distortion
RLC	Run-Length-Coding
SNR	Signal-to-Noise Ratio
TSVQ	Tree-Structured Vector Quantization
VLC	Variable Length Code
VQ	Vector Quantization

$\| \cdot \|_2$	Euclidean norm
$\mathbf{1}_L$	Vector $(1, \dots, 1)^t$ with L components
$\langle \cdot, \cdot \rangle$	Inner product
$\mathcal{T}$, $\mathcal{S}$	Trees
$\tilde{\mathcal{T}}$, $\tilde{\mathcal{S}}$	Leaves of the trees $\mathcal{T}$ and $\mathcal{S}$
$\mathcal{S} \sqsubset \mathcal{T}$	$\mathcal{S}$ is subtree of $\mathcal{T}$
$\mathcal{S} \preceq \mathcal{T}$	$\mathcal{S}$ is pruned subtree of $\mathcal{T}$
α	Vector/scalar encoder
β	Vector/scalar decoder
γ	Index coder
$\|\gamma(i)\|$	Length of variable-length-code $\gamma(i)$
$\mathcal{C}$	Codebook
$\mathcal{C}^*$	Universal codebook
$\mathcal{F}$	Forgetting set
$\mathcal{L}$	Learning set
$\mathcal{V}$	Set of variable-length-codewords
Q	Quantization $Q = \beta \circ \alpha$
$\mathcal{P}_m$, $\mathcal{P}_t$	Set of local rate-distortion points
$\mathcal{P}$, $\mathcal{P}_{\mathcal{T}}$	Set of global rate-distortion points

Preface

The objective of data compression is to minimize the number of bits needed for a given representation of data. We are concerned with *lossy* data compression if this representation is only an approximation of the data. The main issue of this kind of data compression is the trade-off between the approximation error, called *distortion*, and the number of bits needed for the representation, called *rate*.

Theoretically, this problem has already been solved for a general class of data by *rate-distortion (RD) theory*. This theory reveals that only one universal technique, called *vector quantization*, can be used to achieve the optimal RD trade-off. In vector quantization, consecutive data elements are bundled and subsequently approximated by one of several predetermined vectors. The rate is given by the address of the vector and the distortion is derived from approximating the data.

This technique has several drawbacks in practice if applied to concrete data. The statistics of the data must be known in advance and must not change during the compression. In addition, the optimal RD trade-off can only be achieved under unrealistic assumptions, e.g., arbitrary dimension and number of representation vectors. Finally, this technique is quite complex. Thus, many other techniques that achieved satisfactory results for practical data compression have been introduced. For example, in the case of digital video data the most frequently used techniques are motion compensation and transform coding.

Therefore, practical data compression demands not only a good data representation using the options of a single compression technique, but also a choice of combinations of several appropriate techniques. To obtain the optimal representation for concrete data once the compression techniques have been determined is an optimization problem since it is necessary to consider each option for each piece of data. This problem is apparently too complex to be treated with a brute force method. Suboptimal but fast methods could be applied instead. This research area, the combination of data compression and discrete optimization techniques, is called *operational rate-distortion theory* or *rate-distortion optimization*.

One aspect of this dissertation is the evaluation and extension of rate distortion optimization techniques. As we shall demonstrate, there are basically two rate-distortion optimization techniques that achieve identical results. These two techniques are presented for different optimization scenarios. One of the techniques is then extended in order to make it more generally applicable. We compare the complexity of the two techniques and contribute a speed up method.

As mentioned above, vector quantization is a theoretically optimal technique. This technique has practical drawbacks, however, since the data statistics must be known in advance and must not change. These statistical properties are not given for video data, however. Even though subsequent frames show similar statistical behavior, the statistical characteristic may change slightly with time. Thus, in this thesis, a more flexible vector quantization scheme, called *adaptive vector quantization (AVQ)*, that is able to adapt to the data statistics for digital video data is investigated. We combine this technique with state-of-the-art video compression techniques and analyze the resulting performance. In addition, we use RD-optimization techniques to improve the results.

In dealing with the aforementioned issues, this thesis will proceed as follows. Chapter 1 outlines video compression techniques as far as necessary for the understanding of this thesis. We introduce entropy coding, scalar quantization, transform coding, motion compensation and hybrid coding.

Chapter 2 is concerned with vector quantization. A general terminology is presented as well as common design procedures and structures of vector quantization. We describe the adaptive vector quantization terminology and give a short summary of work done in the field previously.

Chapter 3 deals with rate-distortion techniques. Two techniques, the Lagrangian multiplier (LM) algorithm and the incremental computation of the convex hull algorithm, are presented. For a specific optimization scenario, the *hierarchical dependence*, we contribute a new algorithm and prove that it is correct.

In Chapter 4 different RD optimized AVQ based codecs are presented and compared. The first codec is capable of encoding and decoding in real-time using an ad hoc rate control scheme. Subsequently, we apply RD-optimization to improve the rate-control scheme. This is followed by experiments with AVQ and wavelet transform as well as adaptive partition techniques. Lastly, we provide a study with motion compensation.

Chapter 5 evaluates the complexity of rate-distortion techniques. We define implementation-independent performance measures and compare the complexity of the LM- and the ICCH-algorithm. Additionally, a technique is developed to speed up the LM-algorithm.

Chapter 6 summarizes the results and presents our conclusions.

In this dissertation, the following contributions are made:

- We create a new terminology for hierarchical RD-optimization problems and detail the known techniques in this terminology.

- We extent the generalized BFOS algorithm in order to allow a general optimization. The generalized BFOS algorithm might be used to solve hierarchical structured rate allocation problems. This algorithm, however, imposes some restrictions on the hierarchical structure. We propose an extension that follows the BFOS algorithm but, nonetheless, makes it possible to optimize hierarchical structures without restrictions. In addition, we present a proof for the correctness of the algorithm.

- We provide experiments with several AVQ-based codecs. We present experiments with RD-optimization, codebook organization, variable-length coding, wavelet transform, adaptive partition techniques and motion compensation. We show the adaptability of our codec and compare these codecs with a state-of-the-art hybrid codec (H.263).

- We compare the RD-optimization techniques using an implementation-independent complexity measure and provide run-time experiments.
- We propose a speed up method for the LM-algorithm, analyze the performance of the new algorithm and provide run-time experiments.

Chapter 1

Data Compression and Video Coding

In this chapter, we will introduce basic algorithms and concepts that can be found in almost every image and image sequence compression scheme. The objective of this chapter is not to give a detailed and comprehensive overview of these methods. Rather we describe terms that will be referred to in the progress of this thesis. Readers not familiar with data and image compression may consult [5, 64, 55] for an introduction.

1.1 Entropy Coding

In this section we are concerned with *lossless* data compression. In contrast to *lossy* data compression, which approximates the data to achieve high compression ratios, data that has been compressed losslessly can be reconstructed exactly. First, we define the *entropy* that can be used to show the theoretical limit of lossless data compression. Then we show how close practical solutions can approach these limits. The results and definition in this section follow [28, 16].

Let there be a discrete random variable $\mathbf{X}$ with a finite alphabet, $\mathcal{A} = \{a_0, \dots, a_{N-1}\}$, consisting of N symbols, and the probability mass function (pmf) $p(a) = \Pr\{\mathbf{X} = a\}$, $a \in \mathcal{A}$.

The *entropy* $H(\mathbf{X})$ of a discrete random variable $\mathbf{X}$ is defined as

$$H(\mathbf{X}) = -\sum_{a \in \mathcal{A}} p(a) \log_2 p(a).$$

For the case that $p(a) = 0$, we use the convention $0 \log_2 0 = 0$. The entropy is measured in bits/symbol.

A *source code* is a mapping $\gamma : \mathcal{A} \rightarrow \{0,1\}^*$ from the alphabet A to the set $\{0,1\}^*$ of finite length strings consisting of the symbols $\{0,1\}$. With this mapping the set of *variable-length code (VLC) words* $\mathcal{V}$ can be defined as $\mathcal{V} = \{\gamma(a_0), \ldots, \gamma(a_{N-1})\}$.

An important class of source codes is the class of *prefix codes* which satisfy the condition for all $a \in \mathcal{V}$ that no a is the prefix of another word in $\mathcal{V}$.

Then we can define the *expected length* $l(\mathcal{V})$ by

$$l(\mathcal{V}) = \sum_{a \in \mathcal{A}} p(a) |\gamma(a)|$$

where $|\cdot|$ denotes the length of an VLC $\gamma(a)$, $a \in \mathcal{V}$.

A fundamental result is that the entropy is an lower bound of the expected length $l(\mathcal{V})$, i.e.,

$$H(\mathbf{X}) \leq l(\mathcal{V}). \tag{1.1}$$

The definition of $\mathcal{V}$ constrains the length of source codes to be integers. Furthermore, it is known that for $\mathcal{V}$ equality in (1.1) can only be achieved if the probabilities $p(a)$ are all powers of 2, i.e., $p(a) = 2^{-l_a}$, $\forall a \in \mathcal{A}$, $l_a \in \mathbb{N}$. For all other pmf p, the expected length is larger than the entropy. Therefore, it is reasonable to look for the source code that yields the closest possible expected length $l(\mathcal{V})$ to $H(\mathbf{X})$. The construction of such a source code was described by Huffman [39]. The expected length of this code is bounded by

$$H(\mathbf{X}) \leq l(\mathcal{V}) < H(\mathbf{X}) + 1. \tag{1.2}$$

We denote this kind of source codes by *Huffman codes*.

For a high entropy the bound in (1.2) is negligible but for small entropies, $H(\mathbf{X}) < 1$, the resulting performance of a Huffman code is not acceptable.

Thus, we need another entropy coding strategy that does not have the integer length constraint. This problem is overcome by merging M consecutive symbols to create a larger alphabet. However, this technique is too complicated for even small values of M.

Another method is *arithmetic coding* [94, 54]. In this approach, the unit interval $I_0 = [0, 1)$ is divided in N subintervals proportional to the probabilities of the symbols in $\mathcal{A}$. Each interval is assigned to a symbol $a \in \mathcal{A}$. Encoding a symbol means selecting the corresponding interval. The selected interval I_1 is then once more divided into N subintervals and encoding of a symbol again means selection of a corresponding interval and so forth. This process is illustrated in Fig. 1.1. It can be shown that arithmetic encoding achieves the entropy $H(\mathbf{X})$ provided that the encoder runs long enough and uses arbitrary precision for the arithmetic operations. Real implementations use a finite precision arithmetic as described in [54].

The pmf p of the symbols is often referred to as *statistical model*. If the model is not known in advance, it has to be adapted. This is usually done by counting the frequency of previously encoded symbols. For an arithmetic coder, the frequency counts have to be represented so as to rapidly determine the corresponding intervals. Thus, the frequency counts f are organized as cumulative frequencies f_c, i.e. $f_c(a_m) = \sum_{i=0}^{m-1} f(a_i)$. If we use adaptive arithmetic encoding in our experiments, we start with uniform cumulative frequencies that have been derived from $f(a) = 1, \forall a \in \mathcal{A}$. Each time a symbol has been encoded, the corresponding frequency count is incremented by 1. For our experiments we used the adaptive arithmetic coder from [20].

Finally, we introduce a coding technique that is only efficient for a specific kind of statistical models. Assume a binary alphabet $\mathcal{A} = \{ZERO, ONE\}$ with probability of one symbol, say ZERO, close to 1. Then the symbol ONE seldom appears in a sequence of symbols described by $\mathbf{X}$. In this case it could be efficient to bundle subsequent symbols ZERO and encode only the number of such symbols until the appearance of the next ONE. We denote this number of symbols as a *run* and this technique as *run-length coding (RLC)*. In this thesis, RLC is used to encode positions of some few blocks in a map mainly

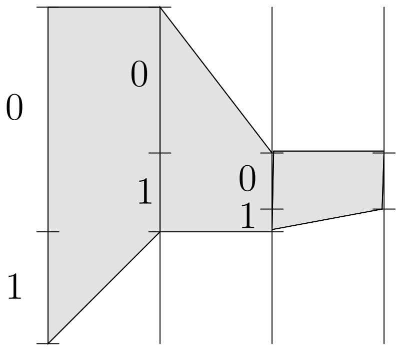


Figure 1.1: Arithmetic coding for the alphabet $\mathcal{A} = \{0, 1\}$ with pmf $p(0) = \frac{2}{3}$, $p(1) = \frac{1}{3}$. The sequence 010 is encoded resulting in the interval $(\frac{11}{27}, \frac{15}{27}]$. This can be represented by 100.

consisting of zeros (cf. Chapter 4). The RLC was empirically optimized for our codec. A run is encoded by the symbols $\$^*Y$ where $\$^*$ is a sequence of 0 or more *escape symbols* defining a run of 32 and $Y \in \{1, \dots, 32\}$ is a symbol describing the final run. This sequence is repeated so as to encode all runs. The symbols $\{1, \dots, 32, \$\}$ are entropy encoded by a Huffman coder or an adaptive arithmetic coder.

1.2 Scalar Quantization and Predictive Coding

Generally speaking, a *signal* is a continuous time and continuous amplitude function. In order to make it digitally processible it has to be converted. The first step of the conversion usually is the *sampling* to make the signal time-discrete. The second step is *quantization* to get discrete amplitude signals. Thus, the purpose of a quantization is to map a continuous amplitude to a discrete set of symbols.

Now we define scalar quantization. *Scalar quantization* consists of a quantizer map $Q : \mathbb{R} \to \mathcal{C}$ and the *codebook* or the set of *reproduction values* $\mathcal{C} = \{c_0, \dots, c_{N-1}\} \subset \mathbb{R}$ with N symbols. In some cases we need supplementary functions to describe various components of scalar quantization more detailed. The *scalar encoder* $\alpha : \mathbb{R} \to \mathcal{I}, x \mapsto i$ if $Q(x) = c_i$, with the index set $\mathcal{I} = \{0, \dots, N-1\}$, maps the real line to an index. The *scalar decoder* $\beta : \mathcal{I} \to \mathcal{C}$ maps an index to its reproduction value. Thus, we have $Q = \beta \circ \alpha$. In some instances it is necessary to encode the indices, e.g., by a variable length code (see Sect. 1.1). For this purpose, we define the *index coder* $\gamma : \mathcal{I} \to \mathcal{V}$, where $\mathcal{V}$ is a set of variable length codes.

The scalar decoder defines an order on reproduction values, $\beta(0), \dots, \beta(N-1)$; we use for example $\beta(0) \leq \cdots \leq \beta(N-1)$. Usually, the scalar encoder is realized by a nearest neighbor (NN) rule, $\alpha(x) = \arg\min_{i \in \mathcal{I}} \|x - \beta(i)\|_2^2$ with the Euclidean norm $\| \cdot \|_2$. For the case $\|x - \beta(i)\|_2^2 = \|x - \beta(j)\|_2^2$, $i \neq j$, we have to define a tie break rule, e.g., we take $\min\{i, j\}$. A NN-rule scalar quantizer partitioned the real line, $\mathbb{R}$, into N *partition cells* $\mathcal{R}_i = \{x \in \mathbb{R} : \alpha(x) = i\}$. Therefore, the partition cells can be described by an interval $[y_{i-1}, y_i)$ with the *boundary points* y_i, $-1 \leq i \leq N$. Thus, scalar quantization can be specified by $\mathcal{C}$ and y_i, $-1 \leq i \leq N$. The partition cell that contains 0, $\mathcal{R}_{\alpha(0)}$, is called *deadzone*. We denote scalar quantization by Q_s, α_s, β_s, etc., when it is necessary to distinguish the scalar quantization from other quantization schemes.

Without further information about the codebook structure, a NN-rule vector encoder α has to search the whole codebook to find a proper reproduction value. However, special codebook structures make the computation of the index $\alpha(x)$ more efficient. An example for this is *uniform* quantization, which is the most common quantization [28]. Here, the difference between adjacent reproduction values c_i as well as between adjacent boundary points[1] is a fixed value Δ, i.e., $c_i - c_{i-1} = \Delta$, $1 \leq i < N$ and $y_i - y_{i-1} = \Delta$, $1 \leq i < N$. Thus, the index $\alpha(x)$ can be computed with few operations. Even though uniform quantization is the simplest example of scalar quantization, this kind of quantization in conjunction with entropy coding (see Sect. 1.1) yields asymptotically optimal results, which means that for a fixed distortion the entropy is minimal over all scalar quantizer types [56]. A typical uniform quantizer

[1] The boundary points y_{-1} and y_N are excluded since they can be defined to be $-\infty$ and ∞ to cover the whole real line, $\mathbb{R}$.

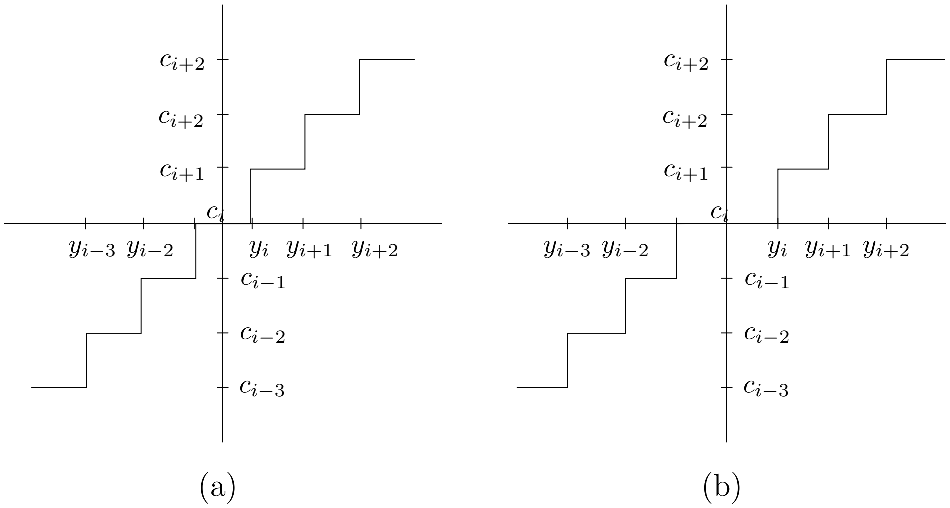


Figure 1.2: Uniform quantization: (a) with a deadzone equal to the other boundary intervals Δ, (b) with a deadzone different to the other boundary intervals.

is depicted in Fig. 1.2a. A uniform quantizer[2] with a deadzone differing from the other boundary intervals is presented in Fig. 1.2b. In this thesis, we only use uniform quantizers with deadzone for scalar quantization. Moreover, the deadzone is required to be centered around 0. Thus, in order to specify scalar quantization, the *binsize* Δ, the *binsize* of the deadzone, and the codebook size N have to be determined.

If consecutive samples are highly correlated, the encoding of the indices produced by scalar quantization could be done more efficiently with *predictive coding*. In predictive coding, previous samples are used to predict the current sample. Then only the error of the prediction is quantized and encoded. A special case is the *differential pulse code modulation* (DPCM) in which only the previous quantized sample is used to predict the current sample. This scheme is depicted in Fig. 1.3. In order to specify the DPCM coder we need only to describe the scalar quantization Q_d for the difference signals.

[2] Apparently, this quantization violates the definition of uniform quantization. Nevertheless, in this thesis, we mean this kind of quantization if we refer to uniform quantization.

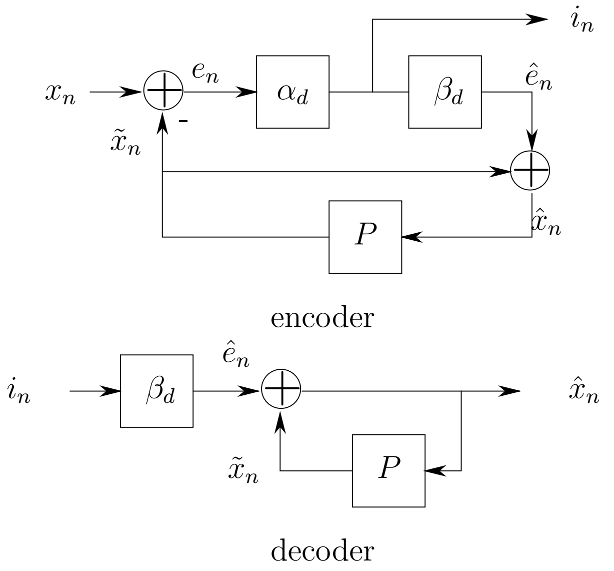
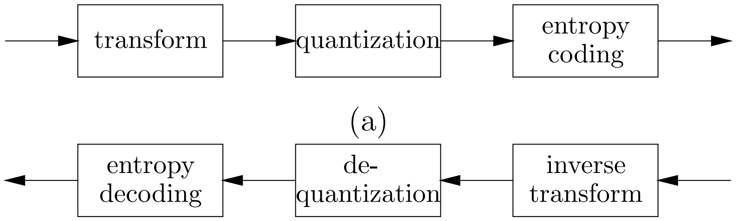


Figure 1.3: DPCM encoder/decoder



(a)

(b)

Figure 1.4: (a) Stages of transform coding and (b) the corresponding transform decoding.

1.3 Transform Coding

Transform coding is an important part of virtually every state-of-the-art image and video coding system. A typical transform coding scheme is shown in Fig. 1.4a. There are usually three stages. In the first stage, the source is transformed, then, in the transform domain, quantization is used, and, finally, the quantized transform coefficients are entropy encoded. The corresponding decoder is depicted in Fig. 1.4b.

The object of the transform stage is to achieve a representation of a signal that is suitable for a subsequent processing. An example of such a transform is the *Karhunen-Loeve transform (KLT)* which has the following properties [65]:

- It is orthogonal.
- It completely decorrelates the signal in the transform domain.
- It contains the most variance in the fewest number of transform coefficients and it makes possible a dimension reduction with the smallest possible mean squared error (MSE).

However, the KLT has several drawbacks, too. First, in order to perform an inverse transform the decoder must know the basis of the transform domain. This increases the amount of bits needed for compression. Secondly, there exists no fast algorithm to date that makes the KLT feasible for real-time applications. Thus, in image and image sequence coding schemes another transform is applied instead with similar properties, but with a predefined basis and with a fast transform algorithm, the *discrete cosine transform (DCT)*.

1.3.1 Discrete Cosine Transform

In order to apply the DCT, an image is decomposed into M blocks G_m, $0 \leq m < M$, of $N \times N$-pixel size. Each block is then transformed separately. The one dimensional discrete cosine transform is a linear transform that can be described by the matrix

$$(C_N)_{nm} = \left(\frac{2}{N}\right)^{1/2} \left[k_m \cos\left(\frac{m(n+\frac{1}{2})\pi}{N}\right)\right] \quad m, n = 0, \dots, N-1 \tag{1.3}$$

where

$$k_j = \begin{cases} 1, & \text{if } j \neq 0 \\ \frac{1}{\sqrt{2}}, & \text{otherwise.} \end{cases}$$

Thus, if we consider the block G_m as matrix, the two dimensional transform can be described by the multiplication $C_N^T G_m C_N$.

The transform coefficients are scalar quantized and afterwards run-length encoded using a zig-zag scan order as depicted in Fig. 1.5. The RLC coding is different to the scheme described in Sect. 1.1. For example, in the H.263 scheme

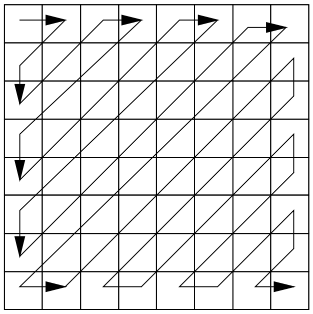


Figure 1.5: Zig-zag scan of the DCT transform coefficients.

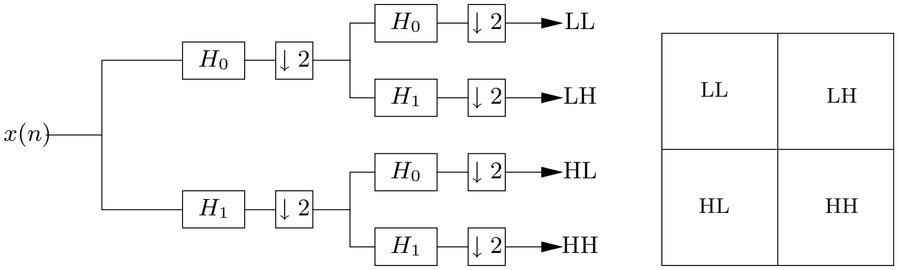
events are encoded consisting of a combination of (LAST, RUN, LEVEL). The value of LAST indicates whether there are more non-zero coefficients in the block or if this was the last, the value of RUN describes the number of zeros preceding the coded coefficient, and LEVEL the non-zero value of the coded coefficient.

The DCT as presented in (1.3) is known to be asymptotically equivalent to the KLT of a Markov-1 signal process with correlation coefficient ρ as $\rho \rightarrow 1$ [65]. Nevertheless, the application of the DCT on image blocks leads to "blocking artifacts" for high compression ratios. Therefore, in the next section, we introduce the *wavelet transform*. Wavelet transform decomposes an image not only into frequency components; the components also have a spatial localization.

1.3.2 Wavelet Transform

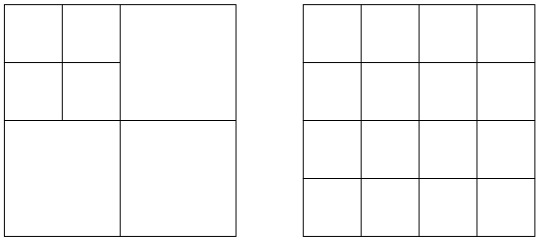
The details of wavelet transform can be found in [84, 17, 51]. Wavelet transform decomposes an image into *subbands* using a *lowpass* filter H_0 and a *highpass* filter H_1. The filter bank for a decomposition of a 2-dimensional signal is depicted in Fig. 1.6. Further decompositions of the subbands can be achieved by cascading the filter bank. A two times *uniform* decomposition and an *octave band* decomposition is shown in Fig. 1.7. General decompositions are called *wavelet packets* [12].

An evaluation of several wavelet filters for image compression can be found in [85]. In reference to this, we use the 9/7-filter developed by Antonini et. al.



(a) (b)

Figure 1.6: Filterbank for subband decomposition (a) and decomposed image (b).



(a) (b)

Figure 1.7: Types of subband decompositions: (a) octave band decomposition, (b) uniform decomposition

[2].

The subband coding methods can roughly be classified into *inter subband* and *crossband* techniques [14]. Inter subband techniques encode each subband separately. Crossband techniques exploit dependences between different subbands. The most popular crossband encoding technique is called *zero-tree* [74, 67]. Zero-trees are based on the observation that coefficients of highpass subbands are likely to be zero if certain coefficients with lower frequency corresponding to the same spatial localization are zero. A zero-tree structure of subband coefficients is presented in Fig. 1.8a. Another crossband coding technique was proposed in [91]. Each of the 16 subbands from a uniform subband decomposition contributes one coefficient to a 16-dimensional vector (Fig. 1.8b).

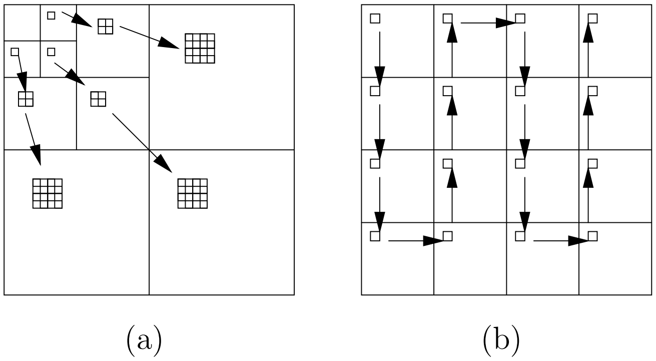


Figure 1.8: Zero-tree structure for crossband coding (a). Grouping of coefficients in a uniform subband decomposition (b).

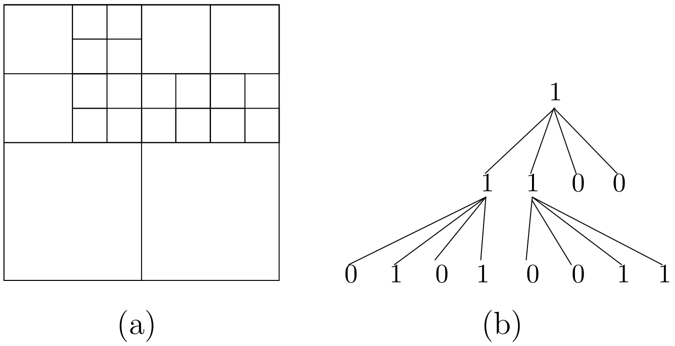


Figure 1.9: Quad-tree example: (a) representation of an image, (b) resulting tree for binary description. Description code: 1 1100 0101 0011

1.4 Image Partition Techniques

Different areas of an image can show different degrees of activity. For example, the background of an image can have a uniform structure and need not to be considered in detail whereas it can be useful to consider the boundary of foreground objects more thoroughly. One method to achieve this is the application of image partition techniques. An important example of such techniques is the *quad-tree* representation of an image. A quad-tree is a tree that contains either leaf nodes without a child node or internal nodes with four children. Thus, the tree structure can easily be described by a binary code. Figure 1.9 shows an example for a quad-tree representing an image and the corresponding binary code. For more details, the reader is referred to [68, 82, 76, 78, 79].

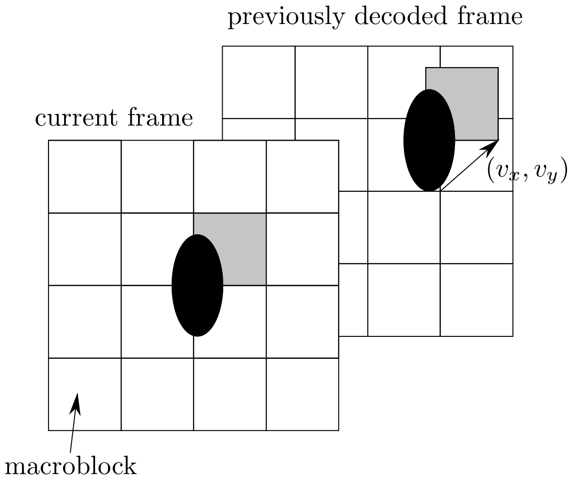


Figure 1.10: Block based motion compensation due to [64]

.

1.5 Motion Compensation

Subsequently, we will call one image from an image sequence a *frame*.

Motion within image sequences leads to a redundancy that can be removed to achieve a better compression ratio. This is usually made by *block motion compensation* (MC)[64]. Block MC creates a prediction of the current frame, called *motion prediction*, using blocks from the previously decoded frame. Figure 1.10 shows a typical scenario for block MC when an object (black ellipse) moves to another position. First, the frame is decomposed into macroblocks. Then, for a given macroblock x (shaded block in the current frame), a matching block $\hat{x}$ in the previous frame is searched (shaded block in the previously decoded frame). This block is used to create a prediction for the current macroblock. Since the decoder knows the previous frame, too, only a motion vector (v_x, v_y) is necessary to determine the prediction. Then only the error of the prediction, $e = x - \hat{x}$, has to be encoded.

The motion vector (v_x, v_y) is determined by the motion estimation procedure [64, 5].

1.6 Hybrid Coding

Two techniques, transform coding and motion compensation, are brought together in *hybrid coding*. Transform coding is applied to reduce redundancy within a frame. Motion compensation reduces the redundancy between several frames.

A hybrid coding scheme is depicted in Fig. 1.11. The encoding algorithm works as follows. First, a frame is decomposed into blocks of 16×16 pixel size called *macroblocks*. Then motion estimation is performed for each macroblock. The coding control decides whether a macroblock is encoded in *intra* or in *inter* mode. This is indicated in Fig. 1.11 with the dashed boxes. In intra mode, the macroblock is decomposed into four 8×8-pixel blocks. These blocks are transformed, quantized, and variable-length encoded. In the figure, this is indicated by T, Q and VLC. In inter mode, the macroblock is predicted from the previously decoded frame using an appropriate motion vector (indicated by P). The error of the prediction is decomposed into 8×8 pixel blocks, transformed, quantized, and variable-length encoded.

The encoder and decoder both have to decode the frame that is currently processed. This frame is used as *reference frame* for the motion prediction when the encoder is employed with the next frame. Thus, decoding is also part of the encoding process. The 8×8-pixel blocks are dequantized, and the inverse transform is applied. The four 8×8 block then are recomposed to macroblocks. If the encoding mode has been inter mode, the motion prediction is added to the decoded macroblock. The inverse transform stage is denoted by T^{-1}, dequantization by Q^{-1}, and the motion prediction and reference frame memory by P.

This scheme is used in all digital video coding standards for “natural” video. All standards have in common that they do not describe an encoding algorithm. The syntax and the semantics of the encoded bit stream rather are specified. The concrete encoding algorithm is left to the designers.

We summarize now the standards for digital video coding. The first video coding standard, H.261, was developed by the International Telecommunication Union (ITU) [41] and established in 1990. The target bitrate for the codec recommended by this standard is $p \cdot 64$ where $1 \leq p \leq 30$. A possible application is video conferencing over ISDN channels. The transform is made by a DCT. The scalar quantization of the transform coefficients is performed using a uniform quantizer with deadzone. Only the DC component in intra mode is separately quantized with a pure uniform quantizer. The motion vectors have integer-pel resolution.

The ISO (International Organization for Standardization) standard MPEG-1 was developed by the Moving Picture Expert Group and established in 1992. The target application of this standard is the coding of video and audio signals for digital storage media like CD-ROM at up to 1.5 Mbit/s [64]. Compared with H.261, the new features are *bidirectional interpolation* and *half-pel* resolution motion prediction. Bidirectional interpolation is a motion prediction technique that uses two frames, one taken from the past and one taken from the future. Half-pel resolution motion prediction allows half-pel resolution of the motion vectors. In order to achieve around 1.5 Mbit/s during the encoding, certain parameters of the MPEG-1 encoding are constrained. For example, the horizontal picture size has to be smaller than 768 pels, the vertical picture size smaller than 576 pels, and the maximal number of macroblocks per frame smaller than 396.

In 1994, MPEG-2, designed for digital video transmission, became an international standard [35, 64]. A bitrate in the range of 2 to 15 Mbit/s is supported. MPEG-2 mainly differs from MPEG-1 in that MPEG-2 is able to handle interlaced frames and has many scalability facilities. Scalability means that the frames can be decoded on various resolutions and quality levels. There are four scalability modes:

- spatial scalability
- SNR scalability
- temporal scalability
- data partitioning

The spatial scalability mode offers the decoder different spatial resolutions of the frames. In the SNR scalability mode, the quantization of transform coefficients can be refined. Temporal scalability makes possible a decoding at different frame rates. Data partitioning splits the bit stream into layers with different priorities. The bit stream with the highest priority contains the low frequency transform coefficient. The lower the priority of the layer, the higher is the frequency of the transform coefficients.

The purpose of the H.263 standard is encoding with very low bitrates ($<$ 64 kbit/s). Compared with H.261, the H.263 standard contains an improved motion compensation scheme and better entropy coding. The motion compensation can be made with half-pel resolution. The motion vectors may have a large range and cross the frame boundaries. In addition, the overlapped block motion estimation technique [43, 57] and bidirectional interpolation can be used for motion prediction. For entropy coding, it is possible to apply syntax based arithmetic coding. This technique uses predefined cumulative frequencies for different encoding contexts.

A further improvement of H.263 standards is called H.263+ and contains many new features like scalability modes and improved error resilience [15]. Currently, H.263++ and H.26L are under development. Techniques like long-term memory and multi-hypothesis MC are taken into consideration [93, 81, 21]

An ISO standard for multimedia needs has been established with MPEG-4. This standard covers a large number of encoding tools for digital video and audio. A scene can be composed with different audio visual objects using a source description language. Moreover, interactivity with the consumer has become possible. The conventional “natural” video coding part is similar to H.263. The most significant improvement is the capability to encode arbitrary boundary objects. In addition, a subband coding with wavelet transform is possible for frames that are encoded only in intra mode [18].

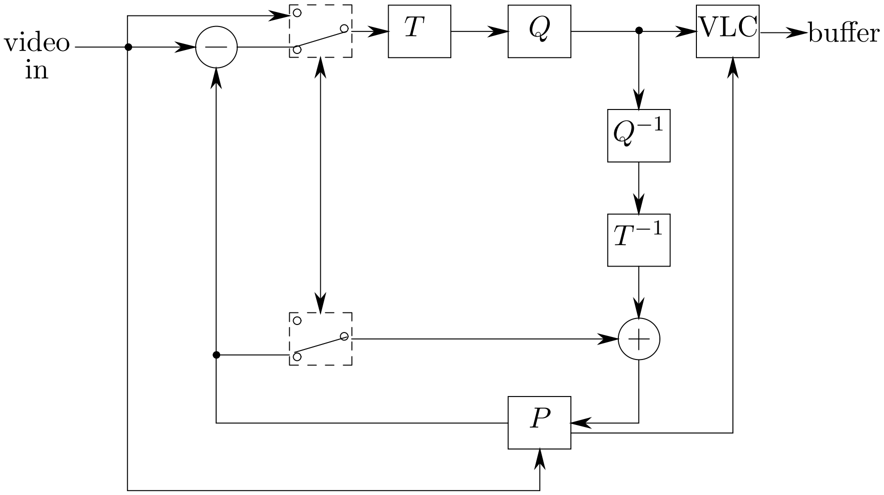


Figure 1.11: Hybrid coding adapted from [42].

Chapter 2

Vector Quantization

Vector quantization (VQ) was first considered by Shannon in his fundamental work [72, 73] and was used to show theoretical bounds of data compression. Shannon's idea was to bundle consecutive separate symbols of a source into vectors in order to represent the source in an efficient way.

The practical feasibility of VQ was studied in the late 1970s and the early 1980s. Only then, practical structures and design algorithms have been developed (see e.g. [48, 26]). Since that time, VQ was successfully applied for image, image sequence, and speech coding [29, 27, 32].

Figure 2.1 shows a basic VQ-system. The *encoder* reads a vector x from a source. Then the encoder tries to find a good representation $\hat{x} \in \mathcal{C}$ of the vector x where the set $\mathcal{C}$ of all possible representations is called the *codebook*. The address $i \in \mathcal{I}$ of the vector $\hat{x}$ in the codebook is transmitted to the *decoder*. With the transmitted index i, the decoder is able to determine $\hat{x}$ from the codebook $\mathcal{C}$. Note that this works only if encoder and decoder use the same codebook $\mathcal{C}$.

The basic VQ approach has the disadvantage that the statistical behavior of the source must be known in advance and that it is not able to deal with non stationary sources, i.e., sources that change their statistical behavior in time. Since the encoding and the decoding process demands identical codebooks for the encoder and decoder, the encoder must not vary the content of the codebook independently. The decoder rather must be informed to synchronize

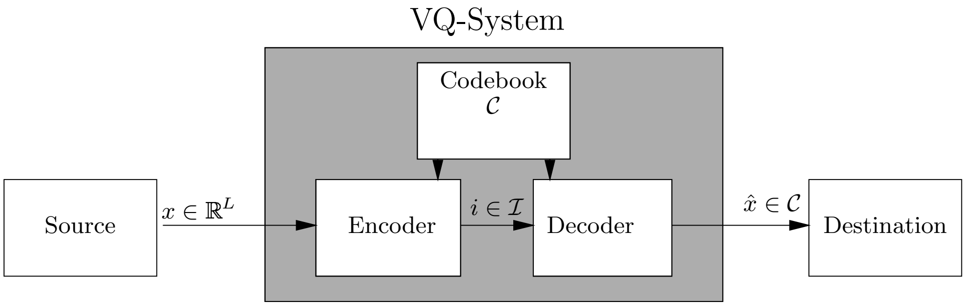


Figure 2.1: Vector quantization

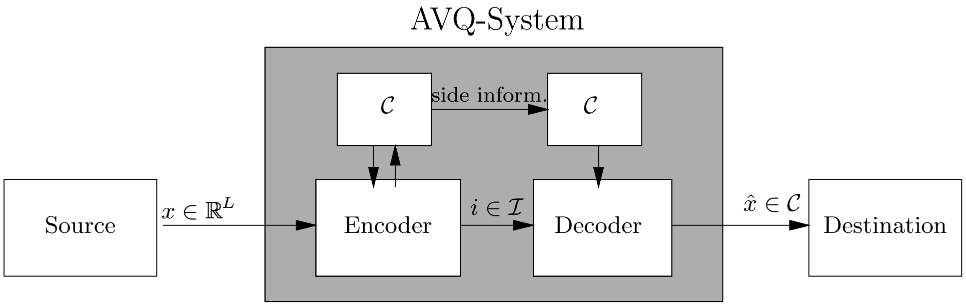


Figure 2.2: Adaptive vector quantization

its codebook. Thus, a special communication structure is needed to transmit changes of the codebook $\mathcal{C}$. The additional information to be transmitted is called *side information*. Figure 2.2 shows VQ with side information. Such kind of VQ is called *adaptive VQ* (AVQ).

In this chapter, we will describe VQ as detailed as it is necessary for the understanding of the following chapters. The notation we use is due to [28, 10, 22]. Readers should consult these references for further details.

First, we introduce in Sect. 2.1 a formal description of a VQ-system. Then, in Sect. 2.2 and 2.3, we describe common VQ design and structures in this notation. Finally, adaptive VQ is introduced, and an overview of work previously done in the field is given (Sect. 2.4).

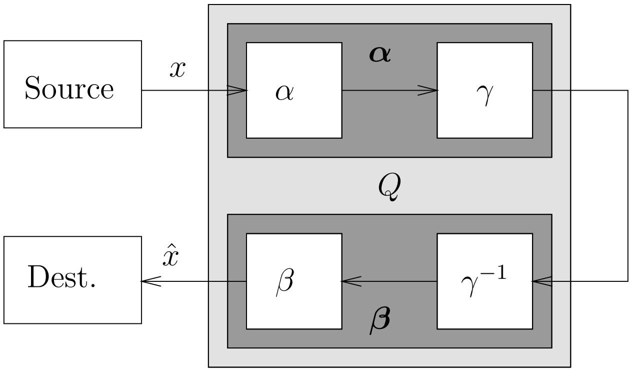


Figure 2.3: Formal scheme of vector quantization [10]

2.1 Basic Definitions and Properties

A (variable rate) VQ-system is based on a codebook $\mathcal{C} = \{c_0, ..., c_{N-1}\}$ with N vectors $c_i \in \mathbb{R}^L$, the index set $\mathcal{I} = \{0, ..., N-1\}$ and a variable-length-code (VLC) given by a set $\mathcal{V}$ of N codewords; the connection between these sets is specified by the *vector encoder* α, *vector decoder* β, and the bijective *index coder* γ where

- $\alpha : \mathbb{R}^L \to \mathcal{I}$ maps a vector to an index,
- $\beta : \mathcal{I} \to \mathcal{C}$ maps an index to a vector in the codebook $\mathcal{C}$, and
- $\gamma : \mathcal{I} \to \mathcal{V}$ maps an index to a VLC-word.

The encoder $\boldsymbol{\alpha} : \mathbb{R}^L \to \mathcal{V}$ is given by $\gamma \circ \alpha$ and the decoder $\boldsymbol{\beta} : \mathcal{V} \to \mathcal{C}$ by $\beta \circ \gamma^{-1}$. The quantization $Q : \mathbb{R}^L \to \mathcal{C}$ can be defined as $Q = \boldsymbol{\beta} \circ \boldsymbol{\alpha} = \beta \circ \gamma^{-1} \circ \gamma \circ \alpha = \beta \circ \alpha$. These terms are illustrated in Fig. 2.3.

In Sect. 2.2 we will show how to create a codebook $\mathcal{C}$, but in this section we assume that $\mathcal{C}$ does already exist. Vector decoder β then can be easily defined by ordering the vectors in $\mathcal{C}$ and mapping $\mathcal{I}$ to $\mathcal{C}$ according to this order.

To assess the distortion between $x = (x^{(0)}, \ldots, x^{(L-1)})^t$ and $Q(x) = \hat{x} = (\hat{x}^{(0)}, \ldots, \hat{x}^{(L-1)})^t$, a distortion measure $d(x, \hat{x})$ is necessary. Usually $d(\cdot, \cdot)$ is defined as $d(x, \hat{x}) = \|x - \hat{x}\|_2^2 = \sum_{i=0}^{L-1} (x^{(i)} - \hat{x}^{(i)})^2$.

Having defined a distortion measure, the vector encoder α often is realized by the nearest neighbor (NN) rule which selects the index of a vector in $\mathcal{C}$ with the smallest distortion, $\alpha(x) = \arg\min_{i \in \mathcal{I}} d(x, \beta(i))$. A tie break rule has to be applied in case the minimum is achieved for two or more indices.

In order to evaluate the performance of a vector quantizer, we consider vectors x described by a random variable $\mathbf{X}$ and the corresponding probability density function (pdf) $f_{\mathbf{X}}$. The average distortion D and the average rate R can be expressed as

$$D = E[d(\mathbf{X}, Q(\mathbf{X}))] = \int_{\mathbb{R}^L} d(x, Q(x)) f_{\mathbf{X}}(x) dx \tag{2.1}$$

and

$$R = E[|\gamma(\alpha(\mathbf{X}))|] = \int_{\mathbb{R}^L} |\gamma(\alpha(x))| f_{\mathbf{X}}(x) dx \tag{2.2}$$

where $|\gamma(i)|$ is the length of the VLC $\gamma(i)$ in bits and E is the expectation operator. The vector encoder partitions the space $\mathbb{R}^L$ into N *partition cells* $\mathcal{R}_i = \{x \in \mathbb{R}^L : \alpha(x) = i\}$. Each of the partition cells has a *partition probability* p_i and a *partition distortion* d_i defined as

$$p_i = prob(\mathbf{X} \in \mathcal{R}_i) = \int_{\mathcal{R}_i} f_{\mathbf{X}}(x) dx \tag{2.3}$$

and

$$d_i = E[d(\mathbf{X}, Q(\mathbf{X})) : \mathbf{X} \in \mathcal{R}_i] \tag{2.4}$$

With these terms we can define the *partial distortion*

$$D_i = d_i \cdot p_i. \tag{2.5}$$

VQ is, theoretically, the best encoding method for a general class of sources. In fact, it is shown in [4] that VQ achieves the asymptotically optimal result for the encoding of stationary and ergodic random processes when providing arbitrary codebook size and vector dimension.

2.2 Codebook Design

The performance of a vector quantizer is mainly determined by the quality of the codebook $\mathcal{C}$. A codebook that does not match the statistical characteristics of the source cannot be compensated substantially by other VQ parameters like α, β, and γ. Thus, designing a codebook is of crucial importance and has been the subject of thorough investigations in the literature. It is known that creating a codebook that minimizes the average distortion, D, is a NP-complete problem [25]. Therefore, less complex but suboptimal iterative design algorithms have been developed [48, 8, 28, 49, 66, 10]. The most famous algorithm for codebook design is the *generalized Lloyd algorithm* (GLA). This algorithm was popularized by Linde, Buzo and Gray [48] and a generalization of the idea published by Lloyd in 1957 [49, 50]. Another technique to determine the number of codebook vectors is the *splitting* method [48].

2.2.1 The generalized Lloyd algorithm

In order to outline the GLA, we need two optimality conditions for the vector quantizer.

Proposition 1 (Nearest Neighbor Condition [28]) *For a given codebook $\mathcal{C}$, the optimal partition cells $\mathcal{R}_i$ satisfy $\mathcal{R}_i \subset \{x : d(x, \beta(i)) \leq d(x, \beta(j)), \forall j \in \mathcal{I}\}$.*

Proof: [28, p. 350f]. □

Proposition 2 (Centroid Condition [28]) *For a given partition $\{\mathcal{R}_i : i \in \mathcal{I}\}$, the optimal codebook vectors satisfy $\beta(i) = cent(\mathcal{R}_i)$ where cent($\mathcal{R}$)$= E[\mathbf{X} : \mathbf{X} \in \mathcal{R}]$ with the expectation operator E.*

Proof: [28, p. 352]. □

In Proposition 1, a condition for the vector encoder α is formulated assuming a fixed vector decoder β, and in Proposition 2, a condition for the vector decoder β is formulated assuming a fixed vector encoder α. This suggests the following iteration step: Starting with a fixed codebook $\mathcal{C}_n$, the vector encoder α_{n+1} is defining the partition cells $\mathcal{R}_i$ by the nearest neighbor rule. Then a new codebook $\mathcal{C}_{n+1}$ can be computed by defining $\beta_{n+1}(i) = \text{cent}(\mathcal{R}_i)$. This iteration step is called the *Lloyd iteration*.

The above form of the Lloyd iteration assumes that the pdf $f_{\mathbf{X}}(x)$ is known and the boundaries of the partition cells $\mathcal{R}_i$ can be analytically determined. In general, however, this is impossible. The probability distribution is rather given by an empirical set of training vectors $\mathcal{B}$. Thus, the centroid computation and the computation of $\mathcal{R}_i$ in the Lloyd iteration can be substituted by discrete computations,

$$\mathcal{R}_i = \{x \in \mathcal{B} : d(x, \beta(i)) \leq d(x, \beta(j)),\ \forall j \in \mathcal{I}\} \tag{2.6}$$

and

$$\text{cent}(\mathcal{R}_i) = \frac{1}{|\mathcal{R}_i|} \sum_{x \in \mathcal{R}_i} x, \tag{2.7}$$

where $|\cdot|$ denotes the set cardinality. For the partition distortions d_i, partition probabilities p_i, and partial distortion D_i, we can proceed in an analogous way.

It can be shown that the average distortion D cannot increase after one Lloyd iteration. Therefore, subsequent applications of the Lloyd iteration creates a sequence of codebooks $\mathcal{C}_m$ with corresponding non-increasing average distortions D_m. This iterative algorithm is called the *generalized Lloyd algorithm* (GLA). It stops if the change of the average distortions D_m is small enough, i.e., $\frac{D_m - D_{m-1}}{D_m} < \epsilon$ (Alg. 1). This always happens for a finite training set $\mathcal{B}$ after a finite number of steps. Furthermore, for this case, it can be shown that the sequence of distortions D_m converges to a local minimum.

In order to let the sequence of distortions converge to a reasonably small local minimum, the initialization of the GLA, i.e., the selection of the first codebook $\mathcal{C}_0$, is crucial. The simplest initialization technique is a random selection of N codebook vectors from the test set $\mathcal{B}$. Advanced techniques use clustering algorithms to create the initial codebook [28, 19].

The GLA optimizes only the average distortion D and does not take into consideration the average rate R. Moreover, it assumes that the index coder is a fixed length coder, i.e., $|\gamma(n)| = \lceil \log_2 N \rceil,\ \forall n$. Thus, the average rate does not vary during the GLA. If optimizing the average rate R is an objective, too, one is concerned with *entropy constrained vector quantization (ECVQ)*. To apply this algorithm, the trade-off between the rate and the distortion must

Algorithm 1 The generalized Lloyd algorithm

```
Given:  initial codebook $\mathcal{C}_0$,
        vector decoder $\beta_0$,
        training set $\mathcal{B}$,
        stopping threshold $\epsilon$,
        average distortion $D_0$ of init. vector quantizer,
        $m = 0$
repeat
  $\alpha_{m+1}(x) \overset{\text{def}}{=} \arg\min_n d(x, \beta_m(n))$
  $\mathcal{R}_n \leftarrow \{x \in \mathcal{B} : \alpha_{m+1}(x) = n\},\ \forall n \in \mathcal{I}$
  $\beta_{m+1}(n) \leftarrow \frac{1}{|\mathcal{R}_n|} \sum_{x \in \mathcal{R}_n} x,\ \forall n \in \mathcal{I}$
  $\mathcal{C}_{m+1} \leftarrow \{\beta_{m+1}(n) : n \in \mathcal{I}\}$
  $D_{m+1} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{x \in \mathcal{B}} d(x, \beta_{m+1}(\alpha_{m+1}(x)))$
  $m \leftarrow m + 1$
until $\frac{D_m - D_{m-1}}{D_m} < \epsilon$
```

be specified by a parameter λ. The interpretation of λ is discussed in detail in Chapter 3.

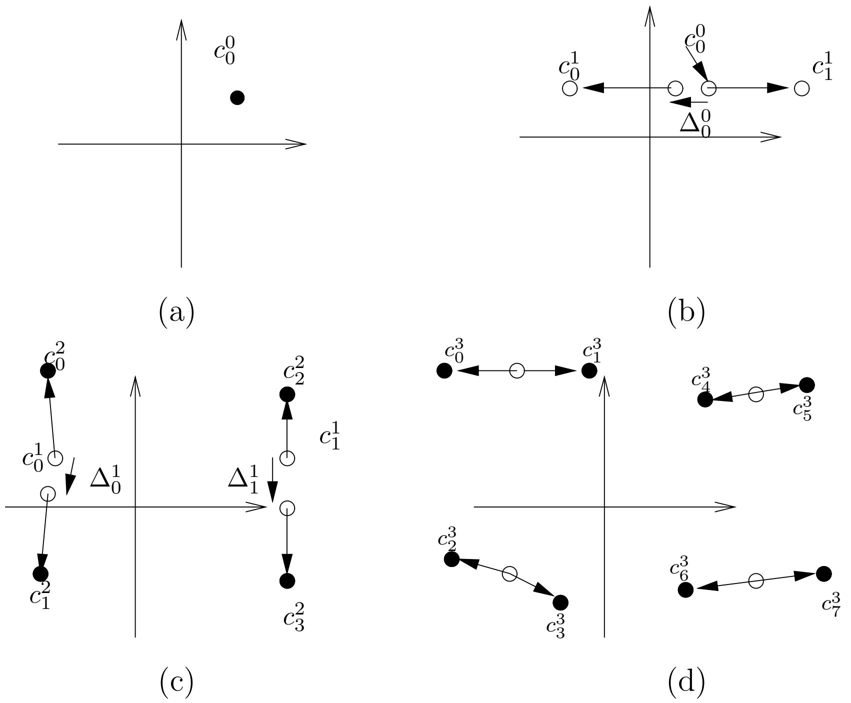
The codebook design for entropy constrained vector quantization is very similar to the GLA with two main differences. First, the vector encoder α has to take the index coder γ into account. Secondly, the index coder also has to be estimated within a Lloyd iteration (see Alg. 2).

2.2.2 Codebook Splitting

The GLA has the disadvantage that the number of code vectors N must be fixed in advance. A technique to create codebooks growing successively larger has been introduced by Linde et al. [48]. In this technique, the first codebook $\mathcal{C}^0 = \{c_0^0\}$ contains only a single vector, namely, c_0^0 =cent($\mathcal{B}$). This vector is then "split" by a small vector, Δ, into $\{c_0^0, c_0^0 + \Delta_0^0\}$ and the GLA is applied resulting in $\mathcal{C}^1$. The new codebook, $\mathcal{C}^1 = \{c_0^1, c_1^1\}$, is "split" to $\{c_0^1, c_0^1 + \Delta_0^1, c_1^1, c_1^1 + \Delta_1^1\}$ and $\mathcal{C}^2$ is created by the GLA etc. The splitting algorithm is depicted in Fig. 2.4.

Algorithm 2 The generalized Lloyd algorithm for ECVQ

```
Given: initial codebook $\mathcal{C}_0$,
       vector decoder $\beta_0$, index coder $\gamma_0$,
       training set $\mathcal{B}$,
       stopping threshold $\epsilon$,
       average distortion $D_0$ and rate $R_0$ of init. vector quantizer,
       trade-off parameter $\lambda$
       $m = 0$
repeat
  $\alpha_{m+1}(x) \overset{\text{def}}{=} \arg\min_n d(x, \beta_m(n)) + \lambda \cdot |\gamma_m(n)|$
  $\mathcal{R}_n \leftarrow \{x \in \mathcal{B} : \alpha_{m+1}(x) = n\}, \forall n \in \mathcal{I}$
  $\beta_{m+1}(n) \leftarrow \frac{1}{|\mathcal{R}_n|} \sum_{x \in \mathcal{R}_n} x$
  $|\gamma_{m+1}(n)| \leftarrow -\log_2 \frac{|\mathcal{R}_n|}{|\mathcal{B}|}$
  $\mathcal{C}_{m+1} \leftarrow \{\beta_{m+1}(n) : n \in \mathcal{I}\}$
  $D_{m+1} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{x \in \mathcal{B}} d(x, \beta_{m+1}(\alpha_{m+1}(x)))$
  $R_{m+1} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{x \in \mathcal{B}} |\gamma_{m+1}(\alpha(x))|$
  $m \leftarrow m + 1$
until $\frac{J_{m+1} - Jm}{J_{m+1}} < \epsilon$, where $J_m = D_m + \lambda \cdot R_m$
```



(a) (b)

(c) (d)

Figure 2.4: The splitting algorithm. A sequence of codebooks, $\mathcal{C}^0 = \{c_0^0\}$, $\mathcal{C}^1 = \{c_0^1, c_1^1\}$, $\mathcal{C}^2 = \{c_0^2, c_1^2, c_2^2, c_3^2\}$, and $\mathcal{C}^3 = \{c_0^3, c_1^3, c_2^3, c_3^3, c_4^3, c_5^3, c_6^3, c_7^3\}$, is created.

This procedure creates a sequence of codebooks, $\mathcal{C}^0, \mathcal{C}^1, \ldots$, doubling the codebook size with each step. The algorithm stops when the desired number of code vectors has been achieved or the average distortion D is small enough.

2.3 Structured Vector Quantization

The function α is used to map any desired vector, $x \in \mathbb{R}^L$, to an index, $i \in \mathcal{I}$. Typically, α implements a NN rule. The index i is selected for which the minimal distortion $d(x, \beta(i))$ is found. A brute force method to determine the desired index i is to compute $d(x, \beta(i))$ for all vectors in the codebook $\mathcal{C}$. However, for large codebook size N and high vector dimension L this approach is too complex. Therefore, structures have been developed to speed up searching the codebook.

2.3.1 Tree Structured Vector Quantization

A structure that is inspired by the splitting method for the codebook design (Sect. 2.2) is the tree-structured VQ (TSVQ). The index $\alpha(x)$ for a given vector x is computed in several stages. In each stage, some codebook vectors of $\mathcal{C}$ are excluded from the search. Only after the last stage it is known which index and which vector in $\mathcal{C}$ has been selected.

We illustrate this in Fig. 2.4. Let us assume $\mathcal{C} = \mathcal{C}^3$. In order to determine the NN of x in $\mathcal{C}$, we compute the distortions $d(x, c_0^1)$ and $d(x, c_1^1)$ of vectors in codebook $\mathcal{C}^1$ in the first stage. Let $d(x, c_0^1)$ be smaller than $d(x, c_1^1)$. In the second stage we consider the vectors in $\mathcal{C}^2$ but only those vectors that have been created by splitting the vector c_0^1 with the smallest distortion in the first stage, i.e., c_0^2 and c_1^2. Let $d(x, c_0^2)$ be smaller than $d(x, c_1^2)$. Then only the split vectors of the vector c_0^2 with the smallest distortion, c_0^3 and c_1^3, are considered. The comparison of the distortions $d(x, c_0^3)$ and $d(x, c_1^3)$ determines the desired vector in $\mathcal{C}$. Therefore, $\alpha(x)$ can be found within $\log_2 N$ stages and in each stage only the distortions of two vectors have to be compared. Apparently, this can be described by a tree (see Fig. 2.5). The search strategy described above, defines a path from the root to one of the leaves. The leaf nodes represent the codebook $\mathcal{C} = \mathcal{C}^3$. All vectors c_n^l of level l in the tree are contained in the *intermediate codebook* $\mathcal{C}^l$. Note that the design procedure for the intermediate codebooks as well as the final codebook, $\mathcal{C}$, in this case slightly differs from the splitting method. In the original splitting method, when the codebook $\mathcal{C}^K$ is created, α_K is given by the nearest neighbor rule and the intermediate codebooks $\{\mathcal{C}^k : k < K\}$ are not taken into account. The design algorithm for the TSVQ, however, applies the vector coder α_K defined by the algorithm described above using vectors from the intermediate codebooks. Thus, the resulting codebook vector $\beta_K(\alpha_K(x))$ is not necessarily the NN from x in $\mathcal{C}^K$ [28].

Since every node in the tree has only two successors, the index coder γ for a TSVQ can be realized as binary code by encoding the path from the root to the desired leaf. This, however, leads to a fixed length code. In order to design a VLC, the interpretation of the codebooks, $\mathcal{C}^0, \mathcal{C}^1, \ldots,$ as a tree structure in

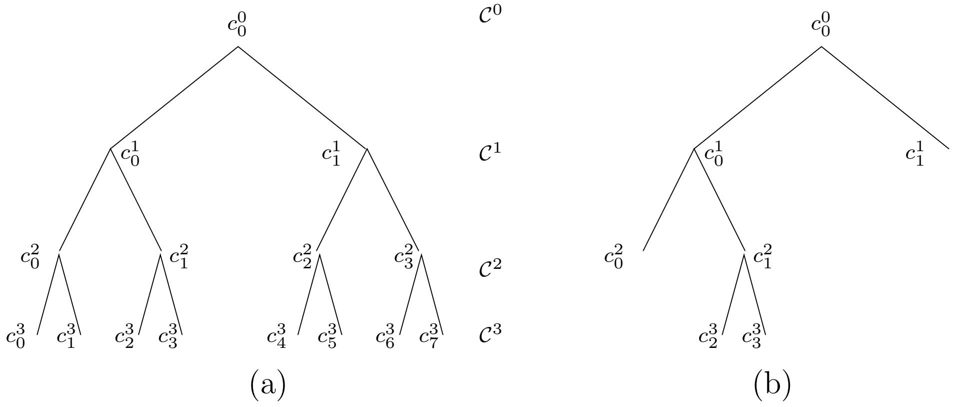


Figure 2.5: Tree-structured vector quantization. (a) Codebook of the full tree $\mathcal{C} = \{c_0^3, c_1^3, c_2^3, c_3^3, c_4^3, c_5^3, c_6^3, c_7^3\}$, (b) pruned tree-structured VQ $\mathcal{C} = \{c_0^2, c_2^3, c_3^3, c_1^1\}$

Fig. 2.5 can be helpful. If some branches are pruned, the length of the path from the root to several leaves is variable. Thus, the index coder, γ, induced by the pruned tree, no longer represents a fixed-length code (FLC). This leads to the so called *pruned tree-structured VQ* (PTSVQ). These tree structures are very important for the progress of the subsequent chapters. Thus, we introduce suitable notation for trees that is due to [6, 11]. It is essential to note that the following definitions of tree-structures are unconventional. For example, we do not formalize the fact that the nodes of a tree are related by edges. However, this relation should be clear from the context. A classical definition of tree structures can be found in [61, 53].

We define a *tree* $\mathcal{T}$ as a finite set of nodes $\mathcal{T} = \{t_0, t_1, \dots\}$ with the root t_0. The *leaves* of $\mathcal{T}$ are denoted by $\tilde{\mathcal{T}}$. Nodes that are not leaves are called *internal* nodes[1]. We say $\mathcal{S}$ is a *subtree* of $\mathcal{T}$ ($\mathcal{S} \sqsubset \mathcal{T}$) if $\mathcal{S} = \{s_0, s_1, \dots\} \subset \mathcal{T}$. $\mathcal{S}$ is a *pruned subtree* ($\mathcal{S} \preceq \mathcal{T}$) of $\mathcal{T}$ if $\mathcal{S} \sqsubset \mathcal{T}$ and $s_0 = t_0$. A *branch* $\mathcal{S}_t$ of $\mathcal{T}$ is a subtree of $\mathcal{T}$ with root t and $\tilde{\mathcal{S}}_t \subset \tilde{\mathcal{T}}$.

With these terms it is possible to describe basic tree structures and relations between trees. An example is shown in Fig. 2.6. The full tree is $\mathcal{T} = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6\}$. The pruned subtree $\mathcal{S} = \{t_0, t_1, t_2, t_3, t_4\} \preceq \mathcal{T}$ is described by the light shape. The branch $\mathcal{S}_{t_2} \sqsubset \mathcal{T}$ is defined by $\{t_2, t_5, t_6\}$

[1] We assume that all internal nodes have the same number of children.

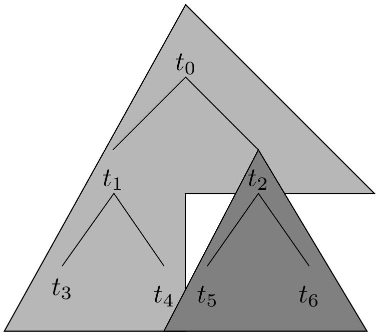


Figure 2.6: Example for the PTSVQ terms.

(dark shape). The leaves are given by $\tilde{\mathcal{T}} = \{t_3, t_4, t_5, t_6\}$, $\tilde{\mathcal{S}} = \{t_3, t_4, t_2\}$ and $\tilde{\mathcal{S}}_{t_2} = \{t_5, t_6\}$.

In order to describe the performance of the PTSVQ we need performance measures for tree structures. Therefore, we introduce *tree functionals.*

Let u be a map defined on each subtree of $\mathcal{T}$, $u : \mathcal{U}_{\mathcal{T}} \to \mathbb{R}$, $\mathcal{U}_{\mathcal{T}} = \{\mathcal{S} : \mathcal{S} \sqsubset \mathcal{T}\}$, then we call u a *tree functional*. In the following we define some important properties of tree-functionals.

Definition 1 *A tree functional is called* affine *if* $\forall \mathcal{S} \in \mathcal{U}_{\mathcal{T}}$ *there is the decomposition property* $\forall \mathcal{R} \preceq \mathcal{S} : u(\mathcal{S}) = u(\mathcal{R}) + \sum_{t \in \tilde{\mathcal{R}}} \Delta u(\mathcal{S}_t)$ *where* $\Delta u(\mathcal{S}_t) = u(\mathcal{S}_t) - u(t)$ *(*$u(t)$ *is short hand for* $u(\{t\})$*, a tree consisting of the single node* t*) and* $\mathcal{S}_t$ *is branch of* $\mathcal{S}$*.*

Definition 2 *A functional* u *is denoted as* monotonically increasing *if* $\mathcal{S} \sqsubset \mathcal{T} \Rightarrow u(\mathcal{S}) \leq u(\mathcal{T})$*. In an analogous way a* monotonically decreasing *functional can be defined.*

A tree $\mathcal{T}$ is *pruned* at node t if all offsprings of t are removed in order to make t a leaf.

We now introduce two examples of tree functionals that arise from PTSVQ. As in Sect. 2.1, a probability distribution for vectors is assumed. Then the vector encoder $\alpha_{\mathcal{S}}$, the vector decoder $\beta_{\mathcal{S}}$, the index coder $\gamma_{\mathcal{S}}$, and the quantizer $Q_{\mathcal{S}}$ of PTSVQ are dependent on the tree structure $\mathcal{S}$. According to (2.1) and

(2.2), we have

$$\delta(\mathcal{S}) = D(\mathcal{S}) = E\left[d(\mathbf{X}, Q_{\mathcal{S}}(\mathbf{X}))\right] = \int_{\mathbb{R}^L} d(x, Q_{\mathcal{S}}(x)) f_{\mathbf{X}}(x) dx \tag{2.8}$$

and

$$l(\mathcal{S}) = R(\mathcal{S}) = E\left[|\gamma_{\mathcal{S}}(\alpha_{\mathcal{S}}(\mathbf{X}))|\right] = \int_{\mathbb{R}^L} |\gamma_{\mathcal{S}}(\alpha_{\mathcal{S}}(x))| f_{\mathbf{X}}(x) dx. \tag{2.9}$$

The maps δ and l are tree functionals [11]. The properties of these tree functionals are summarized by the following lemma.

Lemma 1 *The tree functionals δ and l are affine. Moreover, δ is monotonically decreasing and l is monotonically increasing.*

Proof: see [11]. □

Since each pruned subtree $\mathcal{S} \preceq \mathcal{T}$ defines a different VQ-system with different rate and distortion, one can optimize the average rate-distortion trade-off by selecting an appropriate tree $\mathcal{S}$. Fixing the target rate l_T we seek the subtree $\mathcal{S}^* \preceq \mathcal{T}$ that satisfies $l(\mathcal{S}^*) \leq l_T$ and that leads to the minimum distortion $\delta(\mathcal{S}^*)$, i.e.,

$$\mathcal{S}^* = \arg \min_{\mathcal{S} \preceq \mathcal{T} : l(\mathcal{S}) \leq l_T} \delta(\mathcal{S}). \tag{2.10}$$

Computing $\mathcal{S}^*$ is a rate-distortion optimization problem. This issue is thoroughly discussed in Chapter 3.

2.3.2 Mean-Removed VQ

If the statistical mean of the components of vectors is different from zero or is varying widely, then the codebook structure can be organized more efficiently. Instead of using one large codebook, the codebook size can be reduced by treating the mean of a vector separately and quantizing only vectors with a zero mean.

A typical mean-removed vector quantization (MRVQ) structure is depicted in Fig. 2.7a. First, the mean μ of vector x is computed. Then the *mean-removed residual* of x, $r = x - \mu \mathbf{1}_L$, where $\mathbf{1}_L$ describes the vector $(1, \ldots, 1)^t$

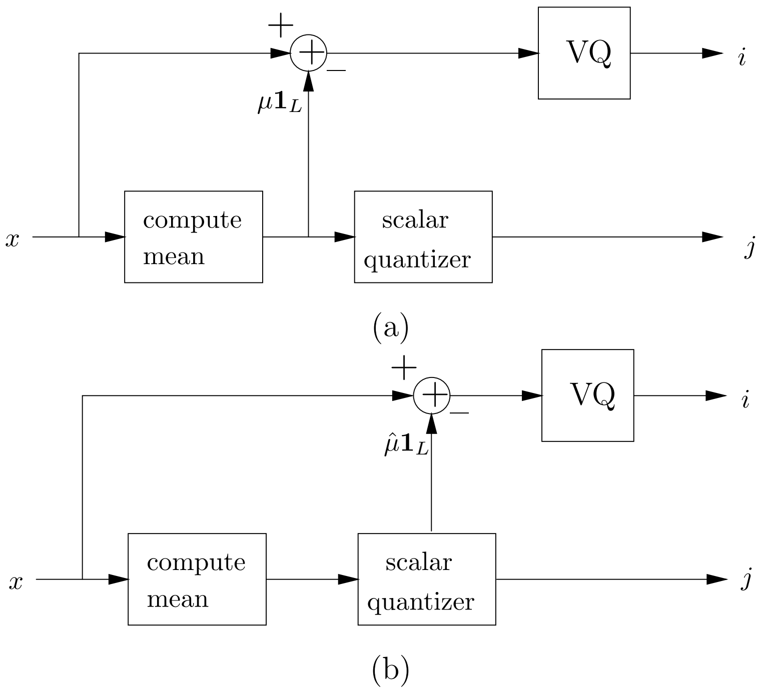


Figure 2.7: Mean-removed vector quantization.

with L components, is passed to the vector quantizer and mapped to an index i by the vector encoder α. Scalar quantization (cf. Sect. 1.2) is employed to quantize the mean, μ, and uses the scalar encoder α_s to produce index j. The decoding is done by $\hat{x} = Q_s(\mu)\mathbf{1}_L + Q(r)$. In this structure, both quantizers act independently. The application of the scalar quantizer does not influence the vector encoding of the vector quantizer and vice versa. In addition, all vectors r are located in a hyper space of $\mathbb{R}^L$ because of $\langle r, \mathbf{1}_L \rangle = 0$, where $\langle \cdot, \cdot \rangle$ denotes the inner product.

Another variant of MRVQ is shown in Fig. 2.7b. Unlike the former approach, the mean-removed residual is computed using the quantized mean, i.e., $r = x - \hat{\mu}\mathbf{1}_L$ with $\hat{\mu} = Q_s(\mu)$. Thus, the vector quantizer is no longer independent of the scalar quantizer. However, it can be shown that this variant leads to a smaller average distortion than the previous one [28, p. 438f]. Therefore, if we refer in this thesis to MRVQ we think of the MRVQ scheme depicted in Fig. 2.7b. Note that in this case the mean-removed residuals r do not lie in a linear subspace of $\mathbb{R}^L$. They rather lie in a "narrow slice" of $\mathbb{R}^L$

that can be characterized by [28]

$$\{x \in \mathbb{R}^L : |\sum_{i=0}^{L-1} x^{(i)}| < L\delta\}$$

where δ is the maximum quantizing error of the scalar quantizer Q_s.

2.3.3 Product Code VQ

In MRVQ, the vector quantizer has two parts: the scalar quantizer and the mean-removed residual vector quantizer. This structure reduces the search complexity significantly. We do not need to search all combination of means and mean-reduced residuals. We only need to search the codebooks of both quantizers separately. Thus, the complexity of one task is reduced by its decomposition into smaller sub-tasks. This idea is generalized by *product code VQ*. We do not introduce product code VQ in its full generality. This can be found in [28]. We rather define it as far as it is of use to this thesis.

A product code VQ for independent quantizers as defined in [28] is shown in Fig. 2.8. The vector x is decomposed into P component vectors, $u_i : 0 \leq i < P$, by the functions f_i. Then the component vectors are fed in independent quantizers Q_i. The quantized component vectors $\hat{u}_i$ are used by the function g to produce the reconstruction $\hat{x}$.

The simplest product code VQ is the partition of a vector into several subvectors. For the case $P = 2$ and the vector $x = (x^{(0)}, \ldots, x^{(L-1)})^t$, we can define

$$f_0 : \mathbb{R}^L \to \mathbb{R}^K, (x^{(0)}, \ldots, x^{(L-1)})^t \mapsto (x^{(0)}, \ldots, x^{(K-1)})^t$$

and

$$f_1 : \mathbb{R}^L \to \mathbb{R}^{L-K}, (x^{(0)}, \ldots, x^{(L-1)})^t \mapsto (x^{(K)}, \ldots, x^{(L-1)})^t$$

with $K < L$. This kind of product code VQ is often referred to as *partitioned VQ* [28].

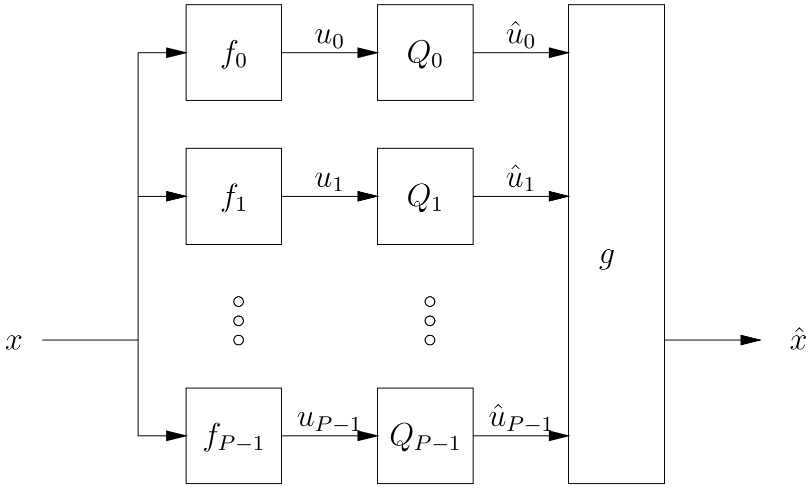


Figure 2.8: Product code VQ for independent quantizers due to [28].

2.4 Adaptive Vector Quantization

In Sect. 2.1, we defined a vector quantizer that can be used for the encoding of vectors produced by a stationary and ergodic random process. In that case, the vector quantizer was theoretically optimal, provided that the codebook size and vector dimension are sufficiently large [4] and the probability density function of the random process is known in advance. However, if the random process is not stationary, e.g., the statistics vary slightly in time, or the probability density function is not known in advance, a structure is desirable that can adapt to new statistics. Thus, we need a more general coding scheme. The terminology of this section is due to [28, 22, 46] and has been, when necessary, slightly extended or restricted.

Adaptability is provided by an adaptive vector quantization (AVQ) scheme. In this scheme, all parameters of VQ are time dependent. Unlike the typical VQ scheme, we not only have to encode vectors from a random process. We also do have to encode the information for the change of the parameters. This information is called *side information*.

Particularly, the contents of the codebook $\mathcal{C}_t$ changes in time and this change has to be dealt with specifically. New vectors have to be inserted and old ones have to be removed. In order to make the encoding of the side information efficient, a "super-codebook" $\mathcal{C}^*$ must exist to constrain the vectors that can be inserted in the codebook $\mathcal{C}_t$. Therefore, the codebook $\mathcal{C}_t$ must always be a subset of $\mathcal{C}^*$. The definition of AVQ will be formalized in the following.

An adaptive VQ-system is defined by a (local) codebook $\mathcal{C}_t$ containing N_t vectors with dimension L_t, an index set $\mathcal{I}_t$ and a VLC given by the set $\mathcal{V}_t$; similar to basic VQ the connection between these sets is specified by the vector encoder α_t, vector decoder β_t and the index coder γ_t. The quantization Q_t is specified as in the case of basic VQ, $Q_t = \beta_t \circ \alpha_t$.

In addition, a *universal codebook* $\mathcal{C}^*$ is defined with $\forall t : \mathcal{C}_t \subset \mathcal{C}^*$. To make the description of the vectors in $\mathcal{C}^*$ easier we also introduce a *universal vector encoder* α^*, *universal vector decoder* β^*, *universal index coder* γ^*, the *universal quantization* Q^*, and the *universal set of VLC-words* $\mathcal{V}^*$. The universal vector encoder can be used to find a proper vector that is inserted in the codebook $\mathcal{C}_t$. The universal vector decoder reconstructs this vector and the universal index coder transmits this vector to the decoder and, thus, produces the side information. Note that, unlike [22], this is a restriction of the set cardinality of $\mathcal{C}^*$. The universal codebook $\mathcal{C}^*$ either is be finite or countable infinite.

In order to describe the change of the codebook $\mathcal{C}_t$ at time t, we define the *learning set* $\mathcal{L}_t = \mathcal{C}_t - \mathcal{C}_{t-1}$ and the *forgetting set* $\mathcal{F}_t = \mathcal{C}_{t-1} - \mathcal{C}_t$. Thus, we can *update* the codebook $\mathcal{C}_t$ at time t by $\mathcal{C}_t = \mathcal{C}_{t-1} \cup \mathcal{L}_t - \mathcal{F}_t$.

For ease of notation we drop the index t when it is not essential.

2.5 Adaptive Vector Quantization Algorithms

The purpose of this section is to give an overview over previously published adaptive vector quantization algorithms. A taxonomy of AVQ was introduced by Fowler [22]. Three general types of AVQ algorithms were classified in [22], namely, constrained-distortion, constrained-rate, and rate-distortion based AVQ algorithms.

2.5.1 Constrained-Distortion AVQ Algorithms

Constrained-distortion AVQ algorithms usually employ a distortion threshold $D_{\max}$ to decide whether the codebook $\mathcal{C}_t$ should be updated or not. Presently, we outline the Paul algorithm and the Wang-Shende-Sayood algorithm.

The Paul Algorithm

In the Paul algorithm (move-to-front variation) [22], β_t is organized in a way that makes it possible to determine the least recently used (LRU) vector, that is, the vectors are sorted with respect to their last use, $\beta_t(0)$ is the recently used vector and $\beta_t(N-1)$ is the LRU vector. Furthermore, α_t is a NN vector coder, $\alpha_t(x) = \arg\min_{i \in \mathcal{I}} d(x, \beta_t(i))$, and γ_t is a variable length coder with the shortest codes for small indices and the longer codes for higher indices. This enables efficient encoding of the recently used vectors.

Now we describe the encoding procedure for a vector x_t. If $d(Q_t(x_t), x_t) \leq D_{\max}$ then the index $\alpha_t(x)$ is encoded by γ_t together with a flag signaling that the codebook is not updated, i.e., $\mathcal{C}_{t+1} \leftarrow \mathcal{C}_t$.

Otherwise, the codebook is updated. The universal codebook $\mathcal{C}^*$ contains all possible source vectors. Thus, the vector that is sent to the decoder and inserted in the codebook is the vector x_t itself. Therefore, Q^* must satisfy $Q^*(x) = x, \forall x$, and the index coder γ^* must be able to encode an index for every source vector. In addition, a flag is sent signaling the codebook update. The learning set is $\mathcal{L}_{t+1} = \{x_t\}$ and the forgetting set is $\mathcal{F}_{t+1} = \{\beta_t(N-1)\}$.

After all, the vector $Q_t(x_t)$ or $Q^*(x_t)$, respectively, are moved to the front of the codebook, i.e., $\beta_{t+1}(0) = Q_t(x_t)$ or $\beta_{t+1}(0) = Q^*(x_t)$, and the vector decoder β is rearranged to maintain the aforementioned LRU order.

Wang-Shende-Sayood Algorithm

The Wang-Shende-Sayood algorithm [22, 89] is very similar to the Paul algorithm. The main difference is the organization of the universal codebook realized by using a lattice [13, 28]. The universal vector coder α^* determines the nearest neighbor of a vector x in the lattice.

2.5.2 Constrained-Rate AVQ Algorithms

Constrained-distortion AVQ algorithms use a distortion threshold $D_{\max}$ to decide whether the codebook should be updated or not. Consequently, it is not possible to predict the encoding rate for an interval of vectors in advance since the number of codebook updates is unknown and, along with it, the rate of side information.

In constrained-rate AVQ algorithms, the number of codebook updates is fixed for a given *adaption interval* of vectors $x_t, x_{t+1}, \ldots, x_{t+\tau-1}$ where τ is the interval length. Therefore, the rate cannot exceed a predetermined target rate.

The constrained-rate AVQ algorithms can roughly be divided into two subclasses: The *local context* and the *codebook retraining* algorithms.

Local context algorithms compute several measurements on the vectors of the adaption interval. This measurement is then used to select the codebook $\mathcal{C}_t$.

In the second class, the codebook-retraining algorithms apply techniques from non-adaptive VQ to adapt the codebook $\mathcal{C}_t$ to changing statistics. Typical approaches use the GLA [30, 33, 34]. Other methods use learning techniques like *Kohonen learning* [44] as proposed by Lancini et al. [45].

Here, we will describe only two algorithms based on the GLA in more detail, namely, the algorithms proposed by Gersho and Yano [33, 34] and by Goldberg and Sun [30].

The Gersho-Yano algorithm splits its codebook vectors to achieve equal partial distortions as far as possible. The Goldberg-Sun algorithm uses the mth adaption interval as test data and runs several iterations of the GLA on codebook $\mathcal{C}_{m-1}$ to create $\mathcal{C}_m$.

Gain-Adaption Algorithm

Gain-adaption VQ [28] is an example of a local context algorithm. In this approach, the average gain, $g = \frac{1}{\tau}\sum_{i=0}^{\tau-1} \|x_{t+i}\|_2^2$, of the vectors in the adaption interval is computed and quantized to $\hat{g}$. Then, the vectors of the adaption interval are divided by $\hat{g}$. Subsequently, the "normalized" vectors are quantized using a static codebook $\mathcal{C}$. The procedure is repeated with the next adaption interval. As side information we have only to transmit the quantized gain $\hat{g}$.

Since the codebook $\mathcal{C}$ does not change during the encoding, it is, at first sight, not clear why the gain-adaption approach is considered as AVQ. But in [22] the reason for that is revealed. A universal codebook can be created by multiplying all vector in $\mathcal{C}$ with all possible quantized gains $\hat{g}$. Then we can interpret the gain as a measurement for the codebook selection.

The Gersho-Yano Algorithm

The Gersho-Yano algorithm [22, 31] is based on the *partial distortion theorem*. The proposition of this theorem is that for an asymptotic optimal vector quantizer the partial distortions D_i (see Sect. 2.1) are nearly constant with value $\frac{\sum_{i=0}^{N-1} D_i}{N}$ as N approaches infinity [28, p. 187].

Therefore, the Gersho-Yano algorithm tries to change the codebook vectors so as to make the partial distortions "as equal as possible". The basic version of the algorithm works as follows. First, consecutive vectors are partitioned into adaption intervals $\mathcal{B}_m = \{x_t, x_{t+1}, \ldots, x_{t+\tau-1}\}$ with the interval length τ and $t = \tau(m-1)+1$. Then for one adaption interval $\mathcal{B}_m$ the partitions $\mathcal{R}_i$ and the partial distortions D_i are computed using $\mathcal{C}_m$, β_m, and the NN vector encoder α_m. Let $i_{\max}$ and i_{min} be the index of the partition with the largest and smallest partial distortion, respectively.

The vector $c_{\max 1} = \beta_m(i_{\max})$ is "split" into two vectors, i.e., a new vector $c_{\max 2}$ is created by adding a small random vector to $c_{\max 1}$. Then one iteration of the GLA is applied with the training set $\mathcal{R}_{i_{\max}}$ and the initial codebook $\{c_{\max 1}, c_{\max 2}\}$. The two resulting codebook vectors $c'_{\max 1}$ and $c'_{\max 2}$ describe the learning set, $\mathcal{L}_{m+1} = \{c'_{\max 1}, c'_{\max 2}\}$. The forgetting set is given by $\mathcal{F}_{m+1} = \{\beta_m(i_{\max}), \beta_m(i_{\min})\}$.

As side information, the new vectors $c'_{\max 1}$ and $c'_{\max 2}$ are transmitted to the decoder using α^* and γ^*. In addition, the indices $i_{\min}$ and $i_{\max}$ are transmitted using the index coder γ_m.

Finally, all vectors in $\mathcal{B}_m$ are encoded using codebook $\mathcal{C}_{m+1}$ and m is incremented by 1.

The Goldberg-Sun Algorithm

The basic version of the Goldberg-Sun algorithm [34, 22] is another example for an adaption algorithm using the GLA. This time, one or more iterations of GLA are applied using the whole adaption interval $\mathcal{B}_m$ as training set and the codebook $\mathcal{C}_{m-1} = \{c_0, \dots, c_{N-1}\}$. The resulting codebook is denoted as $\mathcal{C}'_m = \{c'_0, \dots, c'_{N-1}\}$. Each codeword in $\mathcal{C}_{m-1}$ is compared with its corresponding codeword in $\mathcal{C}'_m$. The learning and the forgetting set are defined as $\mathcal{L}_m = \{c'_i : \|c_i - c'_i\|_2 > \delta\}$ and $\mathcal{F}_m = \{c_i : \|c_i - c'_i\|_2 > \delta\}$ with the predefined threshold δ.

As side information, we have to transmit a flag for every codebook vector c_i indicating whether the vector is updated or not. Then, where necessary, the code vector c'_i is encoded by γ^*.

Finally, the vectors in the mth adaption interval are vector quantized using the new codebook $\mathcal{C}_m$.

2.5.3 Rate-Distortion-Based AVQ Algorithm

The AVQ algorithms presented so far consider either the distortion or the rate to decide whether the codebook should be updated or not. Even though in general the constrained-distortion algorithms try to achieve a low encoding rate, too, and the constrained-rate algorithms also have the objective to achieve a low distortion, this trade-off is not considered explicitly. However, this is done by rate-distortion (RD) based AVQ. The distortion reduction ΔD by the improvement of the codebook and the rate increase ΔR by the side information are taken into account. The trade-off between these values is specified by the RD parameter λ. Typically, RD based AVQ minimizes $J = D + \lambda R$ with D being the overall distortion and R the overall rate including side information. The interpretation of the parameter λ is discussed more thoroughly in Chapter 3. In the literature, two essentially different algorithms are known [22]. A *batch* algorithm of Lightstone and Mitra [47, 46] and an *online* algorithm proposed by Fowler [24, 23].

The batch algorithm analyses an interval of vectors and sends the codebook update in an RD optimized fashion. Subsequently, the vectors are quantized.

The online algorithm decides "on the fly" if the current codebook should be updated or not.

Another algorithm proposed by Chen et al. can be considered as a combination of the Lightstone-Mitra and the Fowler approach. This algorithm decides "on the fly" whether the current vector x_t should be encoded by a vector in the codebook or the x_t itself should serve as a new codebook vector. After an adaption interval, a general codebook update, e.g., moving and deleting of vectors, is performed. All decisions are made in a RD-optimization sense.

However, in the following we consider only the two essentially different algorithm, the online and the batch algorithm.

The Lightstone-Mitra Algorithm

This approach [46, 22] assumes that both encoder and decoder own the codebook $\mathcal{C}_m$ that roughly suits the current source statistic. Then the encoder analyses an adaption interval, $\mathcal{B} = \{x_0, \ldots, x_{M-1}\}$, and if the codebook mismatches the source it is updated as follows. For each vector in the codebook, c_i, the distortion reduction, ΔD_i, by changing c_i and the rate increase, ΔR_i, for the corresponding side information are considered. Subsequently, for each vector in the codebook it is decided whether it should be changed or not.

This scheme is depicted in Alg. 3. First, one iteration of the ECVQ-GLA is applied. The new vector coder α_{m+1}, index coder γ_{m+1}, partition cells $\mathcal{R}_n$, and preliminary vector decoder β'_{m+1} are estimated. Note that, provided β_m is known to the decoder, the estimation of α_{m+1} does not require side information and the amount of side information for the VLC of γ_{m+1} is either negligible or zero using a Huffman coder or adaptive arithmetic coder, respectively [46]. The only substantial side information is needed for updating the codebook, i.e., specifying β_{m+1}. This side information is expressed in RD-terms with $\Delta J_i = \Delta D_i + \lambda \cdot \Delta R_i$ for every potentially new codebook vector c'_i. Only vectors are transmitted for which $\Delta J_i < 0$ holds true. Then the procedure is repeated until the stopping criteria is satisfied. After that, the vectors in $\mathcal{B}$ are quantized using the updated vector quantizer.

Algorithm 3 The Lightstone-Mitra algorithm

```
**Given:** initial codebook $\mathcal{C}_0$, vector decoder $\beta_0$, index coder $\gamma_0$,
           adaption interval $\mathcal{B} = \{x_0, \dots, x_{M-1}\}$,
           stopping threshold $\epsilon$,
           trade-off parameter $\lambda$,
           $m = 0$
**repeat**
    Let $\mathcal{C}_m = \{c_0, \dots, c_{N-1}\}$.
    Define $\alpha_{m+1}(x) \overset{\text{def}}{=} \arg\min_i[d(x, \beta_m(i)) + \lambda \cdot \gamma_m(i)]$
    $\mathcal{R}_n \leftarrow \{x \in \mathcal{B} : \alpha_{m+1}(x) = n\}, \forall n \in \mathcal{I}$
    Calculate a VLC that meets approximately $|\gamma_{m+1}(n)| = -\log_2 \frac{|\mathcal{R}_n|}{|\mathcal{B}|}$.
    Estimate the preliminary vector decoder, $\beta'_{m+1}(n) \leftarrow \frac{1}{|\mathcal{R}_n|} \sum_{x \in \mathcal{R}_n} x$.
    Quantize possible new codebook vectors, $c'_i = Q^*(\beta'_{m+1}(i))$.
    Compute potential distortion reduction for every $c'_i$, $\Delta D_i = \frac{1}{|\mathcal{R}_i|} \sum_{x \in \mathcal{R}_i} d(x, c'_i) - \frac{1}{|\mathcal{R}_i|} \sum_{x \in \mathcal{R}_i} d(x, c_i)$.
    Compute rate increase required by side information for every $c'_i$, $\Delta R_i = \frac{\gamma^*(\alpha^*(c'_i))}{|\mathcal{R}_i|}$.
    Compute RD measure for every $c'_i$, $\Delta J_i = \Delta D_i + \lambda \cdot \Delta R_i$.
    Create forgetting set $\mathcal{F}_{m+1} = \{c_i : \Delta J_i < 0\}$ and learning set $\mathcal{L}_t = \{c'_i : \Delta J_i < 0\}$; update $\mathcal{C}_{m+1}$ and $\beta_{m+1}$ accordingly.
    $m \leftarrow m + 1$
**until** the relative change of the cost function $J$ is small enough ($< \epsilon$).
Quantize the vectors in $\mathcal{B}$.
```

The Generalized Threshold Replenishment Algorithm of Fowler

One drawback of the batch algorithm described above is that the adaption interval $\mathcal{B}$ must be known in advance. The encoder must be able to look ahead in time. The online algorithm of Fowler et al. adapts its codebook considering each vector separately without knowledge about the future. This algorithms is called *generalized threshold replenishment (GTR)*.

In [23], two similar GTR versions are presented. Here, we describe only the, better performing, move-to-front variant. This approach is depicted in Alg. 4.

We describe the procedure for a given time t. It is assumed that a probability $p_{t-1}(i)$ is known for every codebook vector $c_i \in \mathcal{C}_{t-1}$. We start with estimating the index coder γ_t. Then the vector coder α_t is defined. In the next step, the cost of a codebook update is computed. If the update pays in the RD sense, the codebook is updated. Otherwise, the vector is quantized with $Q(x_t)$. The vector $Q^*(x_t)$ or $Q(x_t)$, respectively, is then moved to the front of the codebook.

The estimation of the new probability works as follows. First, we define a window parameter ω useful for the estimation of the number of vectors in the past that have been mapped to c_i, $\eta_t(i) = \omega p_{t-1}(i)$. Let j be described by $j = \alpha_t(x_t)$. If no codebook update occurred, the new probabilities are then estimated by

$$p_t(i) = \begin{cases} [\omega p_{t-1}(i)]/(\omega + 1), & i \neq j \\ [\omega p_{t-1}(i) + 1]/(\omega + 1), & i = j \end{cases}$$

Otherwise, the new probabilities are estimated by

$$\eta_t(i) = \begin{cases} \omega p_{t-1}(i), & i \neq j \\ \omega p_{t-1}(i)/2, & i = j \end{cases}$$

and

$$p_t(i) = \frac{\eta_t(i)}{\sum_{n=0}^{N-1} \eta_t(n)}$$

In either case, the indices of the probability function must be rearranged according to β_t.

Algorithm 4 The GTR algorithm (Move-to-front variant)

```
Given: codebook $\mathcal{C}_{t-1}$,
       vector decoder $\beta_{t-1}$,
       codeword probability, $p_{t-1}(i)$, for each codeword $c_i \in \mathcal{C}_{t-1}$,
       RD trade-off parameter $\lambda$,
       time $t$
Calculate the index coder $\gamma_t$. The variable length codes of $\gamma_t$ satisfy approximately $|\gamma_t(c_i)| = -\log_2 p_{t-1}(i)$.
Define $\alpha_t(x) \overset{\text{def}}{=} \arg\min_i[d(x, \beta_{t-1}(i)) + \lambda \cdot \gamma_t(i)]$
Calculate the change of the RD-function in case of a codebook update, $\Delta d = -d(\alpha_t(x_t), x_t), \quad \Delta r = \gamma_t(x_t),$ and $\Delta J = \Delta d + \lambda \cdot \Delta r$.
if $\Delta J < 0$ then
    $\mathcal{L}_t = \{x_t\}$ and $\mathcal{F}_t = \{\beta_{t-1}(N-1)\}$, update codebook $\mathcal{C}_t$,
    send to the decoder a flag indicating a codebook update and $\gamma^*(\alpha^*(x_t))$.
    Update the vector decoder, $\beta_t(i) = \begin{cases} \beta_{t-1}(i-1) & \text{if } i > 0 \\ Q^*(x_t) & \text{otherwise} \end{cases}$.
else
    Send a flag indicating no codebook update and set $\mathcal{C}_t \leftarrow \mathcal{C}_{t-1}$.
    Compute new vector decoder, let $j = \alpha_t(x_t)$,
    then $\beta_t(i) = \begin{cases} \beta_{t-1}(j) & i = 0 \\ \beta_{t-1}(i) & i > j \\ \beta_{t-1}(i-1) & i \leq j \end{cases}$.
end if
Compute $p_t(i)$ for all $c_i \in \mathcal{C}_t$.
```

Chapter 3

Rate-Distortion Optimization

In this chapter, we will present rate-distortion optimization techniques. The problems that are solved by this kind of algorithms are also known as *bit allocation* or, more general, *rate allocation* problems [84, 82]. First, we present general theoretical results for continuous random variables. We then describe the problem for the discrete case, where we are mainly concerned with two kinds of optimization algorithms: the Lagrangian multiplier algorithm and incremental computation of the solution. These algorithms are discussed for different optimization scenarios. In addition, a new algorithm for a general optimization scenario based on incremental computation is contributed. A proof of its correctness is provided.

3.1 The Rate Allocation Problem

We consider M independent random variables, X_m, $0 \leq m < M$. For each random variable, X_m, there is a distortion-rate function, $D_m(R) : \mathbb{R}^+ \to \mathbb{R}^+,\ 0 \leq m < M$ describing the trade-off between the rate needed to approximate the values of X_m and the corresponding distortion of the approximation. The *rate allocation problem* can be stated as follows:

$$\min \sum_{m=0}^{M-1} D_m(R_m) \ \text{ subject to } \ \sum_{m=0}^{M-1} R_m \leq R_T$$

with the *target rate* R_T.

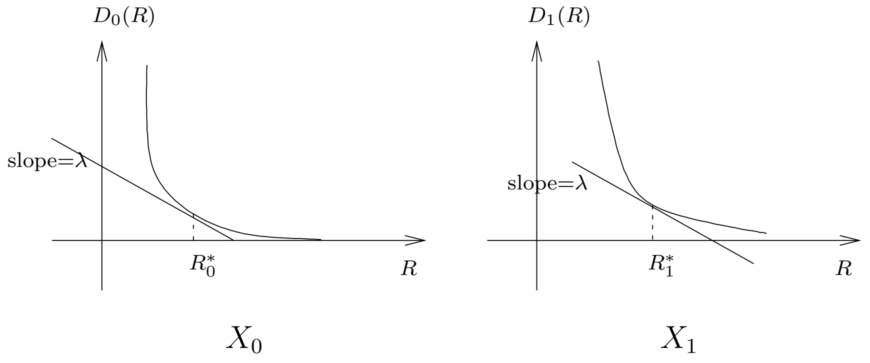


Figure 3.1: Geometric interpretation of the optimal solution of the continuous rate allocation problem.

This optimization problem can be solved by a Lagrangian multiplier approach. Provided that $D_m(R)$, $0 \leq m < M$, is differentiable, the solution $(R_0^*, \ldots, R_{M-1}^*)$ has the property

$$\frac{\partial D_m(R_m^*)}{\partial R} = -\lambda, \; 0 \leq m < M, \tag{3.1}$$

known as *constant slope* property since all curves described by the distortion rate function $D_m(R)$ have the same slope $-\lambda$ at the optimal solution R_m^* [84]. The geometric interpretation of (3.1) is shown in Fig. 3.1. Note that the distortion-rate curves, defined by $D_m(R)$, are always convex [4].

With (3.1) it is possible to derive a closed form approximation for some special cases. For example, if the random variables, X_m, have an independent Gaussian distribution with variance σ_m^2, the approximate rate, R_m^*, is given by the expression [38]

$$R_m^* \approx \frac{R_T}{M} + \frac{1}{2} \log_2 \frac{\sigma_m^2}{\left(\prod_{i=0}^{M-1} \sigma_m^2\right)^{1/M}}. \tag{3.2}$$

3.2 Rate Allocation for Discrete Rates and Distortions

In this section, we introduce the most basic case of practical rate allocation problems. In addition, we discuss different solution methods.

A typical rate allocation problem arises with the encoding process of a vector quantizer (for the terminology see Sect. 2.1). There are M vectors $x_0, \ldots, x_{M-1}$ that have to be encoded and the codebook size of the vector quantizer is N. For each x_m we get N distortions by computing the distortions between x_m and the codebook vectors, $D_m^n = d(x_m, \beta(n))$. In addition, for every x_m we get the encoding costs[1] $R_m^n = |\gamma(n)|$. From the example we can see that we do not deal any longer with distortion-rate functions but rather with distortion-rate sets $\mathcal{P}_m = \{(R_m^n, D_m^n) : 0 \leq n < N\}$. One possible encoding of the vectors $x_0, \ldots, x_{M-1}$ can be described by choosing for each vector x_m an appropriate codebook vector $\beta(i_m) \in \mathcal{C}, i_m \in \mathcal{I}$ of the codebook $\mathcal{C}$. Thus, one possible encoding of the M vectors is determined by an M-dimensional index vector $I = (i_0, \ldots, i_{M-1})$. The *discrete rate allocation problem* can therefore be formulated as

$$\min_{I \in \mathcal{I}^M} \sum_{m=0}^{M-1} D_m^{i_m} \quad \text{subject to} \quad \sum_{m=0}^{M-1} R_m^{i_m} \leq R_T \tag{3.3}$$

where $\mathcal{I} = \{0, \ldots, N-1\}$ describes the set of possible indices.

If adjacent points in the set $\mathcal{P}_m$ that could contribute to an optimal solution are connected by lines, the *operational rate-distortion curve* is defined (Fig. 3.2). Note that, unlike Sect. 3.1, the operational RD-curve is not necessarily convex.

One possible solution method could be exhaustive search in the space of all index vectors. But an examination of M^N solutions is not feasible for large M or N.

3.2.1 Computing the optimal solution

Under the assumption that all rates, R_m^n, are integers, the rate allocation problem can be optimally solved with the Viterbi algorithm [60, 59]. We assume that the reader is familiar with this kind of dynamic programming. Thus, we give only a short description of how to compute the optimal solution.

[1] In this example, the encoding costs are independent of the parameter m. Nevertheless we use the notation R_m^n to consider the general case.

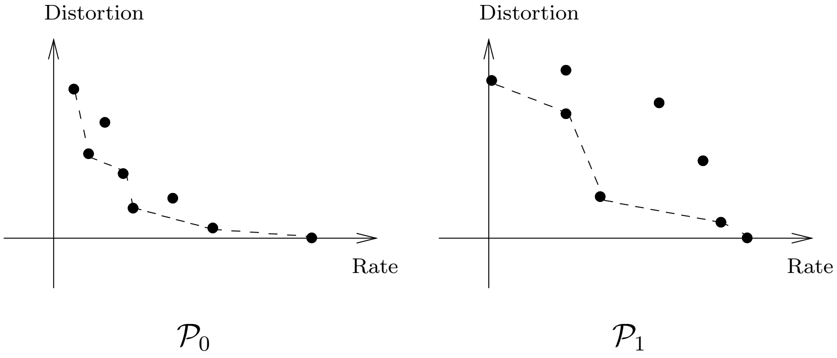


Figure 3.2: Operational rate-distortion-curve.

We create a trellis with the entries $d_{\min}(j,r)$. The vertical axis describes the cumulative rate, i.e., the rate that is achieved summing up the rates taken from points of certain sets $\mathcal{P}_m$. Therefore, we can have R_T cumulative rates where R_T denotes the target rate.

The horizontal axis describes the stages of the trellis. Each stage j can be assigned to a set of RD-points, $\mathcal{P}_j$. Hence, the trellis has M stages.

The entry of the trellis node (j,r), $d_{\min}(j,r)$, describes the smallest achievable distortion for a given cumulative rate r considering all stages $\leq j$, i.e.,

$$d_{\min}(j,r) = \min \sum_{t=0}^{j} D_t^{i_t} \quad \text{subject to} \quad \sum_{t=0}^{j} R_t^{i_t} \leq r, \quad i_t \in \mathcal{I} \tag{3.4}$$

If we assume that the values $d_{\min}(m,r)$ for all $m < j$ and all $r < R_T$ are already known, then the value of $d_{\min}(j,r)$ can be computed with

$$d_{\min}(j,r) = \min\{d_{\min}(j-1, r-R_j^n) + D_j^n : 0 \leq n < N \wedge R_j^n < r\}.$$

Thus, the transition cost from node $(j-1, r-R_j^i)$ to node (j,r) is given by D_j^i. If there are equal rates, $R_j^i = R_j^n$, $i \neq n$, then we take the smallest of the corresponding distortions.

The computation of $d_{\min}(j,r)$ is illustrated in Fig. 3.3. For all rates, R_j^n, $n \in \mathcal{I}$, we must consider the value $d_{\min}(j-1, r-R_j^n) + D_j^n$. The minimum of these values equals $d_{\min}(j,r)$. Only if the value of $d_{\min}(M-1, R_T)$ is known, the optimal path from the last stage $M-1$ to stage 0 can be determined. The complexity of this algorithm is $O(M \cdot N \cdot R_T)$.

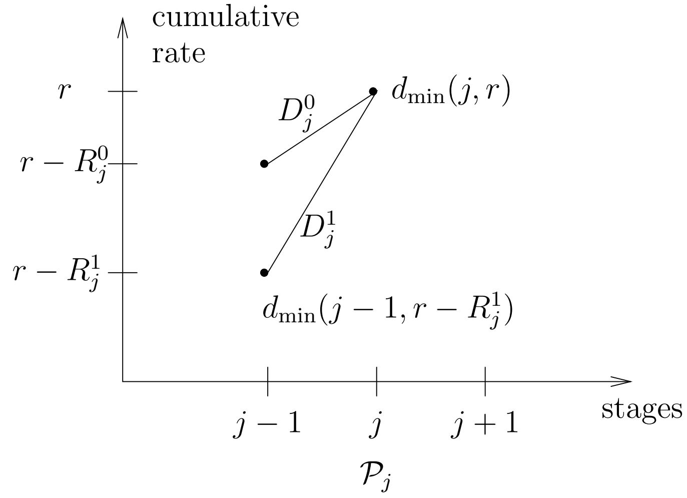


Figure 3.3: Viterbi Algorithm to find the optimal solution

3.2.2 The Lagrangian Multiplier Algorithm

If one is not interested in the optimal solution but only in a sufficient approximation of the solution, the Lagrangian multiplier (LM) technique might be applied. Classical LM approaches assume continuously differentiable functions [52]. But it was shown in [40] that this technique also works in the case of discrete values. Thus, this method can be used to find an approximate solution of the discrete rate allocation problem. If we set

$$J(\lambda, I) = \sum_{m=0}^{M-1} J_m(\lambda, i_m) = \sum_{m=0}^{M-1} D_m^{i_m} + \lambda \cdot R_m^{i_m},$$

the *constrained* optimization problem formulation of (3.3) can be transformed into the *unconstrained* optimization problem

$$\min_{I \in \mathcal{I}^M} J(\lambda, I) = \sum_{m=0}^{M-1} \min_{i_m \in \mathcal{I}} J_m(\lambda, i_m) = \sum_{m=0}^{M-1} \min_{i_m \in \mathcal{I}} [D_m^{i_m} + \lambda \cdot R_m^{i_m}]. \tag{3.5}$$

Moreover, if there exists a solution of the unconstrained problem then it is also a solution of the constrained problem.

Proposition 3 *If $R^*(\lambda) = \sum_{m=0}^{M-1} R_m^{i_m^*}$ and $D^*(\lambda) = \sum_{m=0}^{M-1} D_m^{i_m^*}$ is a solution of the unconstrained optimization problem (3.5) with a given λ and the optimal indices $i_m^* \in \{0, \ldots, N-1\}$, then it is also a solution of the*

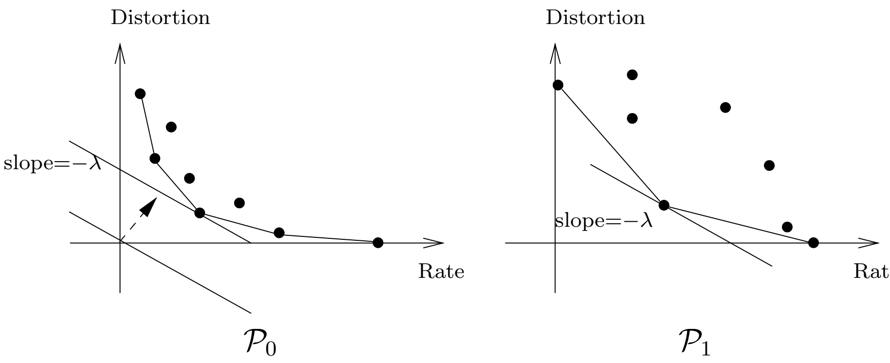


Figure 3.4: Geometrical interpretation of the Lagrangian multiplier.

constrained optimization problem (3.3) with $\min_{I\in\mathcal{I}^M} \sum_{m=0}^{M-1} D_m^{i_m}$ *subject to* $R = \sum_{m=0}^{M-1} R_m^{i_m} \leq R_T = R^*(\lambda)$.

Proof: see [40, 75, 71] □

A geometrical interpretation of the solution is shown in Fig. 3.4. Two sets, $\mathcal{P}_0$ and $\mathcal{P}_1$, are considered. A line with slope $-\lambda$ is drawn through the origin. Then it is shifted toward the lower part of the convex hull (LCH). The first point met by this line is the point minimizing $J_0(\lambda, i_0)$ and $J_1(\lambda, i_1)$, respectively [71, 59]. This shows that even in the discrete case there is a "constant slope" constraint for the solution if only solutions on the LCH are considered. The "constant slope" property is shared by the set $\mathcal{P} = \{(\sum_{m=0}^{M-1} R_m^{i_m}, \sum_{m=0}^{M-1} D_m^{i_m}) : I \in \mathcal{I}^M\}$ of all possible sums where the addends are taken from every set $\mathcal{P}_m : 0 \leq m < M$ one at a time. This time, the point met by the line with slope $-\lambda$ minimizes $J(\lambda, I)$. We call $\mathcal{P}$ the *global* set and $\mathcal{P}_m$ the *local* sets. Together with Proposition 3 we can conclude that the LM-technique can only describe optimal solutions of the rate allocation problem that are located on the LCH of the global set $\mathcal{P}$. If the target rate R_T is not exactly reached, an RD-point on the LCH with the closest rate to R_T is selected. In the following we call this solution *optimal LCH solution*.

Note that (3.5) explicitly defines a solution procedure. Fixing a value for λ, every $J_m(\lambda, i_m)$ can be minimized separately by computing the indices i_m^*. We call such computations for a given value of λ an *LM-iteration*. The complexity of one LM-iteration is $O(N \cdot M)$.

The LM-approach poses the problem to determine the value of λ that yields an optimal LCH solution for the given target bitrate R_T. Thus, a search strategy is necessary. The LM-iteration together with the search strategy describes the *LM-algorithm*.

The next proposition will help to understand the development of search strategies in the following.

Proposition 4 *The optimal rates of (3.5), $R^*(\lambda) = \sum_{m=0}^{M-1} R_m^{i_m^*}$, are monotonically decreasing with λ, that is, if $\lambda_2 \geq \lambda_1 > 0$ then $R^*(\lambda_2) \leq R^*(\lambda_1)$. Likewise, the optimal distortion $D^*(\lambda)$ is monotonically increasing.*

Proof: see [75, 71] □

In later chapters, we will need the following corollary.

Corollary 1 *The optimal rates of* $\min_{i \in \mathcal{I}} J_m(\lambda, i)$ *in (3.5), $R_m^*(\lambda) = R_m^{i_m^*}$, are monotonically decreasing with λ for* $0 \leq m < M$.

Proof: A special case of Proposition 4. □

Using the monotonicity of optimal rates and distortion, search strategies like bisection or Newton's method can be applied [80, 63, 62]. In the following, we describe a fast convex search algorithm to determine the value of λ that leads to an optimal LCH solution of (3.5) [75, 63]. This algorithm works as follows. The search is started with an *initial interval* $[\lambda_U, \lambda_L]$ where $R^*(\lambda_U) \geq R_T \geq R^*(\lambda_L)$. If $R^*(\lambda_U) = R_T$ or $R^*(\lambda_L) = R_T$, we stop since we know from Proposition 3 that we have found an optimal LCH solution already. Otherwise, we compute $\lambda_N = \frac{D^*(\lambda_L) - D^*(\lambda_U)}{R^*(\lambda_U) - R^*(\lambda_L)}$ which is within the search interval $[\lambda_U, \lambda_L]$. If $\lambda_N = \lambda_U$ or $\lambda_N = \lambda_L$, we can stop the search since this indicates that there could exist no other point with a rate between $[R^*(\lambda_L), R^*(\lambda_U)]$ on the LCH of the global set $\mathcal{P}$. If this is not the case, we check if $R^*(\lambda_N) \leq R_T$ or $R^*(\lambda_N) > R_T$ and set $\lambda_L \leftarrow \lambda_N$ or $\lambda_U \leftarrow \lambda_N$, respectively. If $R^*(\lambda_N) = R_T$, we have found an optimal solution. Otherwise, the algorithm is repeated with the smaller interval $[\lambda_U, \lambda_L]$. The procedure is depicted in Alg. 5. To complete the description of the algorithm we only have to specify the initial search interval. A default initialization could be $[0, \infty]$. However, the algorithm can be sped up if it starts with a tight initial interval. In the experiments in this thesis, we used the following scheme to find the initial interval. Assuming a good guess λ_{last} (usually estimated from λ of previous optimizations), the λ_{last} is doubled or halved until an initial interval is found. Then, Alg. 5 is applied.

To the best of the author's knowledge, complexity considerations of this algorithm have not appeared in the literature so far. Moreover, in the author's view it seems that the mainstream opinion is that such considerations are not reasonable because the computation of the RD-points is supposed to have a much higher complexity. However, in Chapter 5 we will see that this actually is not the case for the RD-problems we will be concerned with afterwards. The complexity of the LM-algorithm has the same order of magnitude as the computation of the RD-points and depends highly on the number of iterations that the fast convex search algorithm needs. Let A be the number of iterations. Then the complexity is $O(N \cdot M \cdot A)$. An upper bound of A is $M \cdot N$ since in the worst case all points in $\mathcal{P}$ lie on the LCH and the LM-algorithm could find all points during the search for the optimal λ. Thus, the worst case estimation of the complexity is $O(N^2 \cdot M^2)$. We will reconsider the complexity in Chapter 5.

Algorithm 5 Fast convex search of λ.

```
INPUT: initial search interval $[\lambda_U, \lambda_L]$, target rate $R_T$ with $R^*(\lambda_L) < R_T < R^*(\lambda_U)$
while $R^*(\lambda_U) \neq R_T$ and $R^*(\lambda_L) \neq R_T$ do
  compute $\lambda_N \leftarrow \frac{D^*(\lambda_L) - D^*(\lambda_U)}{R^*(\lambda_U) - R^*(\lambda_L)}$
  compute $R^*(\lambda_N)$ and $D^*(\lambda_N)$
  if $R^*(\lambda_N) = R^*(\lambda_U)$ or $R^*(\lambda_N) = R^*(\lambda_L)$ then
    stop
  end if
  if $R^*(\lambda_N) \leq R_T$ then
    $\lambda_L \leftarrow \lambda_N$
  else
    $\lambda_U \leftarrow \lambda_N$
  end if
end while
```

3.2.3 Incremental Computation of the Rate-Distortion Curve

From Sect. 3.2.2 we already know that the LM-algorithm needs several iterations to find a value for λ that yields an optimal LCH solution. In [28, 75, 90], a different approach is proposed that computes the LCH directly. Starting from the minimal (maximal) rate of the set of global RD-points $\mathcal{P}$, this algorithm computes the LCH of $\mathcal{P}$ for increasing (decreasing) rates until the best approximation of the target rate R_T is reached. This algorithm works as follows. First, the smallest achievable bitrate is searched, i.e., for every local set $\mathcal{P}_m$ the index i_m with the smallest rate $R_m^{i_m}$ is computed. Subsequently, for every local set $\mathcal{P}_m$ the index j_m describing the next point on the LCH of $\mathcal{P}_m$ is determined. Let λ_m be the slope between the RD-points specified by i_m and j_m. The set $\mathcal{P}_n$ with the “steepest”, i.e., minimal, λ_n is selected. The selected index of $\mathcal{P}_n$ is changed, $i_n \leftarrow j_n$. Then, the next point on the LCH of $\mathcal{P}_n$ is searched and so forth. This algorithm is detailed in Alg. 6.

Algorithm 6 Basic algorithm for incremental computation of the convex hull.

```
INITIALIZATION
for all macroblocks $X_m, 0 \leq m < M$ do
  $i_m \leftarrow \arg\min_i \{R_m^i : i \in I\}$
  $j_m \leftarrow \arg\min_i \{\frac{D_m^i - D_m^{i_m}}{R_m^i - R_m^{i_m}} : R_m^i > R_m^{i_m} \wedge D_m^i < D_m^{i_m} \wedge i \in I\}$
  $\lambda_m \leftarrow \frac{D_m^{j_m} - D_m^{i_m}}{R_m^{j_m} - R_m^{i_m}}$
end for
$rate \leftarrow \sum_{m=0}^{M-1} R_m^{i_m}$
OPTIMIZATION
while $rate < target\_rate$ do
  $n \leftarrow \arg\min_{0 \leq m < M} \lambda_m$
  $rate \leftarrow rate + R_n^{j_n} - R_n^{i_n}$
  $i_n \leftarrow j_n$
  $j_n \leftarrow \arg\min_i \{\frac{D_n^i - D_n^{i_n}}{R_n^i - R_n^{i_n}} : R_n^i > R_n^{i_n} \wedge D_n^i < D_n^{i_n} \wedge i \in I\}$
  $\lambda_n \leftarrow \frac{D_n^{j_n} - D_n^{i_n}}{R_n^{j_n} - R_n^{i_n}}$
end while
```

The complexity of Alg. 6 can be estimated as follows. During the initialization, all RD-points are considered to find the smallest rate in every local set $\mathcal{P}_m$. In addition, the first λ_m has to be computed for every $\mathcal{P}_m$. The complexity of the initialization can be estimated by $O(M \cdot N)$. During the optimization, we have to compute a new λ_m each time we find the next point on the LCH. The computation of λ_m needs $O(N)$ operations. Let there be B of such steps. Then the complexity of all steps can be estimated by $O(B \cdot N)$. In the worst case, all points on the LCH are traversed. Therefore, an upper bound for B is $M \cdot N$ and, thus, for the algorithm it is $O(M \cdot N^2)$. The complexity is considered in more detail in Chapter 5.

In the following section we will consider a more complicated kind of dependences between the local sets of RD-points, the *hierarchical* dependences. Finally, in Sect. 3.4 we present *general* dependences.

3.3 Optimization with Hierarchical Dependences

In the previous section, the global set $\mathcal{P}$ of rate-distortion points was created by considering all combinations of points in the local sets, i.e., $\mathcal{P}_0 \times \mathcal{P}_1 \times \ldots \times \mathcal{P}_{M-1}$, and adding up their rates and distortions. The summation was independent in the sense that the selection of a point in a certain local set does not affect the other selections. In this section we present examples which do not have this "independence" property. Thus, we have to extend the terminology to be able to describe such kind of optimization problems. Moreover, in the following we will have to merge two different notations: the one from the last section and the one used for tree structured optimization problems [11]. The resulting notation can be viewed as a generalization of [63].

As an example for a typical hierarchical RD-problem we consider tree-structured VQ. From Sect. 2.3.1 we know that such VQ contains not only one codebook but also a series of K codebooks, $\mathcal{C}_0, \ldots, \mathcal{C}_{K-1} = \mathcal{C}$, where only the last codebook $\mathcal{C}$, containing N codebook vectors, is used for the quantization of vectors and the intermediate codebooks are only needed for the search strategy. We assume that the indices are encoded with a FLC, i.e., $|\gamma(i)| = \lceil \log_2 N \rceil, 0 \leq i < N$. Such VQ can be considered as a full tree with the leaves taken from $\mathcal{C}_{K-1}$. The rate needed to encode one vector $c_n \in \mathcal{C}$, $|\gamma(n)|$, can be determined by counting the number of internal nodes that have to be traversed from the root node to the leaf corresponding to c_n.

A new codebook can be created by pruning the tree and considering the resulting leaves as new codebook vectors. This can be viewed as a rate allocation problem where the rate-distortion points are derived from the expected rates and distortions of all possible pruned trees. Using the notation from Sect. 2.3.1, we can state the problem as:

$$\min_{\mathcal{S} \preceq \mathcal{T}} \; \delta(\mathcal{S}) \;\; \text{subject to} \;\; l(\mathcal{S}) \leq R_T. \tag{3.6}$$

From Sect. 3.2 we know that the optimization can be understood as choosing an index vector $I = (i_0, \ldots, i_{M-1})$. We will show that (3.6) can also be restated in this terminology.

The solution in Sect. 3.2.2 was given by summing up independently a proper combination of rates and distortions from all local sets $\mathcal{P}_m$. Now we have to deal with hierarchically organized nodes of a tree $\mathcal{T} = \{t_0, \ldots, t_{M-1}\}$. Therefore, we will use the affinity of the tree functionals δ and l to investigate them at each node $t \in \mathcal{T}$. For this, we need the following decomposition property.

Proposition 5 *Let u be an affine tree functional defined on $\mathcal{T}$ and $\mathcal{S} \preceq \mathcal{T}$. Then $u(\mathcal{S}) = \sum_{t \in \tilde{\mathcal{S}}} u(t) + \sum_{t \in \mathcal{S} \setminus \tilde{\mathcal{S}}} a(t)$ where $a(t)$ denotes the value of t if t is an internal node and $u(t) = u(\{t\})$ the corresponding value if t is a leaf. The term $a(t)$ can be expressed as $a(t) = u(\{t, \mathit{left}(t), \mathit{right}(t)\}) - u(\mathit{left}(t)) - u(\mathit{right}(t))$ with the left and right child of node t, $\mathit{left}(t)$ and $\mathit{right}(t)$. For $t \in \tilde{\mathcal{T}}$ we define $a(t) = \infty$.*

Proof: Easy conversion of the affinity definition. □

With this proposition in mind, we can assign two possible encoding options to each node t, namely $u(t)$ if t is a leaf node and $a(t)$ if t is an internal node. The value of the tree functional $u(\mathcal{S})$ can then be computed for all $\mathcal{S} \preceq \mathcal{T}$ by the term

$$u(\mathcal{S}) = \sum_{t\in\mathcal{S}} v(t) \tag{3.7}$$

where $v(t)$ is selected from $\{u(t), a(t)\}$ with

$$v(t) = \begin{cases} u(t), & \text{if } t \in \tilde{\mathcal{S}} \\ a(t), & \text{otherwise.} \end{cases}$$

In particular, we can decompose the functionals l and δ. To each node t, we can assign two (l, δ) pairs, namely

$$(l^0(t), \delta^0(t)) \overset{\text{def}}{=} (l(t), \delta(t))$$

and

$$(l^1(t), \delta^1(t)) \overset{\text{def}}{=} \begin{cases} (l(\{t, \text{left}(t), \text{right}(t)\}) - l(\text{left}(t)) - l(\text{right}(t)), & \\ \delta(\{t, \text{left}(t), \text{right}(t)\}) - \delta(\text{left}(t)) - \delta(\text{right}(t))), & \text{if } t \in \mathcal{T} \setminus \tilde{\mathcal{T}} \\ (\infty, \infty), & \text{otherwise.} \end{cases}$$

The (l, δ) pairs can be considered as rate-distortion points. Thus, we assign two possible rate-distortion points to each node $t \in \mathcal{T}$, $(l^0(t), \delta^0(t)) = (R_t^0, D_t^0)$ for the case t is a leaf node and $(l^1(t), \delta^1(t)) = (R_t^1, D_t^1)$ for the case t is an internal node. Therefore, the local sets, $\mathcal{P}_t$, can be defined as $\mathcal{P}_t = \{(R_t^0, D_t^0), (R_t^1, D_t^1)\}$. With the above defined local sets $\mathcal{P}_t$, we can express the tree functional $(l(\mathcal{S}), \delta(\mathcal{S}))$ as follows. We know from (3.7) that $(l(\mathcal{S}), \delta(\mathcal{S}))$ can be computed by summing up a term for every $t \in \mathcal{S}$ depending on whether $t \in \tilde{\mathcal{S}}$ or $t \in \mathcal{S} \setminus \tilde{\mathcal{S}}$. Since there is only one RD-point for the case t is a leaf node and one for the case t is an internal node there is a map $f_{\mathcal{S}}$ from a subtree $\mathcal{S} = \{s_0, s_1, \dots\} \preceq \mathcal{T}$ to an index vector $(i_{s_0}, i_{s_1}, \dots)$,

$$f_{\mathcal{S}} : \mathcal{S} \to \{0, 1\},\ s_m \mapsto \begin{cases} 0 & : \quad s_m \in \tilde{\mathcal{S}} \\ 1 & : \quad \text{otherwise} \end{cases}. \tag{3.8}$$

With $i_{s_m} = f_{\mathcal{S}}(s_m)$ we can rewrite δ and l as

$$\delta(\mathcal{S}) = D(\mathcal{S}, f_{\mathcal{S}}) \overset{\text{def}}{=} \sum_{s_m\in\mathcal{S}} D_{s_m}^{i_{s_m}} \text{ and } l(\mathcal{S}) = R(\mathcal{S}, f_{\mathcal{S}}) \overset{\text{def}}{=} \sum_{s_m\in\mathcal{S}} R_{s_m}^{i_{s_m}}. \tag{3.9}$$

The optimization problem now can be described as:

$$\min_{\mathcal{S}\preceq\mathcal{T}} \sum_{s_m\in\mathcal{S}} D_{s_m}^{i_{s_m}} \quad \text{subject to} \quad \sum_{s_m\in\mathcal{S}} R_{s_m}^{i_{s_m}} \leq R_{\mathcal{T}}. \tag{3.10}$$

As in Sect. 3.2, we define the *global set* $\mathcal{P}_{\mathcal{T}}$ as the "sum of local sets $\mathcal{P}_t$",

$$\mathcal{P}_{\mathcal{T}} \overset{\text{def}}{=} \{(R(\mathcal{S}, f_{\mathcal{S}}), D(\mathcal{S}, f_{\mathcal{S}})) : \mathcal{S} \preceq \mathcal{T}\}.$$

Up to now the use of the index vector $(i_{s_0}, i_{s_1}, \dots)$ seems to be redundant since a tree structure, $\mathcal{S}$, implicitly defines an index vector via (3.8). The next example for a typical hierarchical dependence arising with image and video coding will show that a more general view is useful.

We would like to encode an image macroblock of 16×16 pixel size, a typical macroblock size for image and video coding (cf. Sect. 1.6), in the following way. There exist N different encoding options for the full macroblock, e.g., different vectors in a vector quantizer. Alternatively, the macroblock can be decomposed into four 8×8 pixel blocks. Again, there are N independent encoding options for each 8×8 pixel block and each 8×8 pixel block can be independently decomposed into four 4×4 pixel blocks. Each of these 4×4 pixel blocks can be independently encoded with N coding options. Such a structure can be described by a quad-tree (cf. Sect. 1.4).

The pure tree structure can be described with the notation of Sect. 2.3. But unlike the optimization problem in (3.10) the local set $\mathcal{P}_t$ of the node t contains N points for each coding option and one point (R_t^N, D_t^N) for the case of t being an internal node, i.e., $\mathcal{P}_t = \{(R_t^0, D_t^0), \dots, (R_t^N, D_t^N)\}$. Thus, the index vector $(i_{s_0}, i_{s_1}, \dots)$ is not completely specified by the tree structure $\mathcal{S}$. We know that if t is an internal node, we have only one possible encoding option, $i_t = N$. On the other hand, if t is a leaf node, there are N possible encoding options, $i_t \in \{0, \dots, N-1\}$. Therefore we have to consider the set of all possible index functions,

$$\mathcal{F}_{\mathcal{S}} \overset{\text{def}}{=} \{f : \mathcal{S} \to \{0, \dots, N\}, f(s_m) \in \begin{cases} \{0, \dots, N-1\} & : \quad s_m \in \tilde{\mathcal{S}} \\ \{N\} & : \quad \text{otherwise.} \end{cases}$$

With $i_{s_m} = f(s_m)$, the optimization problem can be described as

$$\min_{(\mathcal{S},f):\ \mathcal{S}\preceq\mathcal{T}, f\in\mathcal{F}_{\mathcal{S}}} \sum_{s_m\in\mathcal{S}} D_{s_m}^{i_{s_m}} \quad \text{subject to} \quad \sum_{s_m\in\mathcal{S}} R_{s_m}^{i_{s_m}} \leq R_{\mathcal{T}} \tag{3.11}$$

Unlike the optimization problem (3.10), we have not only to determine the tree structure, but also the encoding option for the leaf nodes. Obviously, (3.10) is a special case of (3.11). The global set $\mathcal{P}_\mathcal{T}$ amounts to

$$\mathcal{P}_\mathcal{T} = \{(R(\mathcal{S}, f), D(\mathcal{S}, f)) : \mathcal{S} \preceq \mathcal{T},\ f \in \mathcal{F}_\mathcal{S}.\}$$

Now we have to explain the term "hierarchical dependence". Unlike Sect. 3.2, the global set $\mathcal{P}_\mathcal{T}$ is not created by a summation of points independently contributed by several local sets $\mathcal{P}_t$. The selection of an index for a node $t \in \mathcal{T}$ could rather exclude all children of t from the sum if t is selected a leaf-node. Thus, the contribution of node t to the sum depends on the index of parent nodes.

In the following, we present algorithms to solve rate allocation problems discussed above. Section 3.3.1 describes an algorithm to find solutions for problem (3.6) and (3.11) with the Lagrangian multiplier algorithm. In Sect. 3.3.2, an algorithm to solve problem (3.6) is presented based on incremental computation of the LCH of the global set. Finally we contribute an algorithm to solve (3.11) with incremental computation of the LCH.

To keep things simple, the algorithms are shown only for binary trees. The generalization for trees with more children can be made straight-forwardly.

3.3.1 The Lagrangian Multiplier Algorithm

As we have seen in Sect. 3.2.2, the constrained optimization problems (3.6) and (3.11) can be transformed into the unconstrained problems

$$\min_{\mathcal{S} \preceq \mathcal{T}} [\delta(\mathcal{S}) + \lambda \cdot l(\mathcal{S})] \tag{3.12}$$

and

$$\min_{(\mathcal{S}, f):\ \mathcal{S} \preceq \mathcal{T}, f \in \mathcal{F}_\mathcal{S}} \left[\sum_{s_m \in \mathcal{S}} D_{s_m}^{i_{s_m}} + \lambda \sum_{s_m \in \mathcal{S}} R_{s_m}^{i_{s_m}} \right]. \tag{3.13}$$

An algorithm to solve (3.12) and (3.13) has been presented in [82, 63]. This algorithm is based on dynamic programming [3].

In the following, we denote the argument of the solution of (3.13) restricted to a subtree $\mathcal{R} \sqsubset \mathcal{T}$ by $(\mathcal{R}^*, f^*_{\mathcal{R}^*})$, i.e.,

$$(\mathcal{R}^*, f^*_{\mathcal{R}^*}) = \arg \min_{(\mathcal{S},f):\ \mathcal{S}\preceq\mathcal{R}, f\in\mathcal{F}_\mathcal{S}} \left[\sum_{s_m\in\mathcal{S}} D^{i_{s_m}}_{s_m} + \lambda \sum_{s_m\in\mathcal{S}} R^{i_{s_m}}_{s_m} \right]. \qquad (3.14)$$

We call $(\mathcal{R}^*, f^*_{\mathcal{R}^*})$ the *solution for* $\mathcal{R}$.

The minimum value that is achieved by the argument $(\mathcal{R}^*, f^*_{\mathcal{R}^*})$ in (3.14) is denoted by $J(\mathcal{R}^*, f^*_{\mathcal{R}^*})$, i.e.,

$$J(\mathcal{R}^*, f^*_{\mathcal{R}^*}) = \min_{(\mathcal{S},f):\ \mathcal{S}\preceq\mathcal{R}, f\in\mathcal{F}_\mathcal{S}} \left[\sum_{s_m\in\mathcal{S}} D^{i_{s_m}}_{s_m} + \lambda \sum_{s_m\in\mathcal{S}} R^{i_{s_m}}_{s_m} \right].$$

The "principle of optimality" for the dynamic programming solution for a fixed value of λ reads:

> If a node $t \in \mathcal{T}$ is considered, we do not know in advance whether $t \in \mathcal{T}^*$ or not. But if $t \in \mathcal{T}^*$, then $(\mathcal{S}^*_t, f^*_{\mathcal{S}^*_t})$ with the branch $\mathcal{S}_t$ of $\mathcal{T}$ also belongs to the solution of (3.13), i.e., $\mathcal{S}^*_t \subset \mathcal{T}^*$ and $\forall s \in \mathcal{S}^*_t : f^*_{\mathcal{S}^*_t}(s) = f^*_{\mathcal{T}^*}(s)$.

With this principle, a recursive algorithm can be developed. Let us consider node t. We assume that the solution for the left branch, $\mathcal{S}_{\text{left}(t)}$, $(\mathcal{S}^*_{\text{left}(t)}, f^*_{\mathcal{S}^*_{\text{left}(t)}})$ and for the right branch $(\mathcal{S}^*_{\text{right}(t)}, f^*_{\mathcal{S}^*_{\text{right}}(t)})$ is known. Then we can compute $(\mathcal{S}^*_t, f^*_{\mathcal{S}^*_t})$ as follows. We compute $J^{\sum}_t$, the value of $J(\mathcal{S}^*_t, f^*_{\mathcal{S}^*_t})$ for the case t is an internal node, and J_t, the corresponding value if t is a leaf node. $J^{\sum}_t$ can be calculated by

$$J(\mathcal{S}^*_{\text{left}(t)}, f^*_{\mathcal{S}^*_{\text{left}(t)}}) + J(\mathcal{S}^*_{\text{right}(t)}, f^*_{\mathcal{S}^*_{\text{right}}(t)}) + (D^N_t + \lambda \cdot R^N_t)$$

and the cost for the case t is a leaf, J_t, by

$$\min_{i\in\{0,\dots,N-1\}} D^i_t + \lambda \cdot R^i_t.$$

If $J_t \leq J^{\sum}$, then $\mathcal{S}^*_t = \{t\}$ otherwise $\mathcal{S}^*_t = \{t\} \cup \mathcal{S}^*_{\text{left}(t)} \cup \mathcal{S}^*_{\text{right}(t)}$. In the former case, we have $f^*_{\mathcal{S}^*_t}(t) = \arg\min_{i\in\{0,\dots,N-1\}} D^i_t + \lambda \cdot R^i_t$, in the latter case $f^*_{\mathcal{S}^*_t}(t) = N$. The solution $(\mathcal{S}^*_t, f^*_{\mathcal{S}^*_t})$ for $\mathcal{S}_t$ can be directly computed for the leaf nodes of $\mathcal{T}$ since no child nodes have to be taken into account and only J_t must be calculated. For an internal node $t \in \mathcal{T} \setminus \tilde{\mathcal{T}}$, $(\mathcal{S}_t^*, f_{\mathcal{S}_t^*}^*)$ can be computed when $(\mathcal{S}_{\text{left}(t)}^*, f_{\mathcal{S}_{\text{left}(t)}^*}^*)$ and $(\mathcal{S}_{\text{right}(t)}^*, f_{\mathcal{S}_{\text{right}}^*(t)}^*)$ are known. Thus, $(\mathcal{S}_{t_0}^*, f_{\mathcal{S}_{t_0}^*}^*)$ can be computed for the root node, t_0, by traversing the tree from the leaves to the root.

The complete procedure is shown in Alg. 7. The algorithm computes the indices $i_t^* = f_{\mathcal{S}_t^*}^*(t)$ and the minimal Lagrangian cost function $J(\mathcal{S}_t^*, f_{\mathcal{S}_t^*}^*)$ for all $t \in \mathcal{T}$. The optimal tree structure $\mathcal{S}_{t_0}^*$ consists of all nodes that can be reached from the root traversing only nodes t for which $i_t^* = N$.

The computational complexity is dominated by the term

$$i_t = \arg \min_{0 \leq i < N} [D_t^i + \lambda \cdot R_t^i]$$

Thus, the LM-iteration can be estimated by $O(M \cdot N)$ for a fixed value of λ. Like the LM-iteration in Sect. 3.2.2, a proper value for λ must be determined. This can be done with the fast convex search algorithm described in Alg. 5 since for even hierarchical dependences there exists a monotonic property similar to Proposition 4.

Proposition 6 *The optimal rates of (3.13), $R^*(\lambda) = \sum_{s_m \in \mathcal{T}^*} R_{s_m}^{i_{s_m}^*}$, are monotonically decreasing with λ, that is, if $\lambda_2 \geq \lambda_1 > 0$ then $R^*(\lambda_2) \leq R^*(\lambda_1)$. Likewise, the optimal distortion $D^*(\lambda)$ is monotonically increasing.*

Proof: see [71] □

Like $\mathcal{P}$ in Sect. 3.2.2, the solution of (3.12) and (3.13) describes a point of the LCH of the global set $\mathcal{P}_{\mathcal{T}}$ [11, 82].

3.3.2 The Generalized BFOS

In this section we present an algorithm to solve (3.12). It was developed by Breiman et al. [6] for the optimization of classification and regression trees. This approach, in turn, was used by Chou et al. to solve rate allocation optimization and related problems [11]. The tree functionals δ and l in (3.6) are assumed to satisfy several conditions, namely to be affine, monotonic, and having only one option for the case node t is a leaf.

Algorithm 7 LM-iteration for hierarchical dependences

```
Given: RD-parameter $\lambda$
LM_iteration(node $t$)
{
if $t \in \mathcal{T} \setminus \tilde{\mathcal{T}}$ then
   LM_iteration(left($t$))
   LM_iteration(right($t$))
else
   $J^*_{\text{left}(t)} \leftarrow \infty$
   $J^*_{\text{right}(t)} \leftarrow \infty$
end if
$J^{\sum}_t \leftarrow J^*_{\text{left}(t)} + J^*_{\text{right}(t)} + (D^N_t + \lambda \cdot R^N_t)$
$i_t \leftarrow \arg\min_{0 \le i < N}[D^i_t + \lambda \cdot R^i_t]$
$J_t \leftarrow D^{i_t}_t + \lambda \cdot R^{i_t}_t$
if $J^{\sum}_t < J_t$ then
   $i^*_t \leftarrow N$
   $J^*_t \leftarrow J^{\sum}_t$
else
   $i^*_t \leftarrow i_t$
   $J^*_t \leftarrow J_t$
end if
}
```

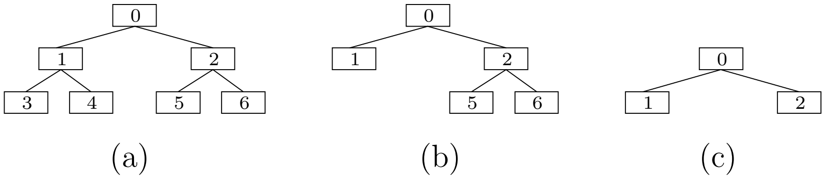


Figure 3.5: Sequence of pruned trees.

The key of the algorithm is that one can construct the optimal LCH solution $\mathcal{S}^* \preceq \mathcal{T}$ by successive pruning of several branches of $\mathcal{T}$ (Fig. 3.5), i.e., $\mathcal{T} = \mathcal{S}_0 \succeq \mathcal{S}_1 \succeq \ldots \succeq \mathcal{S}^*$. In [11] it is proven that the corresponding sequence of RD-points $(l(\mathcal{S}_0), \delta(\mathcal{S}_0)), \ldots, (l(\mathcal{S}^*), \delta(\mathcal{S}^*))$ describes the LCH of the global set $\mathcal{P}_\mathcal{T} = \{(l(\mathcal{S}), \delta(\mathcal{S})) : \mathcal{S} \preceq \mathcal{T}\}$.

Let $\mathcal{S}_t$ be the branch that was pruned to reach $\mathcal{S}_{i+1}$ from $\mathcal{S}_i$. The affinity of the components of $\mathbf{u} = (l, \delta)$ assures that the difference between $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$, $\Delta\mathbf{u_i} = (l(\mathcal{S}_i) - l(\mathcal{S}_{i+1}), \delta(\mathcal{S}_i) - \delta(\mathcal{S}_{i+1}))$ can be described with $\Delta\mathbf{u_i} = \Delta\mathbf{u}(\mathcal{S}_t) = \mathbf{u}(\mathcal{S}_t) - \mathbf{u}(t)$ where

$$\mathbf{u}(\mathcal{S}) = \begin{pmatrix} l(\mathcal{S}) \\ \delta(\mathcal{S}) \end{pmatrix}.$$

To manage the pruning and the change of the tree efficiently, a special data structure is used,

$$\text{node}(t) = \begin{pmatrix} \Delta\mathbf{u}(\mathcal{S}_t) \\ \lambda(t) \\ \lambda_{\min}(t) \\ \text{left}(t) \\ \text{right}(t) \end{pmatrix}, \tag{3.15}$$

where $\Delta\mathbf{u}(\mathcal{S}_\mathbf{t}) = (\Delta l, \Delta\delta)$ denotes the rate decrease Δl and the distortion increase $\Delta\delta$ if node t is pruned, $\lambda(t)$ denotes the ratio $-\frac{\Delta\delta}{\Delta l}$, left$(t)$ and right(t) the left and right child of node t, and $\lambda_{\min} = \min\{\lambda(t), \lambda_{\min}(\text{left}(t), \lambda_{\min}(\text{right}(t))\}$.

Apart from the initialization, the generalized BFOS algorithm can be divided into two phases: In the first phase, the node t_p with the smallest $\lambda(t)$ is searched and pruned. This t_p can be determined as follows. Every node t contains $\lambda_{\min}(t) = \min\{\lambda(s) : s \in \mathcal{S}_t\}$ the smallest value $\lambda(s)$ over all nodes

s in the branch $\mathcal{S}_t$. This information is also available for the left and right child node. Therefore, $\lambda_{\min}$ only has to be compared with $\lambda(t)$, $\lambda_{\min}(\mathrm{left}(t))$ and $\lambda_{\min}(\mathrm{right}(t))$ to find out whether the node with the smallest λ is t itself, in the left, or in the right subtree, respectively. If node t contains the smallest value of λ, it has to be pruned otherwise the search continues recursively to $\mathrm{left}(t)$ or $\mathrm{right}(t)$. If M is the number of nodes and the search starts at the root t_0 of the full tree $\mathcal{T}$, the search for the smallest value of λ needs $O(\log M)$ steps. This phase is called *search and prune phase*.

After the node t_p has been pruned, the data structure (3.15) has to be updated. This applies to all nodes that lie on the way from the root to t_p, i.e., to all $t \in \{t : t_p \in \mathcal{S}_t\}$. Let Δ be $\Delta\mathbf{u}(t_p)$. Since the tree functionals δ and l are affine, each $t \in \{t : t_p \in \mathcal{S}_t\}$ can be updated in the following way: $\Delta\mathbf{u}(t) \leftarrow \Delta\mathbf{u}(t) - \Delta, \lambda(t) \leftarrow -\frac{\Delta\delta(t)}{\Delta l(t)}$. Afterwards, we have to update $\lambda_{\min}(t) \leftarrow \min\{\lambda(t), \lambda_{\min}(\mathrm{left}(t)), \lambda_{\min}(\mathrm{right}(t))\}$. This phase is called *update phase*. The complete algorithm is shown in Alg. 8.

3.3.3 Extension of the Generalized BFOS

The generalized BFOS can be used to optimize tree structured rate allocation problems. This algorithm, however, imposes several assumptions on the tree functionals δ and l, namely monotonicity and only one option for every t to be a leaf. This is the reason for the simplicity of the algorithm. The rate decreases if the tree gets smaller. The monotonicity assures that once nodes are removed by pruning they do not need to be reconsidered. If the monotonicity assumption is dropped, this feature is lost. It can happen that for smaller rates the tree increases and nodes that have been pruned before are, nevertheless, part of the optimal LCH solution. Furthermore, if the assumption is dropped that only one option is provided for every node t to be a leaf the value of the functionals can change without changing the tree structure.

In this section, we develop an algorithm for affine tree functionals without any constraint. This algorithm can be used to solve optimization problems like (3.13). In addition, it is possible to maintain the "spirit" of the generalized BFOS (GBFOS) algorithm.

Algorithm 8 The generalized BFOS

```
INITIALIZATION
for each leaf node t do
    $\Delta\mathbf{u}(S_t) \leftarrow 0$
    $\lambda_{min}(t) \leftarrow \infty$
end for
for each interior node $t$ do
    $\Delta\mathbf{u}(S_t) \leftarrow \mathbf{u}(S_t) - u(t)$
    $\lambda(t) \leftarrow -\frac{\Delta\delta(t)}{\Delta l(t)}$
    $\lambda_{min}(t) \leftarrow \min\{\lambda(t), \lambda_{min}(left(t)), \lambda_{min}(right(t))\}$
end for
$\mathbf{u} \leftarrow \mathbf{u}(t_0) + \Delta\mathbf{u}(S_{t_0})$
while $\lambda_{min}(t_0) < \infty$ do
    SEARCHING. Start with root node $t_0$
    while $\lambda(t) > \lambda_{min}(t_0)$ do
        if $\lambda_{min}(left(t)) = \lambda_{min}(t_0)$ then
            $t \leftarrow$ left($t$)
        else
            $t \leftarrow$right($t$)
        end if
    end while
    $\Delta \leftarrow \Delta\mathbf{u}(S_t)$
    $\lambda_{min}(t) \leftarrow \infty$
    UPDATE
    while $t \neq t_0$ do
        $t \leftarrow$parent($t$)
        $\Delta\mathbf{u}(S_t) \leftarrow \Delta\mathbf{u}(S_t) - \Delta$
        $\lambda(t) \leftarrow -\frac{\Delta\delta(S_t)}{\Delta l(S_t)}$
        $\lambda_{min}(t) \leftarrow \min\{\lambda(t), \lambda_{min}(left(t)), \lambda_{min}(right(t))\}$
    end while
    $\mathbf{u} \leftarrow \mathbf{u} - \Delta$
    if $l(S_{t_0}) < l_{target}$ then
        break
    end if
end while
```

In the following, we investigate the data structure (3.15) for one node t during the GBFOS algorithm. We show that the restrictions of the GBFOS are implied by this data structure in order to motivate that a new data structure is necessary. We know from (3.9) that the functionals can be interpreted as the sum of points from the local sets $\mathcal{P}_t$ selected by the function f. In the GBFOS algorithm, there is only one possible RD-point for t being a leaf, namely (R_t^0, D_t^0). In addition, there is a variety of implicit RD-points that are given by the tree functionals of all possible subtrees of the branch $\mathcal{S}_t$, $\{(l(\mathcal{S}), \delta(\mathcal{S}) : \mathcal{S} \preceq \mathcal{S}_t\}$. These RD-points have to be considered by the GBFOS algorithm but do not appear explicitly in the data structure (3.15).

Since the GBFOS starts with the highest possible bitrate, the monotonicity of the tree functional assures that the algorithm starts with the maximal subtree $\mathcal{S}_t$. The value of $\Delta\mathbf{u}(t) = \mathbf{u}(\mathcal{S}_t) - \mathbf{u}(t)$ describes the rate and distortion difference between the *local* point $\mathbf{u}(t) = (l(t), \delta(t)) = (R_t^0, D_t^0)$ and the *branch* RD-point $(l(\mathcal{S}_t), \delta(\mathcal{S}_t)) = (R_t^b, D_t^b)$ from the currently "selected" subtree, and, thus, we define $\lambda_0(t) = \lambda(t)$ in order to describe the slope between the local point and the branch point. This holds true throughout all steps of the GBFOS algorithm. Even if a node in $\mathcal{S}_t$ is pruned, the update procedure assures that the updated $\lambda(t)$, denoted by $\lambda_1(t)$, still describes the slope between the RD-point of the new branch and the local RD-point that does never change. This is shown by Fig. 3.6a. The black circle describes the single local RD-point. The open circle with the highest rate describes the RD-point of subtree $\mathcal{S}_t$. All other open circles correspond to subtrees $\mathcal{S} \preceq \mathcal{S}_t$. Therefore, the data structure (3.15) can manage only one local RD-point. The case that a branch point is smaller than the local point may cause problems since it can happen that $\lambda(t) = \lambda_{\min}(t_0)$ but the pruning of t increases the rate. Thus, we can conclude that the restrictions of the GBFOS are implied by the data structure (3.15).

In contrast to the generalized BFOS algorithm, the algorithm presented in the following computes the LCH of the global set $\mathcal{P}_\mathcal{T}$ counter clockwise. Thus, the algorithm starts with the smallest rate, and the sign of $\lambda(t)$ is reversed. Nevertheless, with slight changes, this algorithm can also be used to compute

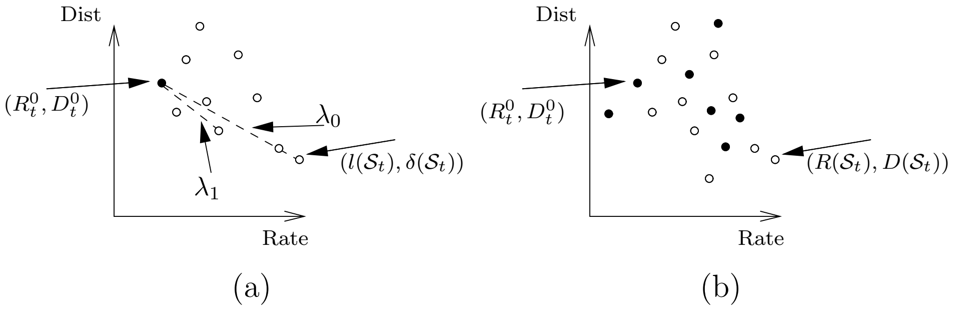


Figure 3.6: RD-points for a node t. For the GBFOS, (a) shows only one local RD-point (black circle). The open circles describe RD-points assigned to subtrees $\mathcal{S} \preceq \mathcal{S}_t$. For the extension of the GBFOS (b) there exist some more local RD-points.

the LCH of $\mathcal{P}_\mathcal{T}$ clockwise.

If the two conditions for the tree-functionals, monotony and only one local point, are dropped, the implicit data structure has to be enhanced since $\lambda(t)$ can now also describe the slope between two local points, and without the monotony it can also describe the slope of the step from a local point to a branch point when the local point has a smaller rate than the branch point (Fig. 3.6b). Therefore, we define an enhanced data structure.

$$node(t) = \begin{pmatrix} P_t[i], 0 \le i \le N \\ P_b(t) \\ f_c(t) \\ f_n(t) \\ \lambda(t) \\ \lambda_{min}(t) \\ \text{left}(t) \\ \text{right}(t) \end{pmatrix}. \tag{3.16}$$

Here, $\lambda_{\min}$, left(t) and right(t) have the same meaning as in the implicit data structure (3.15). The first component in the data structure, $P_t[i]$, is an array consisting of $N+1$ RD-points. We take the first N points into consideration, $P_t[i]$, $0 \le i < N$, that are the points of the local set $\mathcal{P}_t$ corresponding to the case t is a leaf. Thus, for $0 \le i < N$ we set $P_t[i] = (R_t[i], D_t[i]) \overset{\text{def}}{=} (R_t^i, D_t^i)$. Note that the notation $(R_t[i], D_t[i])$ differs purposely from the previously defined notation (R_t^i, D_t^i). This new notation is introduced because of a slight difference in both notations for the Nth RD-point. We will see that in our algorithm $(R_t[N], D_t[N]) \neq (R_t^N, D_t^N)$. The next component, $P_b(t)$, describes the RD-point of the "currently selected branch" as described before and is called *branch point*. The component $f_c(t)$ describes the selection function for the node t and $f_n(t)$ a potentially next candidate for $f_c(t)$. The node t is a leaf node with RD-point $P_t[f_c(t)] = (R_t[f_c(t)], D_t[f_c(t)])$ if $0 \leq f_c(t) < N$ and an internal node if $f_c(t) = N$. For ease of notation, we set in the following $P_t[N] = P_t^b$ and, thus, $P_t[N] = (R_t[N], D_t[N]) \neq (R_t^N, D_t^N)$. The next selection function $f_n(t)$ is chosen in order to minimize the value of $\lambda(t)$, $f_n(t) = \arg\min_i\{\frac{D_t[i]-D_t[f_c(t)]}{R_t[i]-R_t[f_c(t)]} : 0 \leq i \leq N \wedge R_t[i] > R_t[f_c(t)]\}$. To illustrate these terms, an example is provided in Appendix A.

It is an essential difference to the GBFOS that the following algorithm is applied on all nodes $t \in \mathcal{T}$ throughout, independent of the selected indices i_t, e.g., even if parent nodes are leaf nodes. This is due to the fact that, without the monotony condition, the size of the optimal tree can decrease or increase if the target rate increases.

First, we must consider the initialization of the data structure. It is assumed that tree structure $\mathcal{T}$ is given and the local points $P_t[i]$ do not change during the algorithm. Hence the values of $\text{left}(t)$, $\text{right}(t)$ and $P_t[i], 0 \leq i < N$ are fixed. Note, that the values of $\text{left}(t)$ and $\text{right}(t)$ serve the purpose to find all nodes in the full tree $\mathcal{T}$ and indicate children nodes of t even if t is a leaf in the optimal tree $\mathcal{T}^*$. The remaining elements of the data structure have to be initialized. This is made by the procedure INIT depicted in Alg. 9. For all $t \in \mathcal{T}$, the branch point P_t^b describes the subtree $S \preceq S_t$ with the smallest achievable rate for the case t is an internal node. Therefore, P_t^b can be recursively computed by $P_t^b = (R_t^N, D_t^N) + P_{\text{left}(t)}[f_c(\text{left}(t))] + P_{\text{right}(t)}[f_c(\text{right}(t))]$ where $(R_t^N, D_t^N) \in \mathcal{P}_t$ is the RD-point for the case t is an internal node. Then the point with the smallest rate $P_t[f_c(t)]$ over all points $P_t[n]$, $0 \leq n \leq N$ is calculated.

After the initialisation, the point $R_{t_0}[f_c(t_0)]$ describes the point in $\mathcal{P}_{\mathcal{T}}$ with the smallest rate.

Algorithm 9 INIT

```
INIT(node t)
{
if (left($t$) or right($t$)) then
    $P_t^b \leftarrow (R_t^N, D_t^N)$
else
    $P_t^b \leftarrow (\infty, \infty)$
end if
if left($t$) then
    INIT(left($t$))
    $P_t^b \leftarrow P_t^b + P_{\text{left}(t)}[f_c(\text{left}(t))]$
end if
if right($t$) then
    INIT(right($t$))
    $P_t^b \leftarrow P_t^b + P_{\text{right}(t)}[f_c(\text{right}(t))]$
end if
$f_c(t) \leftarrow \arg\min_i\{R_t[i] : P_t[i] = (R_t[i], D_t[i]) \wedge 0 \leq i \leq N\}$
FIND_NEXT_LAMBDA(t)
$\lambda_{min}(t) \leftarrow \min\{\lambda(t), \lambda_{min}(left(t)), \lambda_{min}(right(t))\}$
}
```

Like the GBFOS algorithm, we now have to determine two phases: The search for the smallest value of $\lambda(t)$ (search and prune phase) and the subsequent update of the data structure (update phase).

The search for the smallest value of $\lambda(t)$ can be organized as it is in the GBFOS algorithm. For each node t, it can be determined whether the smallest λ has been found or if it is in the left or in the right branch. Starting with the root node and assuming M nodes in the full tree $\mathcal{T}$, the node t_p can be found in about $\lceil \log M \rceil$ steps.

For the GBFOS algorithm, the pruning has the consequences that the selection function $f_c(t_p)$ is changed. Before the change, the branch-point was selected, i.e., $f_c(t_p) = N$. After the change, the (single) local point is selected, i.e., $f_s(t_p) = 0$. The "pruning" in the extended case is more complicated. The value of $f_c(t_p)$ is changed to a value, indicated by the next selection function value $f_n(t_p)$. There are three possible cases:

1. $f_n(t_p) < N$ and $f_c(t_p) = N$
2. $f_n(t_p) < N$ and $f_c(t_p) < N$
3. $f_n(t_p) = N$

The first case describes the situation that can arise with the GBFOS algorithm. Before $f_c(t_p)$ is changed, t_p was an internal node. After that, t_p is a leaf node. In the second case, $f_c(t_p)$ still indicates a local RD-point after the change. In the third case, t_p becomes an internal node. In all cases the selection function is changed $f_c(t_p) \leftarrow f_n(t_p)$. A new next selection function value $f_n(t_p)$ must be computed as well as the new corresponding $\lambda(t_p)$. This is done by the procedure FIND_NEXT_LAMBDA detailed in Alg. 10. Afterwards, the value of $\lambda_{\min}$ must be maintained for each node that is met traversing the tree from the root t_0 to t_p; this is achieved by computing $\lambda_{\min} \leftarrow \min\{\lambda(t), \lambda_{\min}(\text{left}(t)), \lambda_{\min}(\text{right}(t))\}$ at the end of every recursion[2].

[2] This is, strictly speaking, a part of the update phase. However, in the extended GBFOS, this has to be done for the "pruned" node, too. Thus, it is convenient to carry out this computation at the end of the recursion of Alg. 11.

The extended "search and prune" algorithm is depicted in Alg. 11. The value $\Delta \leftarrow P_t[f_n(t)] - P_t[f_c(t)]$ is computed for the update phase later on.

Algorithm 10 find next lambda and next selection function value

```
FIND_NEXT_LAMBDA(node t)
{
for all $0 \leq i \leq N$ do
  if $((R_t[i] > R_t[f_c(t)])$ and $(D_t[i] <= D_t[f_c(t)]))$ then
    $\lambda_h[i] \leftarrow \frac{(D_t[i]-D_t[f_c(t)])}{(R_t[i]-R_t[f_c(t)])}$
  else
    $\lambda_h[i] \leftarrow 0$
  end if
end for
$f_n(t) \leftarrow \arg\min_{0 \leq i \leq N} \lambda_h[i]$
$\lambda(t) \leftarrow \lambda_h[f_n(t)]$
if $\lambda(t) = 0$ then
  $f_n(t) = \infty$
end if
}
```

In the update phase, all nodes on the way from t_p to the root node t_0 have to be updated. This already has been indicated in Alg. 11 by the call of the procedure UPDATE in each recursion. Therefore, to complete the description of the extension of the GBFOS, the update procedure must be specified.

The update is applied after the "search and prune" procedure. If the node t_p was "pruned" the nodes $t \in \{t : t_p \in \mathcal{S}_t\}$ have to be updated. Since the local points $P_t[i] : 0 \leq i < N$ do not depend on the child nodes of t we have to investigate only the impact of the change on the branch point P_t^b. Apparently, P_t^b can only change if the value of $P_s[f_c(s)]$ of one of the descendants s of t that contribute to P_t^b is changed. The amount of the change is stored in Δ. There are three cases for each node t:

1. $f_c(t) = N$

2. $f_n(t) = N$

Algorithm 11 Search and "prune" algorithm

```
SEARCH(node t)
{
if ($\lambda_{min}(t) \neq \lambda(t)$) then
   if ($\lambda_{min}(t) = \lambda_{min}(\text{left}(t))$) then
      SEARCH(left($t$))
   else
      SEARCH(right($t$))
   end if
   UPDATE($t$)
else
   $\Delta \leftarrow P_t[f_n(t)] - P_t[f_c(t)]$
   $f_c(t) \leftarrow f_n(t)$
   FIND_NEXT_LAMBDA($t$)

end if
$\lambda_{min}(t) \leftarrow \min\{\lambda(t), \lambda_{min}(left(t)), \lambda_{min}(right(t))\}$
}
```

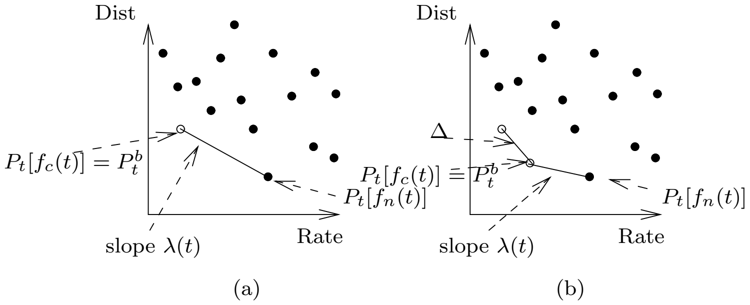


Figure 3.7: Update case 1: (a) before, (b) possible situation after update

3. $f_n(t) < N \wedge f_c(t) < N$

If in the first case P_t^b is changed by Δ then the values of $\lambda(t)$ and $f_n(t)$ must be adapted (Fig. 3.7). This can be done with the procedure FIND_NEXT_LAMBDA (Alg. 10). In the second case, after $P_t^b \leftarrow P_t^b + \Delta$, the new value of $\lambda(t)$ can be computed directly by $\lambda(t) \leftarrow \frac{D_t^b - D_t[f_c(t)]}{R_t^b - R_t[f_c(t)]}$ (Fig. 3.8). In the third case the branch RD-point P_t^b lies, before the update, above the line that is defined by the points $P_t[f_c(t)]$ and $P_t[f_n(t)]$. It must be verified whether the branch RD-point after the update $P_t^b \leftarrow P_t^b + \Delta$ lies below this line or not. If it does, then P_t^b is the next point, i.e., $f_n(t) = N$, after the update (Fig. 3.9). The procedure UPDATE is provided in Alg. 12. In update case 2 and 3, the point $P_t[f_c(t)]$ is not changed. However, this point is the only point that could change a branch point in parent nodes. Thus, in updated case 2 and 3 we set the change amount Δ to zero, $\Delta \leftarrow 0$. This is equivalent to break off the whole update procedure.

Now, we can decide whether t_p is part of the optimal tree or not. If there is a path from the root t_0 to t_p consisting only of internal nodes, then t_p belongs to the optimal subtree ($\mathcal{S}^* \preceq \mathcal{T}$) with the rate $R(\mathcal{S}^*, f_c)$ and the distortion $D(\mathcal{S}^*, f_c)$.

An example of the procedures INIT, SEARCH and UPDATE can be found in Appendix A.

Note that a similar algorithm was proposed in [80], however, without a proof and without the strict following of the GBFOS approach in this thesis.

Algorithm 12 Update

```
UPDATE(node t)
{
$P_t^b \leftarrow P_t^b + \Delta$
if ($f_c(t) = N$) then
    FIND_NEXT_LAMBDA(t)
else
    if $f_n(t) = N$ then
        $\lambda(t) \leftarrow \frac{D_t^b - D_t[f_c(t)]}{R_t^b - R_t[f_c(t)]}$
    else
        if $(-R_t^b * \lambda(t) + D_t^b < -R_t[f_c(t)] * \lambda(t) + D_t[f_c(t)])$ then
            $\lambda(t) \leftarrow \frac{D_t^b - D_t[f_c(t)]}{R_t^b - R_t[f_c(t)]}$
            $f_n(t) \leftarrow N$
        end if
    end if
    $\Delta \leftarrow (0, 0)$
end if
}
```

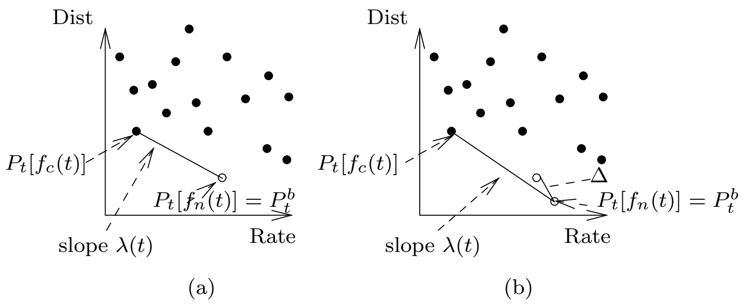


Figure 3.8: Update case 2: (a) before, (b) possible situation after update

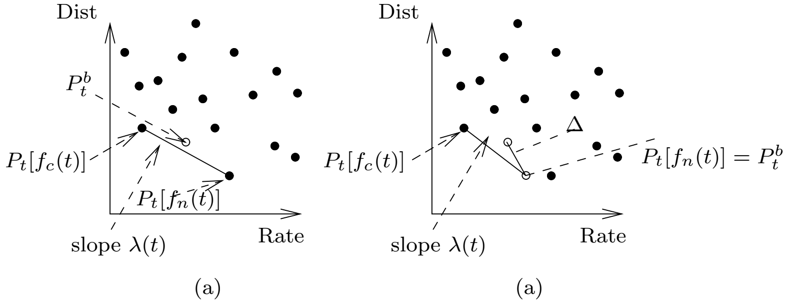


Figure 3.9: Update case 3: (a) before, (b) possible situation after update

3.3.4 Proof of Correctness

In this section, we prove the correctness of our algorithm. We claim that the complete LCH of $\mathcal{P}_{\mathcal{T}}$ can be computed by the algorithm described in Sect. 3.3.3. We prove this in three steps. First, we show that the point on the LCH with the smallest rate is found after the application of INIT (Alg. 9). The second step is to show that SEARCH (Alg. 11) is able to find points on the LCH if it is applied to a data structure already describing a point on the LCH. Finally, we show that the procedure SEARCH is even able to find the edges of the LCH. Then we conclude that the algorithm is able to compute the complete LCH.

Before we will describe some useful terms, we take a look at the data structure (3.16). Some values are fixed during the complete optimization process like left(t), right(t) and $P_t[i] : 0 \leq i < N$ which describe the tree structure and the local points. The other components of (3.16) may have any desired value. We call a concrete allocation of such values a *configuration*.

If the configuration contains reasonable values, e.g., P_t^b contains the sum of the current child nodes, $\lambda(t)$ contains the slope between the current and the next point, etc. In this case, we call the configuration *consistent*. Each consistent configuration of $\mathcal{T}$ defines a tree and selection function pair, $(\mathcal{S}^*, f_c)$. The selection function is defined by the data structure, and the tree $\mathcal{S}^*$ consists of all nodes $t \in \mathcal{T}$ that can be reached from the root by traversing nodes s for which $f_c(s) = N$ is true.

Definition 3 *For every node t the* **forward lambda** $\lambda_f(t)$ *is defined as* $\lambda_f(t) \overset{\text{def}}{=} \min\{\frac{D-D_t[f_c(t)]}{R-R_t[f_c(t)]} : (D,R) \in \{P_t[i] : 0 \leq i \leq N\} \wedge R > R_t[f_c(t)]\}$. *If this set is empty, we set* $\lambda_f(t) \leftarrow \infty$. *Analogously, the* **backward lambda** $\lambda_b(t)$ *is defined as* $\lambda_b(t) \overset{\text{def}}{=} \max\{\frac{D-D_t[f_c(t)]}{R-R_t[f_c(t)]} : (D,R) \in \{P_t[i] : 0 \leq i \leq N\} \wedge R < R_t[f_c(t)]\}$. *If this set is empty, we set* $\lambda_b(t) \leftarrow -\infty$.

Definition 4 *We call a consistent configuration of tree $\mathcal{T}$* **convex**

1. *if there is a λ such that $\forall t \in \mathcal{T} : \lambda_f(t) \geq \lambda \geq \lambda_b(t)$ and*

2. *if $\forall t \in \mathcal{T} : \lambda(t) = \lambda_f(t)$.*

Lemma 2 *Let $(\mathcal{S}^*, f_c)$ be a tree defined by a convex configuration of $\mathcal{T}$. Then $(R(\mathcal{S}^*, f_c), D(\mathcal{S}^*, f_c))$ describes a point on the LCH of $P_{\mathcal{T}}$.*

Proof: $\mathcal{T}$ has a convex configuration. Let λ be taken from Def. 4. Then, if the tree is traversed from the leaves to the root the term $D_t[f_c(t)]+\lambda \cdot R_t[f_c(t)]$ is always the minimum for all possible RD-points $P_t[i] : 0 \leq i \leq N$ including the branch node, i.e., $D_t[f_c(t)] + \lambda \cdot R_t[f_c(t)] = \min\{D_t[n] + \lambda \cdot R_t[n] : 0 \leq n \leq N\}$. Consequently $P_t[f_c(t)]$ lies on the LCH of $\mathcal{P}_{\mathcal{S}_t}$ (see, for example, [82, 71]). In particular, this is true for the root node t_0. Thus, $P_{t_0}[f_c(t_0)]$ apparently describes a point on the LCH of $\mathcal{P}_{\mathcal{T}}$.□

The next lemma shows how to find a convex configuration.

Lemma 3 *Let $\mathcal{T}$ be a tree with root t_0. After applying INIT to a data structure of $\mathcal{T}$, the resulting configuration is convex. Let (S^*, f_c) be defined by this configuration. Then, $(R(S^*, f_c), D(S^*, f_c))$ is the point on the LCH of $P_{\mathcal{T}}$ with the smallest rate.*

Proof: The procedure INIT determines recursively for each node $t \in \mathcal{T}$ the point with the smallest rate. Thus, $P_{t_0}[f_c(t_0)]$ describes the point of $\mathcal{P}_{\mathcal{T}}$ with the smallest rate which is always a point on the LCH. Part 2 of Def. 4 is satisfied by the search of the procedure FIND_NEXT_LAMBDA (Alg. 10). From Def. 3, we conclude $\lambda_b = -\infty$. Then, if we set $\lambda \leftarrow \lambda_{\min}(t_0)$, part 1 of Def. 4 is satisfied. □

The next lemma shows that we can find points on the LCH of $\mathcal{T}$ if we already know a point on the LCH of $\mathcal{T}$.

Lemma 4 *Let $\mathcal{T}$ be a tree with root t_0 and a convex configuration. After applying SEARCH to $\mathcal{T}$, the resulting configuration is still convex.*

Proof: The configuration of $\mathcal{T}$ is convex. This means that we can find a λ that complies with Def. 4. A good candidate is $\lambda_{\min}(t_0)$, thus, we set $\lambda \leftarrow \lambda_{\min}(t_0)$. If we apply SEARCH, first the "search and prune" phase is launched. A node t_p which contains $\lambda(t_p) = \lambda$ is found. We set $f_c^{\text{old}}(t_p) \leftarrow f_c(t_p)$, $\lambda^{\text{old}}(t_p) \leftarrow \lambda(t_p)$ and $\lambda_f^{\text{old}}(t_p) \leftarrow \lambda_f(t_p)$. The algorithm sets $f_c(t_p) \leftarrow f_n(t_p)$ and computes new $f_n(t_p)$ and $\lambda(t_p)$. With the change of $f_c(t_p)$, the values of $\lambda_f(t_p)$ and $\lambda_b(t_p)$ change, too. For the new $\lambda(t_p)$, the condition $\lambda(t_p) \geq \lambda$ must be true. Otherwise, the $\lambda^{\text{old}}(t_p)$ might not have been $\lambda_f^{\text{old}}(t)$ before the "pruning" which is contradicting part 2 of Def. 4. In order to satisfy part 1 of Def. 4, we see $\lambda_f(t_p) = \lambda(t_p) \geq \lambda$ and it remains to be shown that $\lambda_b(t_p) \leq \lambda$. But we know that $\lambda_b(t_p) \geq \lambda$ since λ describes the slope between $P_t[f_c^{\text{old}}(t_p)]$ and $P_t[f_c(t_p)]$. If $\lambda_b(t_p) > \lambda$ we set j to be the argument for which $\frac{D_t[j]-D_t[f_c(t)]}{R_t[j]-R_t[f_c(t)]} = \lambda_b(t_p)$. Then either $R_t[j] > R_t[f_c^{\text{old}}(t_p)]$ which contradicts part 2 of Def. 4 or $R_t[j] \leq R_t[f_c^{\text{old}}(t_p)]$ contradicting part 1 of Def. 4. Thus, we have $\lambda_b(t_p) = \lambda$.

Up to now we have shown that the "pruned" node t_p satisfies convexity after the application of SEARCH. The next step is to show the same for the updated nodes.

In the update phase, only nodes $s \in \{s : t_p \in \mathcal{S}_s\}$ are considered. There are three different cases in the procedure UPDATE (Alg. 12). We show that convexity of node s is maintained in each case.

1. The first case is displayed in Fig. 3.7 on Page 73. First, we set $\lambda^{\text{old}}(s) \leftarrow \lambda(s)$. Since $P_s[f_c(s)] = P_s^b$, $P_s[f_c(s)]$ is changed by Δ and this changes the values of $\lambda(s)$, $\lambda_f(s)$, and $\lambda_b(s)$. Since $\lambda \leq \lambda^{\text{old}}(s)$ and λ is the slope of Δ, we have $\lambda^{\text{old}}(s) \leq \lambda(s)$ after the update, and, thus, we still have $\lambda_f(s) \geq \lambda$. Analogously, one can see that the new $\lambda_b(s)$ satisfies $\lambda_b(s) \leq \lambda$, and, thus, the convexity is preserved.

2. In the second case, we have as similar situation (see Fig. 3.8 on Page 75). The value of $\lambda(s) = \lambda_f(s)$ decreases but still $\lambda(s) > \lambda$. The backward lambda $\lambda_b(s)$ does not change.

3. The third case is depicted in Fig. 3.9. Here, we have two possible situations after P_s^b was changed by Δ.

   (a) If the new point does not "cross" the line defined by $(P_s[f_c(s)], P_s[f_n(s)])$, the convexity is not violated.

   (b) If the new point crosses this line, it is assured that $R_s^b > R_s[f_c(s)]$. Otherwise, the new point must also have crossed the line defined by $P_s[f_c(s)]$ and the slope $\lambda_b(s)$. This means that λ, the slope of Δ, satisfies $\lambda < \lambda_b$. But this contradicts the choice of λ and, thus, the convexity. The new $\lambda(s) = \lambda_f(s)$ becomes smaller, but no smaller than λ since the slope of Δ is λ. Here again, $\lambda_b(s)$ does not change and the convexity is preserved.

In summary, convexity is preserved if the procedure SEARCH is applied on an already convex configuration. □

Corollary 2 *The application of SEARCH on a convex configuration produces a non decreasing sequence* $\lambda^0_{\text{min}}(t) \leq \lambda^1_{\text{min}}(t) \leq \ldots$.

Proof: This follows from the proof of Lemma 4. Taking λ_{min} as λ from Def. 4 again leads to a convex configuration after the "pruning" and update. □

The next lemma shows that the procedure SEARCH determines edges of the LCH of $\mathcal{T}$.

Lemma 5 *Let $\mathcal{T}$ be a tree with root t_0 and with a convex configuration. Let $P_k = (D_k, R_k) \overset{\text{def}}{=} \arg\min_{(D,R)}\{R : (D,R) \in LCH\ of\ P_{\mathcal{T}} \wedge R > R_{t_0}[f_c(t_0)]\}$ and let us assume that this set is not empty. Then after finitely many successive applications of SEARCH on $\mathcal{T}$, P_k is determined.*

Proof: Let $\lambda = \frac{D_k - D_{t_0}[f_c(t_0)]}{R_k - R_{t_0}[f_c(t_0)]}$. First, we show that the algorithm stops after a finite number of steps. We denote $\lambda_{\min}(t_0)$ that is achieved after n applications of the procedure SEARCH with $\lambda^n_{\min}(t_0)$. From Corollary 2 we know $\lambda^0_{\min}(t_0) \leq \lambda^1_{\min}(t_0) \leq \lambda^2_{\min}(t_0) \leq \ldots$.

Furthermore, we know that after a finite number of steps , say n, $\lambda^n_{\min} = \infty$. This can be seen from the fact that every application of SEARCH increases $R_u[f_c(u)]$ for a node u and $R_u[f_c(u)]$ can be either a fixed local point $P_u[i]$, $0 \leq i < N$ or the sum of fixed local points of a finite number of children. The fact that every node has only a finite number of local points concludes the assertion.

Now let be $\lambda^0_{\min}(t_0) < \lambda^1_{\min}(t_1) < \lambda^2_{\min}(t_0) < \ldots$ without loss of generality. We apply the procedure SEARCH m times until $\lambda^{m-1}_{\min}(t_0) < \lambda \leq \lambda^m_{\min}(t_0)$. From Lemma 4 we conclude that the configuration is still convex. After $m-1$ steps we can be sure that $f_c(t_0)$ has not changed. If it had changed, the slope of the change would be smaller than λ, which either means that we have found a new point on the LCH of $\mathcal{P}_{\mathcal{T}}$ with smaller rate than P_k or the point P_k is not on the LCH. This contradicts the choice of P_k. On the other hand, after m steps we can be sure, that $f_c(t_0)$ changes. If it does not, we can conclude that the point P_k can not be achieved at all. This is due to the fact that such a step must be induced by $\lambda(t_0)$ or a Δ defined in any node t_p. But $\lambda(t_0) > \lambda^m_{\min}(t_0)$ and the slope of $\Delta = (\Delta R, \Delta D)^t$ is $\lambda(t_p)$ which must satisfy $\lambda(t_p) > \lambda^m_{\min}(t_0)$ if $\Delta R > 0$ and $\lambda(t_p) < \lambda^m_{\min}(t_0)$ if $\Delta R < 0$. Thus, no consistent configuration can cross the line defined by $P_{t_0}[f_c(t_0)]$ and $\lambda^m_{\min}(t_0)$. Thereby, the lemma has been proven. □

Proposition 7 *If the tree $\mathcal{T}$ is initialized with INIT and if SEARCH is applied on $\mathcal{T}$, the complete LCH of $\mathcal{P}_{\mathcal{T}}$ is computed.*

Proof: Let $\mathcal{P}^{\text{alg}}_{\mathcal{T}}$ be the set of computed points by our algorithm. Further, let $\mathcal{P}_{\mathcal{T}}^{\mathrm{LCH}}$ be the LCH of $\mathcal{P}_{\mathcal{T}}$. From Lemma 3 and 4 it follows that $\mathcal{P}_{\mathcal{T}}^{\mathrm{alg}} \subset \mathcal{P}_{\mathcal{T}}^{\mathrm{LCH}}$. From Lemma 3 and 5 we can conclude $\mathcal{P}_{\mathcal{T}}^{\mathrm{LCH}} \subset \mathcal{P}_{\mathcal{T}}^{\mathrm{alg}}$. Altogether, we have $\mathcal{P}_{\mathcal{T}}^{\mathrm{LCH}} = \mathcal{P}_{\mathcal{T}}^{\mathrm{alg}}$. □

3.4 Optimization with General Dependences

Up to now, we considered only dependences that can be treaded very efficiently. We now investigate more general dependences.

Let $I = (i_0, \ldots, i_{M-1})$ be the vector of the selected encoding indices. The general rate allocation problem can be formulated as

$$\min_{I \in \mathcal{I}^M} \sum_{m=0}^{M-1} D_m^I \quad \text{subject to} \quad \sum_{m=0}^{M-1} R_m^I \leq R_T.$$

Thus, the set of local RD-points is also dependent on I, $\mathcal{P}_m(I) = \{(R_m^I, D_m^I) : I \in \mathcal{I}^M\}$.

As before, the constrained problem can be transformed into an unconstrained problem using a Lagrangian multiplier. An LM-algorithm to solve the unconstrained problem in the general case is proposed in [71]. Here, we present the algorithm for an important special case if the rates and distortions of $\mathcal{P}_m(I)$ depend only on the previous index, i_{m-1}, and the current index, i_m [92, 58].

Such dependences can occur, e.g., in the video compression standard H.263 [42]. This is due to the encoding method of the quantizer factor Q. The quantizer factor, Q_m, can be adjusted for each macroblock m. In order to encode Q_m efficiently, the quantizer factor, Q_m, is encoded with differential coding. Since the difference is computed considering the previous macroblock $m-1$, the encoding costs of macroblock m depend on the quantizer factor Q_{m-1}. A similar dependence is given in the encoding of motion vectors. In this case, however, the differential encoding of the current motion vector depends on several previous macroblocks.

We now describe a single LM-iteration. As before, the full LM-algorithm derives from the LM-iteration and the search for the value of λ that yields the

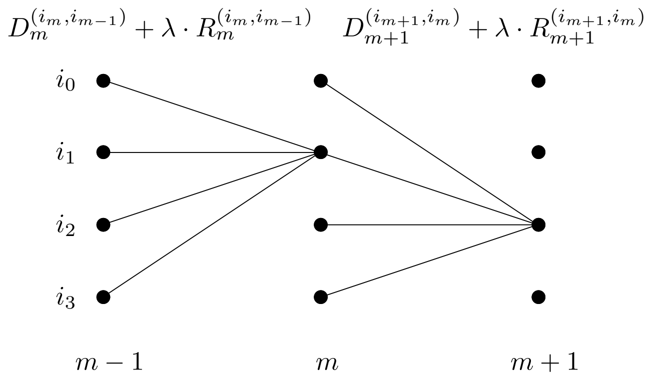

$D_m^{(i_m,i_{m-1})} + \lambda \cdot R_m^{(i_m,i_{m-1})}$
$D_{m+1}^{(i_{m+1},i_m)} + \lambda \cdot R_{m+1}^{(i_{m+1},i_m)}$
i_0
i_1
i_2
i_3
$m-1$
m
$m+1$

Figure 3.10: Viterbi algorithm for dependent RD-points.

optimal LCH solution. If we assign each quantizer step size Q_m to an index $i_m \in \mathcal{I}$, the unconstrained problem can be written as

$$\min_{I \in \mathcal{I}^M} \sum_{m=0}^{M-1} D_m^{(i_m,i_{m-1})} + \lambda \cdot R_m^{(i_m,i_{m-1})}$$

This problem can be solved by the Viterbi algorithm (cf. Sect. 3.2.1). The Viterbi trellis is depicted in Fig. 3.10. Each node $d(n,j)$ contains the smallest cumulative Lagrangian cost resulting in state $i_n = j$,

$$d(n,j) = \min_{i_m \in \mathcal{I}: 0 \leq m < n \wedge i_n = j} \sum_{m=0}^{M-1} D_m^{(i_m,i_{m-1})} + \lambda \cdot R_m^{(i_m,i_{m-1})}.$$

The transitional costs of the trellis from node $(m-1, i_{m-1})$ to (m, i_m) are given by the term $D_m^{(i_m,i_{m-1})} + \lambda \cdot R_m^{(i_m,i_{m-1})}$.

Therefore, a single LM-iteration, if computed with the Viterbi algorithm, has the complexity $O(M \cdot N^2)$.

To the best of the author's knowledge, there is no algorithm known that solves the general dependence rate allocation problem by incremental computation of the convex hull. Thus, such kind of problems can only be solved by the Lagrangian multiplier algorithm.

Chapter 4

Adaptive Vector Quantization and Video Coding

In this chapter, we will evaluate adaptive vector quantization (AVQ) in the context of image sequence coding. To do so, we develop several codecs to analyze AVQ combined with state-of-the-art coding concepts.

The first codec is designed to meet real-time constraints. This codec is presented in Sect. 4.1.1. We proceed to analyze the performance of the ad hoc encoding strategy of the real-time codec by employing a rate-distortion (RD) optimization technique (Sect. 4.1.3). Subsequently, in Sect. 4.2, the AVQ approach is applied in the wavelet transform domain. Furthermore, in order to encode the transform coefficients more efficiently, we consider adaptive partition techniques (Sect. 4.3). The last study, presented in Sect. 4.4, combines motion compensation and AVQ.

The description of our encoding framework can be divided into four stages:

1. Preprocessing
2. Encoding modes
3. Encoding parameters
4. RD-optimization

The *preprocessing* covers all steps needed to create appropriate vectors, e.g., wavelet transform, motion compensation, and the transformation process from a frame to a vector sequence. The *encoding modes* describe all possible modes that can be used to encode a vector and how to transmit the corresponding information, if necessary. This stage does **not** describe how to chose the modes. This is the purpose of the RD-optimization stage later on. The third stage, *encoding parameters*, specifies typical AVQ parameters like $\alpha, \beta, \gamma, \mathcal{C}^*$, details of the scalar quantization, variable-length-coding (VLC), and the handling of the codebook update. The *RD-optimization* translates the encoding modes into a rate-allocation problem that can be solved by appropriate rate-allocation algorithms from Chapter 3.

Now we outline these stages for the codecs in this chapter.

Preprocessing. The preprocessing stage is different for the several codecs presented in this chapter. The first step is always to get a YUV representation of the frame. The frame size is, if not otherwise stated, QCIF (176 × 144). There is almost no further preprocessing in the real-time codec in Sect. 4.1.1. The codec described in Sect. 4.2 uses a wavelet transform and regroups the transform coefficients. An additionally partition of the frame by quad-trees is made in the preprocessing in Sect. 4.3. And, finally, motion compensation is applied in the preprocessing stage of Sect. 4.4.

Encoding modes. In principle, the encoding modes are the same for all codecs. There are three encoding modes for each vector x_m:

Mode 0. Replenishment mode. A vector in the previously decoded frame from the same position as x_m is taken to encode x_m.

Mode 1. VQ mode. In this mode, a suitable vector from the codebook $\mathcal{C}$ is taken to represent x_m.

Mode 2. Update mode. A vector $c^* \in \mathcal{C}^*$ is taken to represent x_m, transmitted as side information, and inserted in the codebook $\mathcal{C}$.

Encoding parameters. The codebook size is fixed during the encoding. Thus, if new vectors are inserted into the codebook we need a replacement strategy. Usually, we employ a strategy similar to the least-frequently-used (LFU) strategy, even though we present experiments with other strategies.

The encoding of vector c^* from the universal codebook, $\mathcal{C}^*$, is, in principle, made with scalar quantization of the components. If it is appropriate, we use a DPCM scheme.

RD-optimization. The encoding mode ω_m for each vector x_m must be selected during the RD-optimization. According to the different encoding options of vectors, several sets of RD-points have to be created and are passed to an appropriate RD-optimization technique.

4.1 Basic Codec

4.1.1 Real-Time Codec

This section describes the basic codec structure due to [7, 70].

Preprocessing. The preprocessing stage decomposes the luminance frame into blocks of 4×4 pixel size. The blocks are scanned, using a spiral scheme, in order to get 16-dimensional vectors. After the scanning, the means, μ_m, of the vectors, x_m, are computed.

Encoding modes. It is assumed that a previously decoded frame, $x_0^r, \ldots, x_{M-1}^r$, (reference frame) is known to the encoder and decoder. For each vector x_m, an approximating vector $\hat{x}_m$ shall be created. The three encoding modes are realized as follows:

Mode 0 Vector x_m is encoded by $\hat{x}_m \leftarrow x_m^r$.

Mode 1 Vector x_m is encoded with MRVQ. Then the mean μ_m is quantized, $\hat{\mu}_m = Q_\mu(\mu_m)$; x_m is encoded by $\hat{x}_m \leftarrow Q(x_m - \hat{\mu}_m \mathbf{1}_L) + \hat{\mu}_m \mathbf{1}_L$.

Mode 2 Vector x_m is encoded with a proper vector $c_m^* = Q^*(x_m - \hat{\mu}_m \mathbf{1}_L) \in \mathcal{C}^*$ and $\hat{\mu}_m$, i.e., $\hat{x}_m \leftarrow \hat{\mu}_m \mathbf{1}_L + c_m^*$.

Here, $\mathbf{1}_L$ denotes the vector $(1, \ldots, 1)^t$ of dimension L. The resulting modes are collected in a mode map $\Omega = (\omega_0, \ldots, \omega_{M-1}) : \omega_m \in \{0, 1, 2\}$. The positions of the vectors encoded in mode 1 or 2 are sent to the decoder using run-length encoding. For each vector x_i encoded in mode 1 or 2, additional information has to be sent. First, a flag indicating the mode is transmitted. Then, in both cases we send an index for the scalar quantized mean $\gamma_\mu(\alpha_\mu(\mu_m))$. If x_m is encoded in mode 1 we also have to send an index of the quantized shape vector $\gamma(\alpha(x_m - \hat{\mu}_m \mathbf{1}_L))$. For mode 2, we send $\gamma^*(\alpha^*(x_m - \hat{\mu}_m \mathbf{1}_L))$.

Encoding parameters. By reasons of encoding efficiency, the codebook of the adaptive vector quantizer is organized as follows. Every vector in the codebook $\mathcal{C}$ has got a frequency count (FC) $f : \mathcal{C} \rightarrow \mathbb{N}$ that is responsible for the order of the vectors in the codebook. The vector decoder β satisfies

$$i < j \Rightarrow f(\beta(i)) \geq f(\beta(j)). \tag{4.1}$$

Therefore, the vectors in $\mathcal{C}$ can be ordered where the vector with the highest FC, $\beta(0)$, can be found at the top of the codebook and the vector with the lowest FC, $\beta(N-1)$, at the end of the codebook. When a vector $c_i = \beta(i)$ is used for mode-1 encoding, the frequency count of this vector is incremented by 1, i.e., $f(c_i) \leftarrow f(c_i) + 1$. Then the vector decoder β is rearranged to re-establish the condition (4.1). Thus, the vector c_i is moved upwards to the top of the codebook. If a vector $c_m^* = Q^*(x_m)$ is used for mode-2 encoding, the new vector c_m^* has to be inserted in the codebook. For this task, the new vector requires a frequency count. Empirically, the value $f(\beta(\lfloor \frac{N}{2} \rfloor)) + 1$ has proven a good performance, i.e., we take the FC of the vector in the middle of the codebook and increment it by 1. Since the codebook size should stay constant, a vector of $\mathcal{C}$ has to be removed. In this codec, we scheduled the vector with the smallest frequency count to be removed. After that, the vector decoder, β, has to be recomputed to maintain (4.1).

The vector encoder, α, realizes a nearest neighbor quantizer (according to the Euclidean norm) with additional speed up techniques that are described in detail in [7]. The index coder, γ, is a Huffman coder. The corresponding probabilities have been estimated using the test sequences *Miss America*, *Mother & Daughter*, and *Salesman*. The codebook size is $N = 512$ and the vector dimension is $L = 16$.

The vectors of the universal codebook $\mathcal{C}^*$ are encoded as follows. The quantization $Q^*(x)$ is done by encoding the shape vector x with DPCM. The corresponding scalar quantizer Q_d has binsize 8 and deadzone 8. Furthermore, the codebook size is $N_d = 64$. The parameters for Q_μ are binsize 4, deadzone 8, and $N_\mu = 64$. The scalar quantizers Q_d and Q_μ have been empirically optimized in [7].

RD-optimization. Since the objective of the codec is to meet real-time constraints, the selection of the encoding modes is not RD-optimized. We rather use a fast ad hoc rate control scheme working as follows. The mean squared error (MSE) between the vectors in the reference frame $x_0^r, \ldots, x_{M-1}^r$ and the original frame $x_0, \ldots, x_{M-1}$ is computed, i.e., $e_m^0 = \frac{1}{L}\|x_m - x_m^r\|_2^2$, $0 \leq m < M$. Then the vectors x_m are sorted with respect to decreasing errors, e_m^0, that is, with the highest errors first. This priority queue is processed as long as the target bit budget R_T or the target time budget T_t is exhausted. The vectors that have not been processed up to that time are encoded in mode 0. The other vectors are either encoded in mode 1 or in mode 2. The decision whether a vector is encoded in mode 1 or 2 is made by a threshold value *tol*. Let e_m^1 be the MSE if x_m is encoded in mode 1. Then, x_m is encoded in mode 1 if $e_m^1 \leq$*tol*, otherwise, mode 2 is selected. The tolerance *tol* is computed with

$$tol = \max(30, \min(e^d, 150))$$

where e^d describes the MSE of the difference between the current frame (x_0, …, x_{M-1}) and the reference frame ($x_0^r, \ldots, x_{M-1}^r$).

Now we describe the encoding of the UV-components. The chrominance components are not treated with AVQ. We rather apply only scalar quantization. First, the UV components are subsampled in order to get one U- and one V-component per luminance vectors. Then for each UV-pair it is decided if it should be encoded or not. This decision is made in a similar fashion to the vectors of the luminance component. A priority queue is created according to the MSE of the original and the reference frame. The chrominance components are processed until a bit budget R_T^{UV} or a time budget T_t^{UV} is exhausted. The UV-components are quantized and transmitted by Q_μ and γ_μ respectively. The resulting modemap Ω^{UV} is encoded with run-length encoding. The bitrate spent for the encoding of the chrominance components is constrained to about $\frac{1}{10}$ of the bitrate for the complete frame.

4.1.2 Experiments with the Codebook Organization

In this subsection we investigate the importance of the codebook organization for the coding performance. In the codebook organization of the last section, a least-frequently-used (LFU) strategy was employed to determine the vectors to be removed. Furthermore, the mode 2 vectors were inserted in the middle of the codebook according to the order given by the frequency count. Other strategies are conceivable, e.g.,

- the just mentioned LFU removal of vectors with the insertion in different areas of the codebook, e.g., in the upper quarter,
- a least-recently-used (LRU) removal strategy for the vectors, and
- a theoretically optimal insertion strategy developed in [95] combining LRU and LFU strategies.

In the following, we describe experiments in codebook organization. For this, we substitute the codebook management of the codec introduced in Sect. 4.1.1 by several strategies. We investigate six different update strategies.

1. Removal of the LFU vectors and insertion in the middle of the codebook. The frequency count of a vector is increased by 1 each time it is used for mode-1 encoding. Vectors encoded in mode 2 get the frequency count $f(\beta(\lfloor\frac{N}{2}\rfloor)) + 1$.
2. Removal of the LFU vectors and insertion in the lower quarter of the codebook. The frequency count of a vector is increased by 1 each time it is used for mode-1 encoding. Vectors encoded in mode 2 get the frequency count $f(\beta(\lfloor\frac{N*3}{4}\rfloor))$.

3. Removal of the LFU vectors and insertion in the upper quarter of the codebook. The frequency count of a vector is increased by 1 each time it is used for mode-1 encoding. Vectors encoded in mode 2 get the frequency count $f(\beta(\lfloor \frac{N}{4} \rfloor))$.

4. LFU update strategy published in [70]. The frequency count of a vector is increased by 1 each time it is used for mode-1 encoding. Vectors encoded in mode 2 get the frequency count $f(\beta(N-1)) + (f(\beta(0)) - f(\beta(N-1)))/4$ rounded to an integer.

5. Removal of LRU vectors with a move to front strategy [23].

6. Update strategy published in [9]. This update strategy is based on the *gold washing* method in [95]. The codebook is divided into two parts (part-1 and part-2 of the codebook). Part-1 of the codebook contains vectors that have been proven to match the current statistical characteristic of the image sequence. Part-2 contains the vectors that have been recently inserted and must prove their match for the current statistical properties of the image sequences. Part-1 is organized with the move to front method. If a vector is encoded in mode 1, it is inserted at the top of the codebook. The other vectors are pushed down by one position. The vectors in part-2 of the codebook have got a frequency count. Each time a vector is encoded in mode 1, its frequency count is increased by 1. Each time a vector is encoded in mode 2, it is inserted at the top of part-2 with frequency count 0. The other vectors in part-2 are pushed down by one position. The frequency count of the previously last vector is compared with a threshold θ. If it is smaller than θ, it is discarded. Otherwise, this new vector is inserted at the top of part-1. The other vectors are pushed down by one position. The last vector is discarded. This codebook structure is illustrated in Fig. 4.1.

We start with a comparison of the first four LRU strategies. The codec is run for several image sequences and bitrates. The mean PSNR is computed dropping the first 15 frames in order to give the codec time to initialize its codebook. The results are shown in Fig. 4.2.

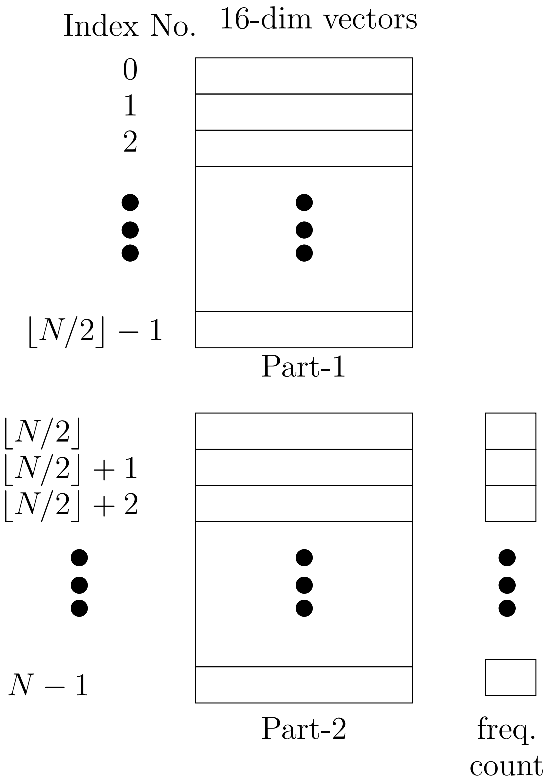


Figure 4.1: Codebook parts in gold washing algorithm due to [9].

The results for the LRU strategies 1–3 are nearly indistinguishable. Therefore, we are free to take strategy 1 as representative for these LRU-strategies to make the following comparisons easier. Strategy 4 performs significantly worse than the other three.

The same experiments for comparison of the update strategies 1–4 are repeated for the gold washing strategy 6 with different thresholds θ. Figure 4.3 shows the results for $\theta = 1$ and $\theta = 2$. We see that the strategy with $\theta = 1$ performs slightly better compared with $\theta = 2$. This suggests taking the strategy with $\theta = 1$ as a representative for strategy 6.

We proceed to compare the performance of all update strategies. This is shown in Fig. 4.4. Apparently, the update strategies perform very similar except for strategy 4.

It remains to account for the deviating behavior of strategy 4 by carrying out the following experiment. Each time a new vector is inserted in the codebook with mode 2, it receives a time stamp, i.e., the frame number is stored. If a vector is encoded with mode 1, the difference between the current frame and the time stamp of this vector is computed. This value is called *age* of a vector. The mean age of mode-1 encoding is computed per frame, excluding the most frequent vector $\beta(0)$ that is usually the zero vector with the time stamp 0. The results are shown in Fig. 4.5. The mean age of strategy 1 is increasing with time. In contrast, the mean age of strategy 4 is oscillating around 10. In the former case, the codec uses for mode-1 encoding vectors that have been inserted more than 100 frames ago. In the latter case, vectors of 10 frames in the past are used on the average throughout.

An analysis of the two update rules reveals a large difference between the frequency counts of the first and the second vectors, $\beta(0)$ and $\beta(1)$. Even though the value $\beta(0) - \beta(N-1)$ is divided by four, the resulting FC is often higher than $\beta(1)$. This leads to a move-to-front update for the new vectors encoded in mode 2. But in contrast to the move-to-front strategy 5 the vectors encoded in mode 1 are still updated by an increment of the FC. Thus, the new vectors inserted in mode 2 displace the old vectors. No vector, except $\beta(0)$, stays in the codebook any longer.

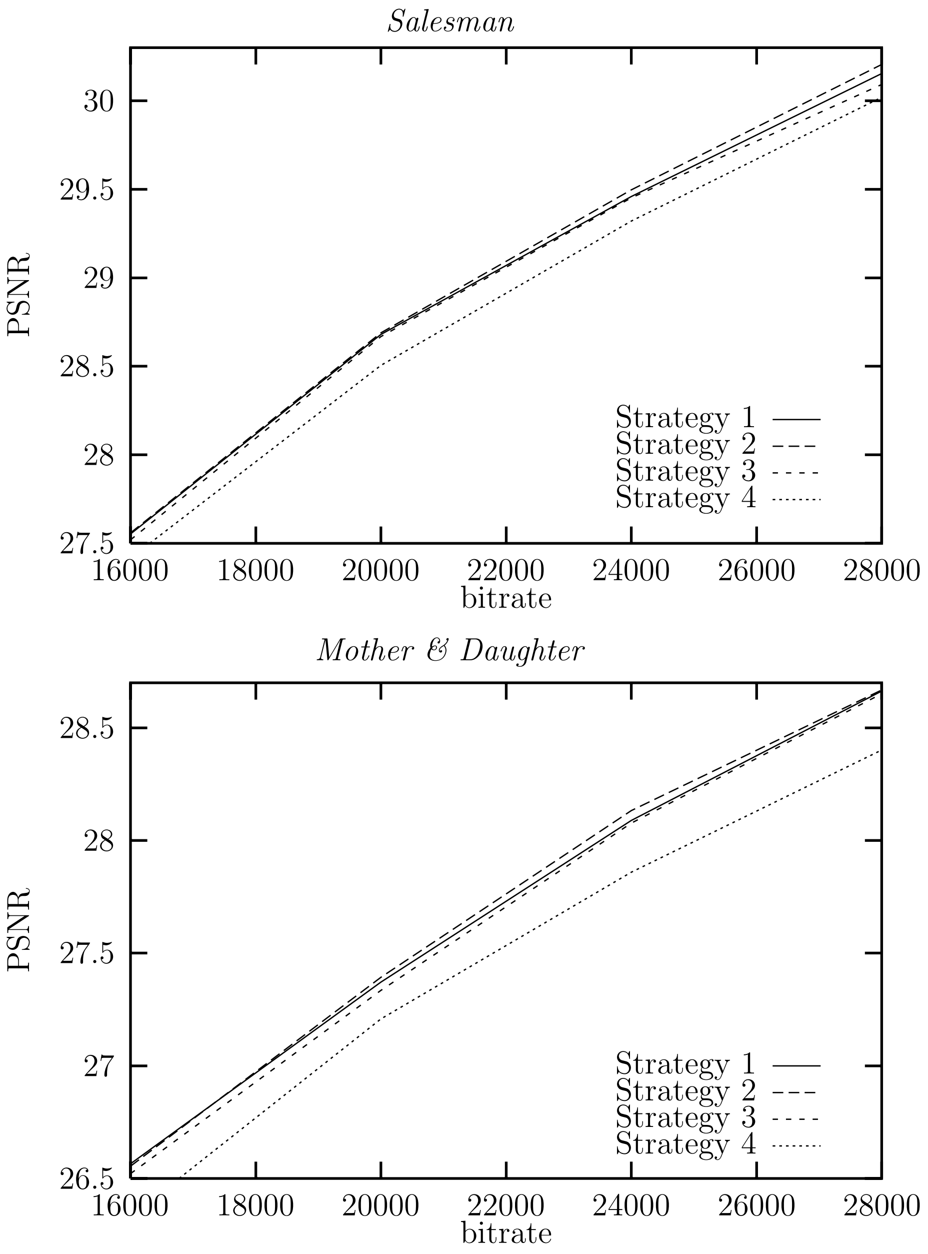
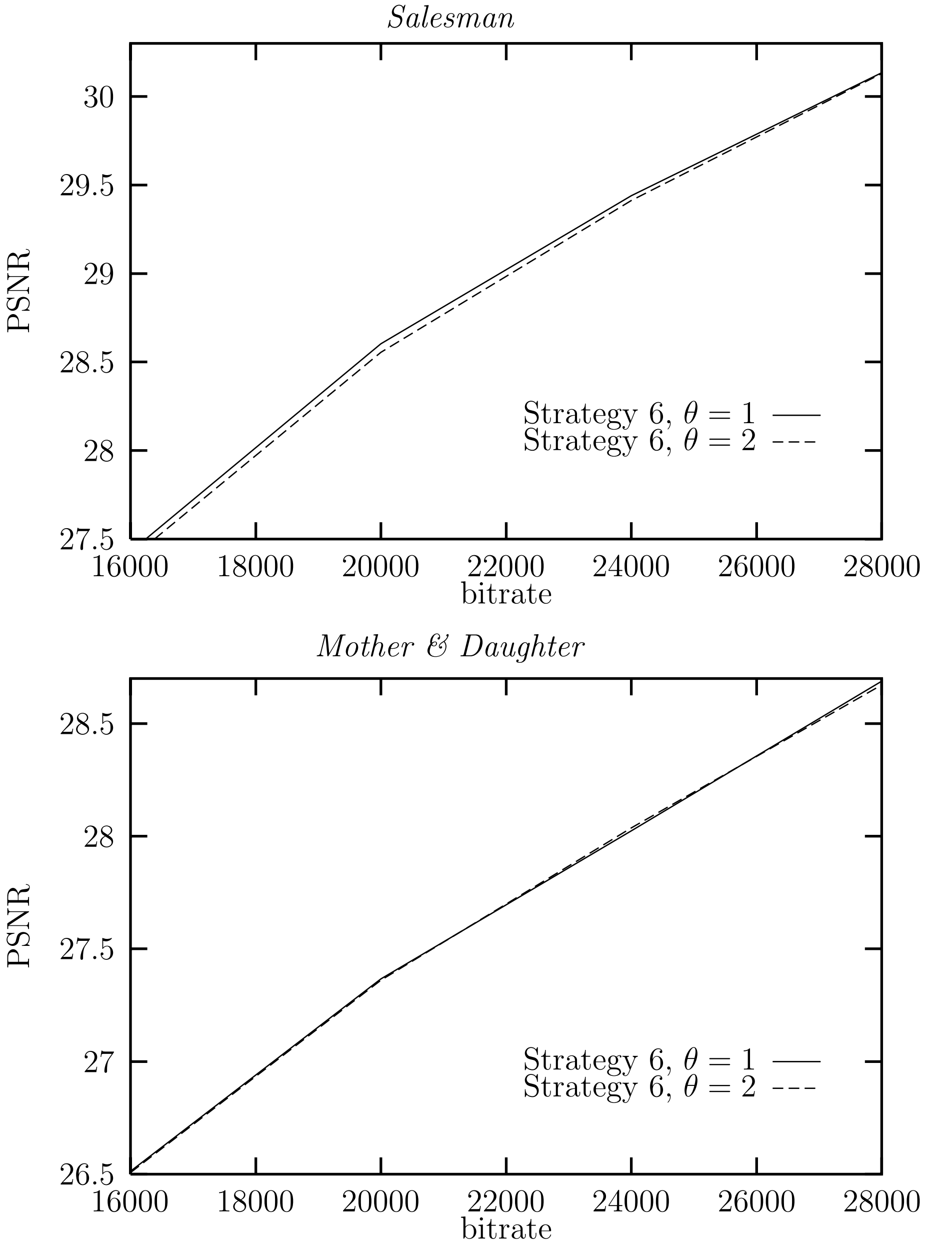


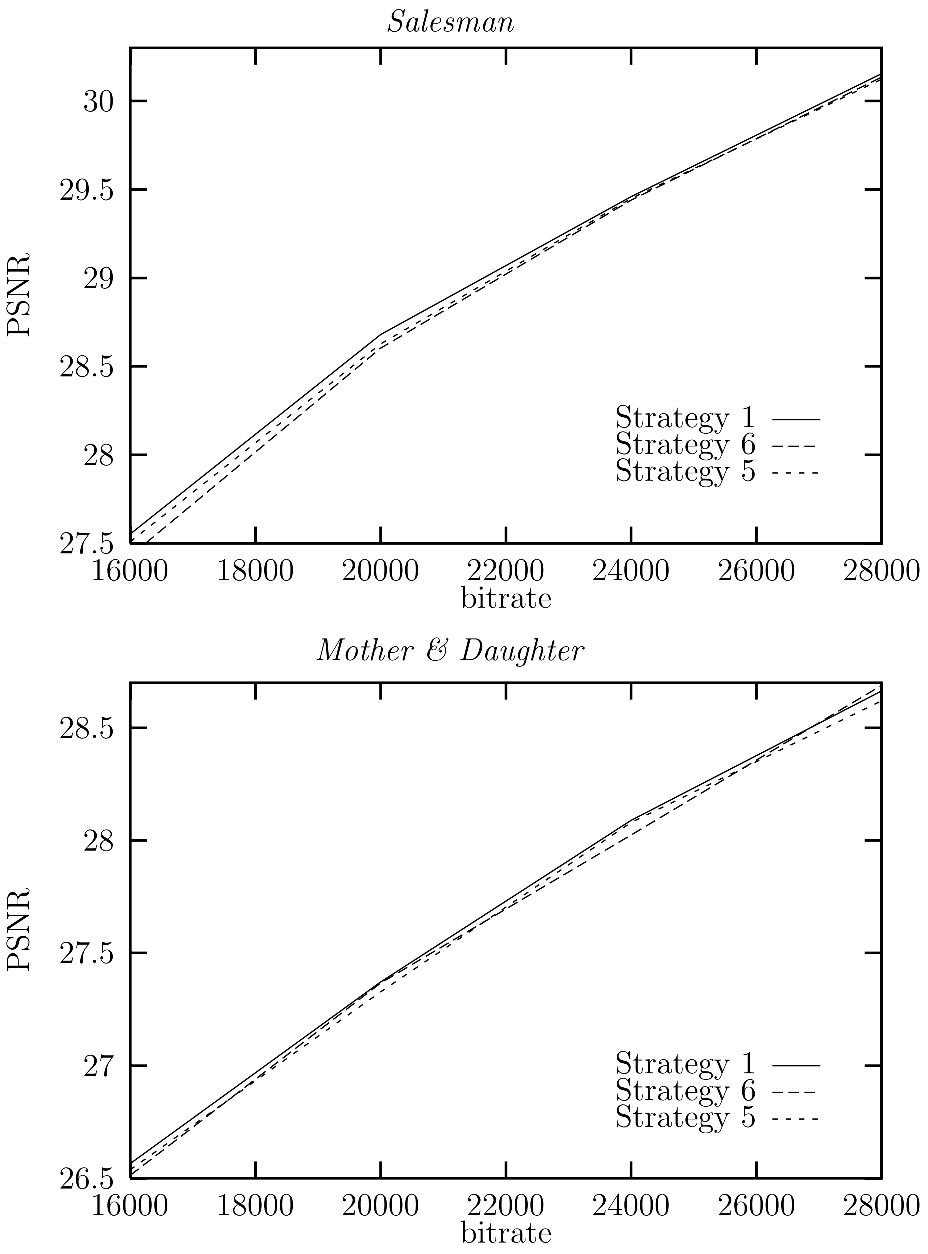
Figure 4.2: Experiments with different LRU update strategies.



Salesman

Mother & Daughter

Figure 4.3: Experiments for different thresholds following the GW update strategy.



Salesman

Mother & Daughter

Figure 4.4: Comparison of all update strategies.

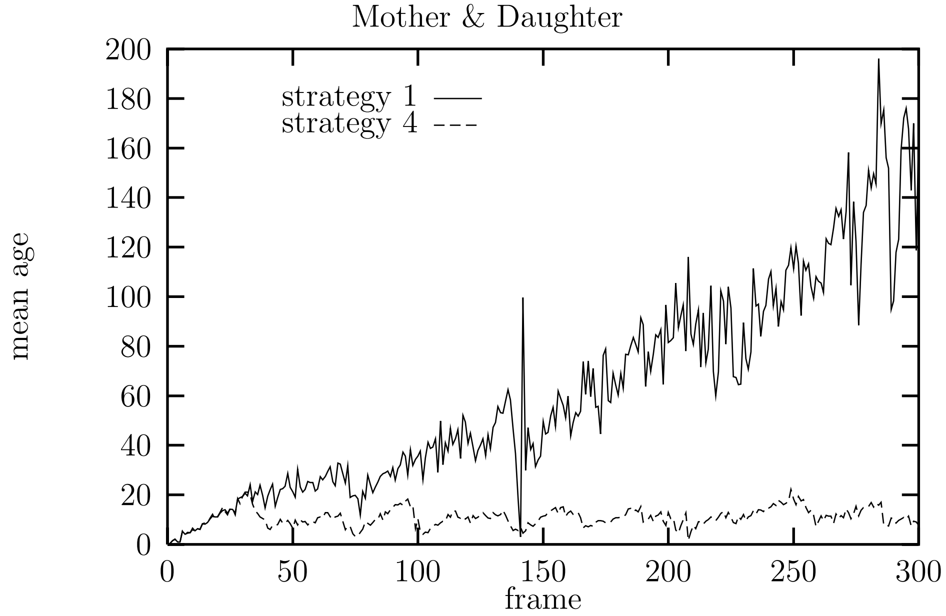


Figure 4.5: Mean age of vectors encoded in mode 1. (28000 bit/s).

A second experiment with the time stamp can be seen in Fig. 4.6. Here, a histogram of the time stamps of the codebook vectors was computed after all frames have been encoded.

It can be seen that the last codebook for strategy 1 contains vectors from all over the sequence. The last codebook of strategy 4 contains mainly vectors of the last 20 frames.

This supports the assumption that the inserted vectors completely displace the old vectors after only a few frames.

The experiments show that the codebook management strategy is, if carefully chosen, of no importance for the performance of the codec. Thus, in the following we use strategy 1 if not otherwise stated.

4.1.3 RD-Optimization of the Real-Time Codec

In Sect. 4.1.1, a fast encoding strategy was used to guarantee the real-time property of the codec. Since we want to evaluate the efficiency of this ad hoc strategy, we need an optimized codec for comparison. Thus, in this section, we develop an RD-optimized version of the real-time codec due to [36]. How-

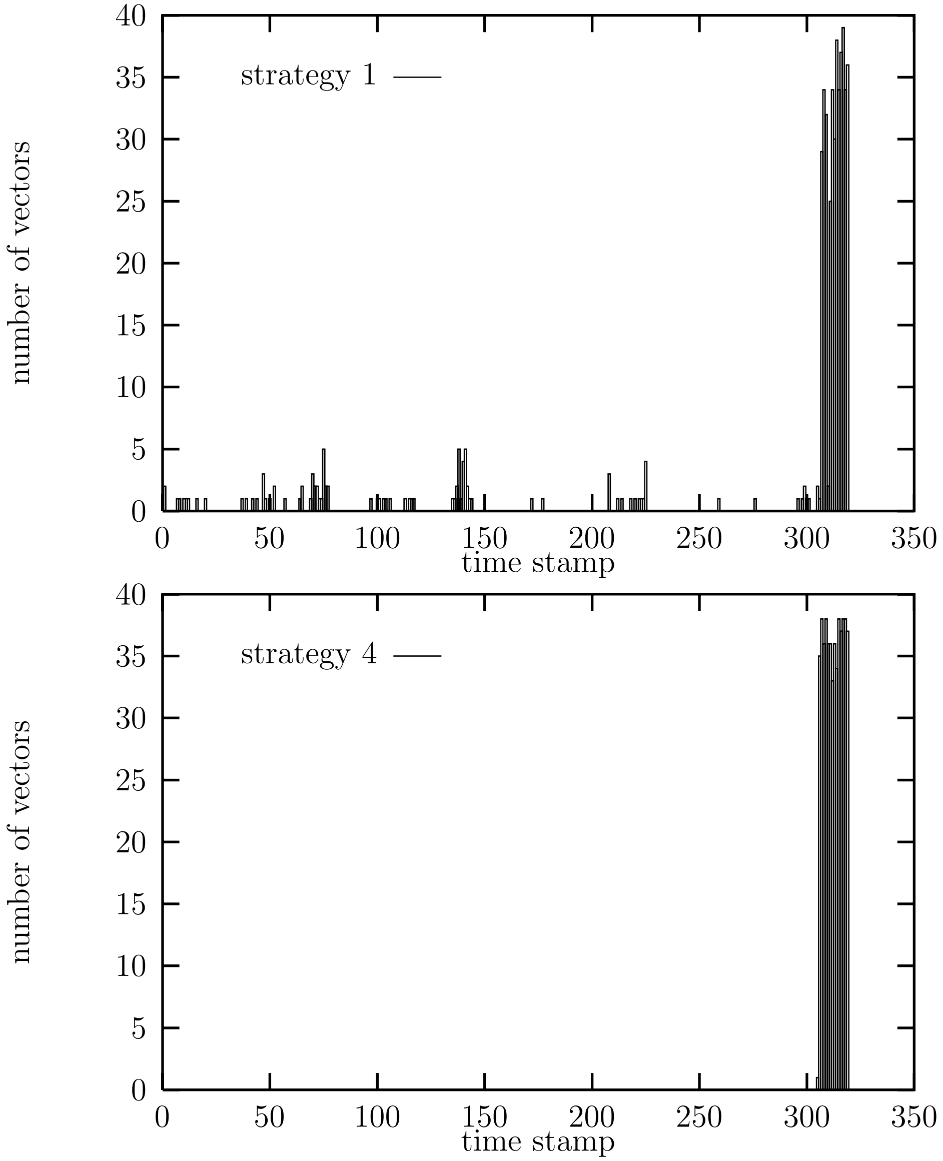


Figure 4.6: Time stamps in the last codebook for *Mother & Daughter*

ever, the RD-optimization technique imposes some restrictions on the codec structure. Sect. 4.1.1 shows that efficient RD-optimization techniques assume certain kinds of dependences between the sets of RD-points, $\mathcal{P}_m$, that can be derived for every vector, x_m, in the frame and the selection index vector, I. The dependences may only become hierarchical at the most.

Let us now reconsider what happens if the real-time codec encodes a vector x_m. If the vector is encode in mode 1, the corresponding codebook vector can change its codebook position. If the vector is encoded in mode 2, it is inserted in the codebook, a vector is replaced, and the codebook positions of some other vectors are changed. Therefore, in each case the codebook structure, and thus, the rate for the encoding of the following vectors can be changed. This leads to a general dependence since the encoding of a block depends on all preceding blocks of the frame. But this kind of dependence can not be handled reasonable by RD-optimization techniques. Thus, we have to change the codec structure in order to allow RD-optimization.

In the following we describe the RD-optimized codec.

Preprocessing. The preprocessing is made as described for the real-time codec.

Encoding modes. The three encoding modes and the mode map encoding are identical, as well.

Encoding parameters. Some of the encoding parameters are different to Sect. 4.1.1. The vector encoder, α, is now realized by using an RD-measure. Fixing an RD-trade-off, λ, that will be determined in the RD-optimization stage, α can be described by $\alpha(\lambda, x) = \arg\min_{i \in \mathcal{I}}[d(x, \beta(i)) + \lambda \cdot \gamma(i)]$ where $d(x, y) = \|x - y\|_2^2$.

The codebook organization is also slightly different. Each time a vector $c_i \in \mathcal{C}$ in Sect. 4.1.1 was used for mode-1 encoding, its frequency count $f(c_i)$ was incremented by 1, and the vectors in the codebook were rearranged with respect to f. Furthermore, each time a new vector c_m^* was encoded in mode 2, it was immediately inserted in the codebook. This strategy does not work here for reasons discussed above. Therefore, a similar strategy, more suitable to the RD-optimization, is applied. The order of the vectors in $\mathcal{C}$ is not changed during the encoding of a frame. Each time a vector in the codebook is used for mode-1 encoding, its FC is incremented by 1 without changing the codebook order. Each time a vector is encoded in mode 2, it is stored separately and cannot be accessed again during the encoding of the frame. When the encoding of a frame is completed, the codebook is updated. At first, the vectors are sorted with respect to the new FC. Let k vectors be encoded in mode 2. Then, the k vectors with the least frequency counts are removed from $\mathcal{C}$. The k new vectors get the frequency count $f(\beta(\lfloor \frac{N}{2} \rfloor)) + 1$ and are inserted in the codebook. After this procedure is finished, the codebook is sorted again with respect to the FC.

RD-optimization. The RD-optimization, i.e., the computation of λ, can be made with the incremental computation of the convex hull for independent RD-sets (see Alg. 6). For all M vectors x_m in the frame, the sets $\mathcal{P}_m$ have to be computed.

We consider the construction of $\mathcal{P}_m$ for the encoding of an arbitrary vector x_m. Therefore, all possible encoding modes for this vector are considered. For mode 0, there is only one RD-point. If x_m is represented by x_m^r, we have the distortion $d(x_m, x_m^r)$, where $d(x, y) = \|x - y\|_2^2$, and no encoding costs. The corresponding RD-point is $(R_m^0, D_m^0) = (0, d(x_m, x_m^r))$. For mode 1, there are N RD-points. For each vector $\beta(i)$, $0 \le i < N$, there is a distortion $D_m^{i+1} = d(x_m, \beta(i) + \beta_\mu(\alpha_\mu(\mu_m))\mathbf{1}_L)$ and a rate $R_m^{i+1} = \gamma(i) + \gamma_\mu(\alpha_\mu(\mu_m)) + 1$ where the addend 1 comes from the flag indicating mode 1 (mode flag). The last RD-point for $\mathcal{P}_m$ is given by mode-2 encoding. Therefore, the encoding cost for the DPCM encoding of the vector, the mean encoding, and the mode flag, $R_m^{N+1} = \gamma^*(\alpha^*(x_m)) + \gamma_\mu(\alpha_\mu(x_m)) + 1$, and the remaining distortion, $D_m^{N+1} = d(x_m, \beta^*(\alpha^*(x_m)) + \beta_\mu(\alpha_\mu(\mu_m))\mathbf{1}_L)$ are taken. Note that the encoding of the positions of the mode-1 or mode-2 vectors is not object of RD-optimization since this would lead to a general dependence within the encoding of the vectors x_m. However, in order to estimate the rate needed for the encoding of these positions in the current frame the corresponding rate from the preceding frame is taken.

The UV-component encoding is applied as described for the real-time codec.

4.1.4 Comparison of the Codecs

We now compare the real-time codec with the RD-optimized version. The PSNR course for several sequences at bitrate 28000 is shown in Fig. 4.9. At the beginning of the encoding, the PSNR curves start from a low level and after about 15 frames achieve a “stabilization” of the PSNR values. In the following, we call this initial area the *initialization phase*. Moreover, during the initialization phase, the RD-codec shows an increase in the PSNR values much faster than the real-time codec. Another observation can be made for the salesman sequence at the end of the initialization phase. After the fast improvement at the beginning, the RD-codec is outrun by the real-time codec for a short period. After this period, the performance RD-optimized coder again clearly does much better. Generally, the RD-optimized codec outperforms the real-time codec by more than 1 dB if the initialization phase is neglected.

In order to explain the faster increase of the RD-codec during the initialization phase, we provide statistics about the frequency of the selected modes for the RD-optimized (Fig. 4.7) and the real-time codec (Fig. 4.8). One can see that the real-time codec starts with mode-2 encoding. Since at the beginning the codebook is empty, i.e., it contains only zero vectors, it seems reasonable to fill the codebook with new vectors. But mode-2 encoding needs a large bitrate because of the side information that is necessary to transmit the whole vector. Therefore, there are only few blocks of the frame encoded until the bit budget is exhausted.

At the beginning of the encoding, the strategy of the RD-optimized codec is to use only mode-1 vectors. Since the codebook contains only zero vectors, the frame is filled with zero shape vectors and the means. Hence, the codec can encode much more blocks for a given bit budget because of the lower cost of mode-1 encoding. Thus, at first the RD-codec tries to find a fast approximation of the frame, and only then it starts filling the codebook. Therefore, the PSNR curve of the RD-codec increases faster in the initialization phase.

The different strategies applied during the initialization also explain the observed period at the end of the initialization phase of *Salesman*. At this point, the codebook of the real-time codec is already filled with useful vectors, but the RD-optimized codec still has an empty codebook. Thus, for a short period, the RD-codec has to invest a substantial number of bits to fill the codebook. During this time, the real-time codec shows a better performance since it can apply mode-1 encoding using vectors from previous frames.

One might argue that it is not surprising if an RD-optimized codec performs better than without RD-optimization. This is, however, not as obvious as it seems. First of all, the codecs differ not only in the method that determines the encoding modes but also in the usage of already encoded vectors. For example, in case of the real-time codec, vectors that are encoded in mode 2 are immediately inserted in the codebook and could be used again in the same frame for mode-1 encoding of subsequent vectors. This is on the other hand not possible for the RD-optimized codec. Our experiments, however, discovered that the real-time codec hardly uses mode-2 vectors in the same frame again.

Secondly, we know that inserted vectors can stay in the codebook for a long time and, thus, influence the efficiency of the coding of later frames. The RD-optimization, however, considers only one frame at a time and does not purposely insert vectors that lead to a coding gain for later frames. But the selection of the mode-2 vectors seems to have a similar "look-ahead" quality for both codecs.

The performance of the average PSNR value for several bitrates is shown in Fig. 4.10. The first 15 frames were not considered for the computation of the average in order to fade out the initialization phase.

4.1.5 Experiments with Variable Length Coding

Up to this point, we only used Huffman coding for variable length coding. Presently, we show how to improve the codecs by employing arithmetic coding. In the first experiment the cumulative frequencies needed for arithmetic coding

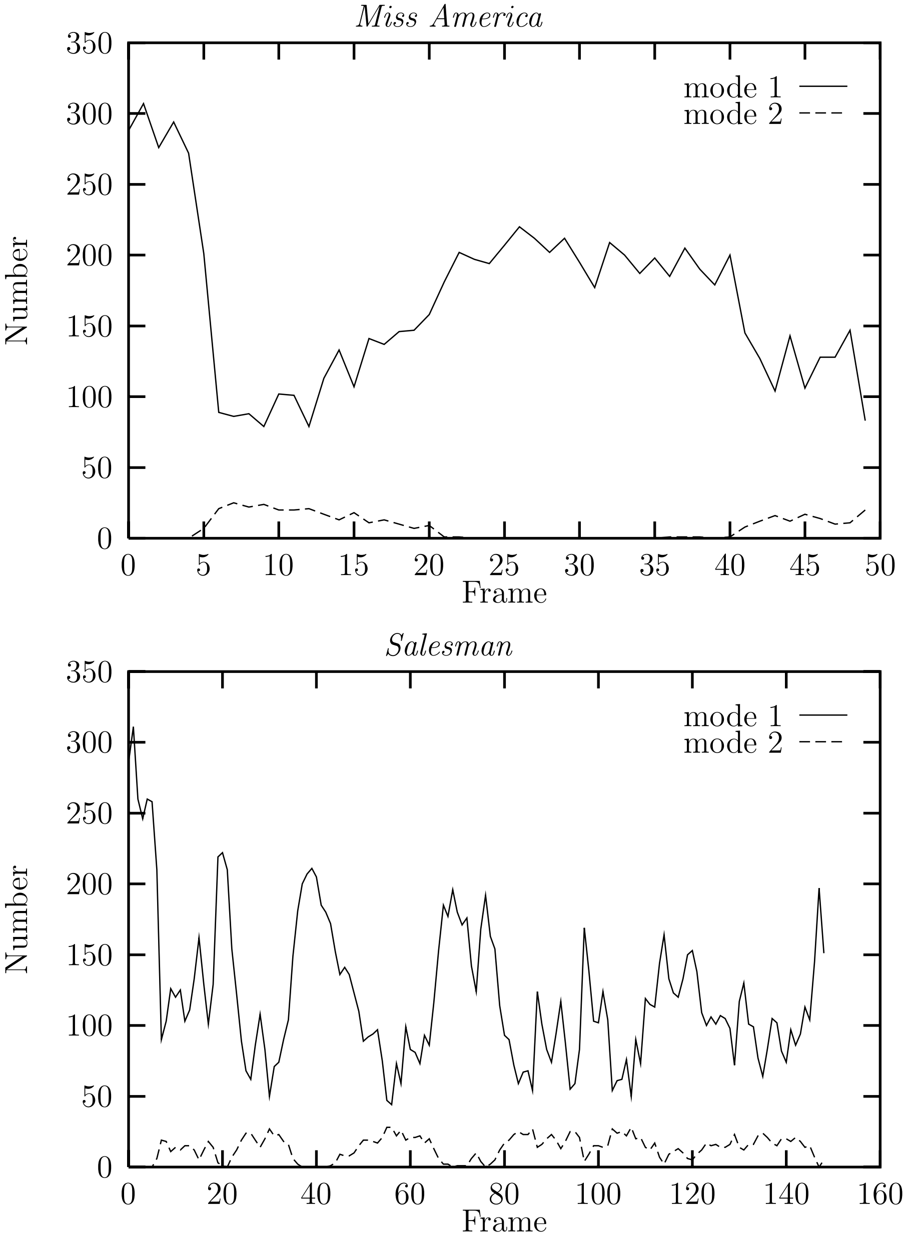


Figure 4.7: Number of vectors per frame encoded in mode 1 or mode 2 for the RD-optimized codec at 28000 bit/s. The corresponding figure for *Mother & Daughter* can be found in Fig. B.2 on Page 161.

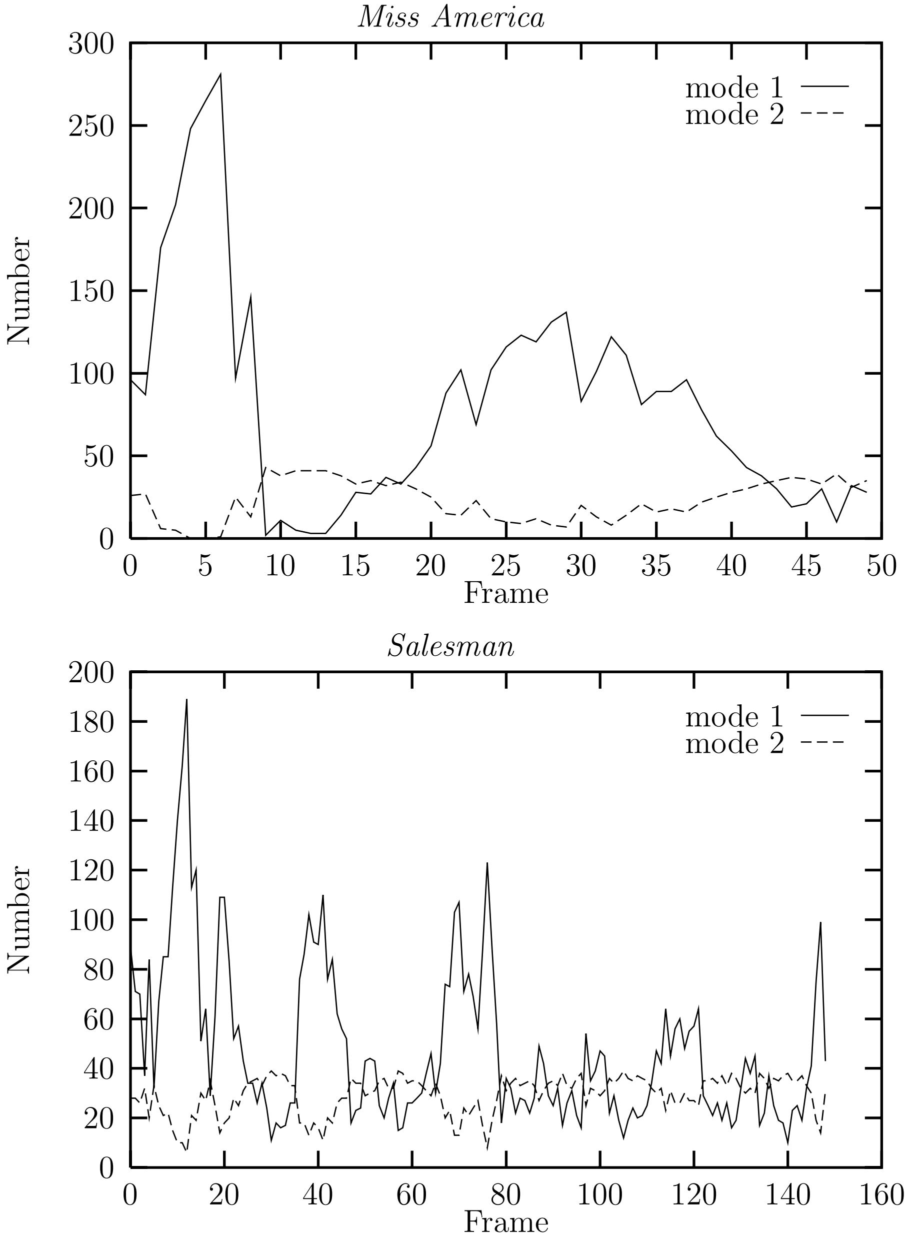


Figure 4.8: Number of vectors per frame encoded in mode 1 or mode 2 for the real-time-codec at 28000 bit/s. The corresponding figure for *Mother & Daughter* can be found in Fig. B.3 on Page 162.

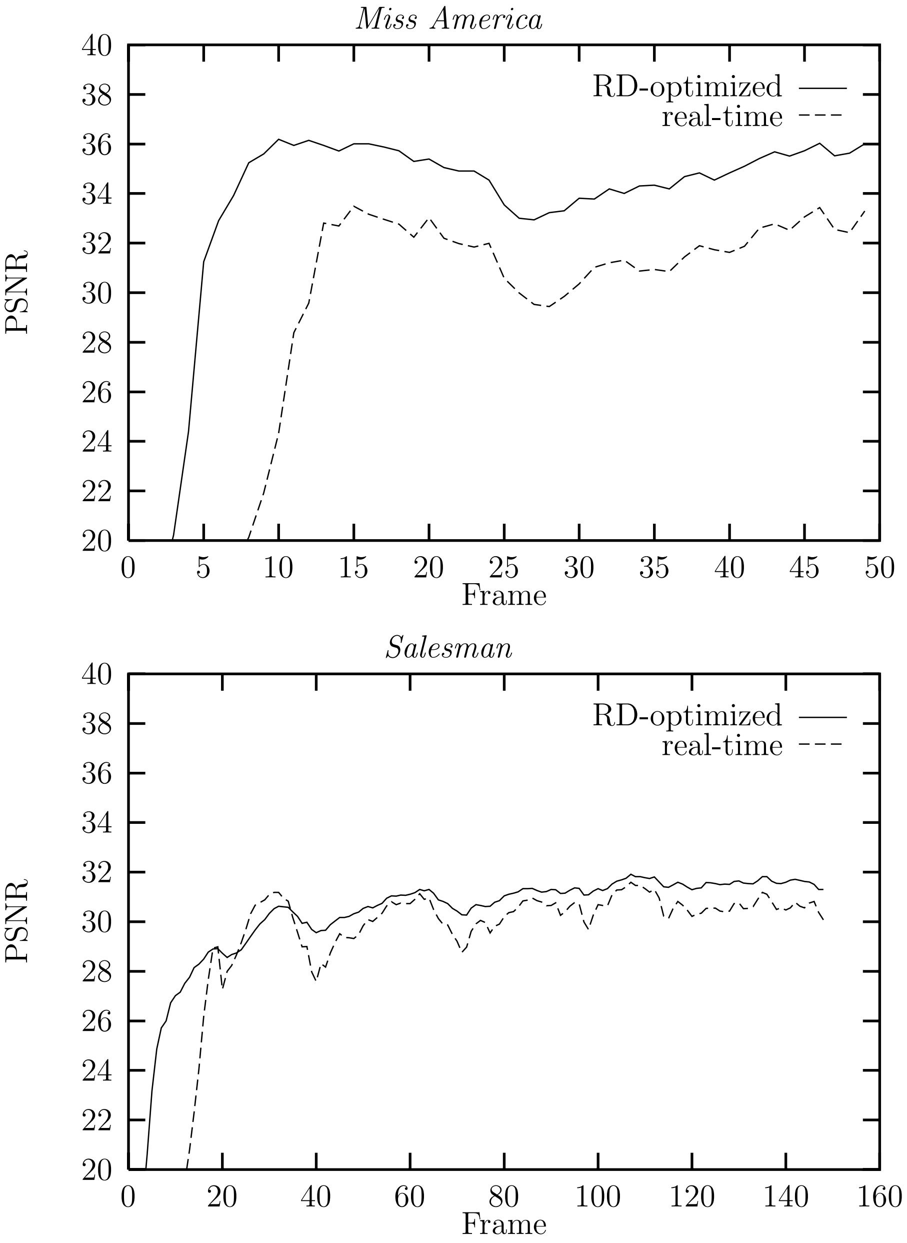


Figure 4.9: PSNR course for the real-time codec compared with its RD-optimization for the sequences *Miss America* and *Salesman* at bitrate 28000 bit/s. The corresponding course for *Mother & Daughter* can be found in Fig. B.1 on Page 161.

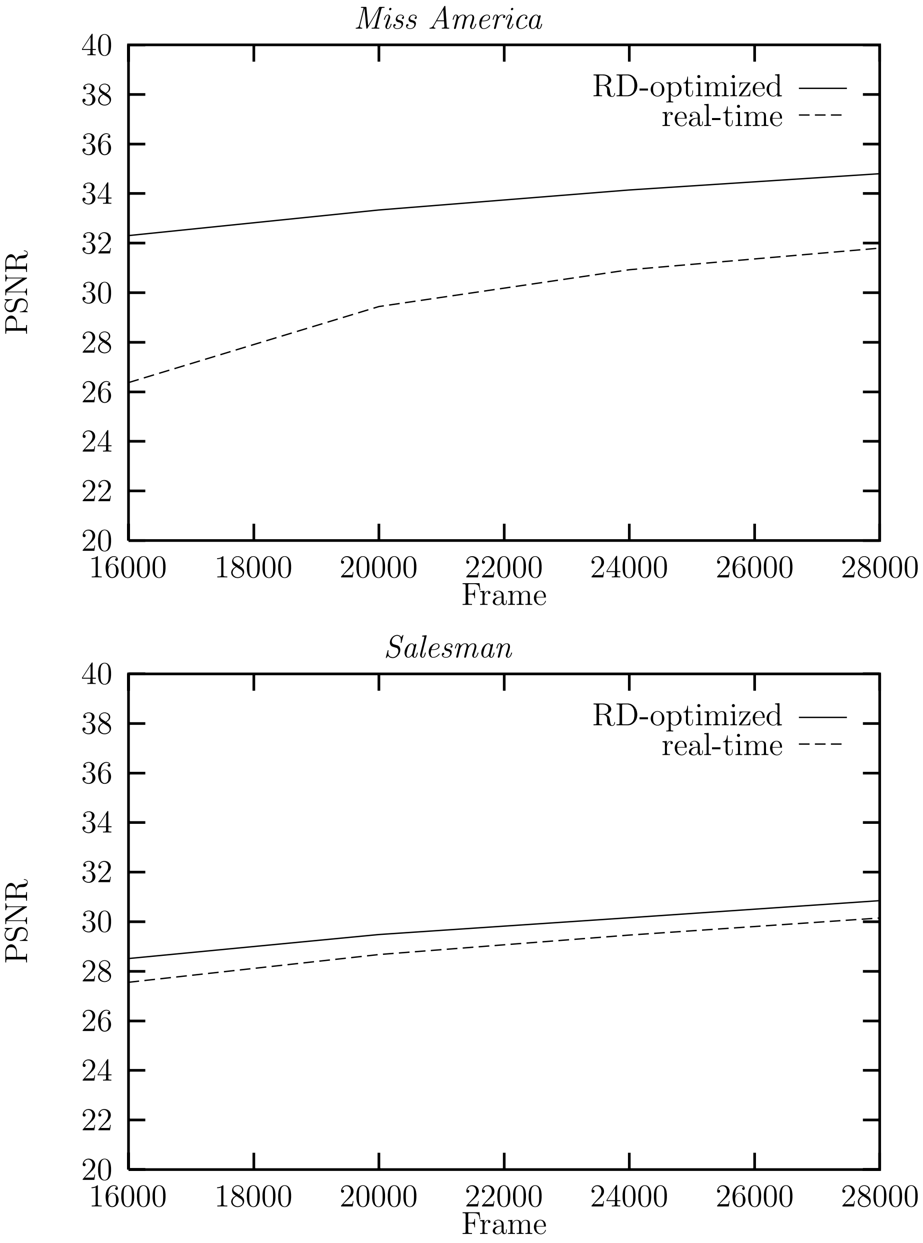


Figure 4.10: Average PSNR values for several bitrates for the real-time codec compared with its RD-optimization for the sequences *Miss America* and *Salesman* at 28000 bit/s. The corresponding figure for *Mother & Daughter* can be found in Fig. B.4 on Page 162.

of the symbols are derived from the symbol frequencies that have been used to create the Huffman code. In the second experiment, we use an adaptive arithmetic coder starting with equal probability for all symbols. The results are shown in Fig. 4.11 for the RD-codec and in B.5 for the real-time codec. Note that the adaptive arithmetic coder of the real-time codec adapts its frequencies for each symbol, but for the RD-codec the optimization constraints enforce an adaption that occurs only after the frame has been fully encoded; i.e., the symbol frequencies are kept separately during the encoding and are only added to the cumulative frequencies in one piece after the frame has been encoded.

The figures show that the Huffman coder is outperformed by the arithmetic coders. The best performing coder is the adaptive arithmetic coder.

From now on we use the adaptive arithmetic coder as default variable length coder[1] if not otherwise stated.

4.2 Wavelet Transform

Many of the image and most of the video compression algorithms make use of transform coding. Normally, a decorrelation transform is applied that concentrates the signal energy on a few components. This property is then exploited in subsequent quantization and VLC encoding (cf. Sect. 1.3). Theoretically, a vector quantizer is able to encode decorrelated vectors more efficiently than a scalar quantizer because of the so-called sphere packing property; moreover a vector quantizer is able to exploit higher order statistical dependences [28, 56]. Thus, transform coding with VQ theoretically outperforms scalar quantization schemes. But we expect much more than only exploiting higher order statistics from combining these schemes. Since we deal with an adaptive vector

[1] Note that, unlike Huffman coding, arithmetic coding does not assign a codeword to every symbol. Thus, we run into difficulties if we use arithmetic coding, since for RD-optimization we need the codeword lengths $|\gamma(a)|$ for the encoding of symbol a. In addition, the index coder γ does not satisfy its definition. However, for convenience' sake we continue to use this notation. The length $|\gamma(a)|$ is computed by $-\log_2 p(a)$ where $p(a)$ is the estimated probability of symbol a.

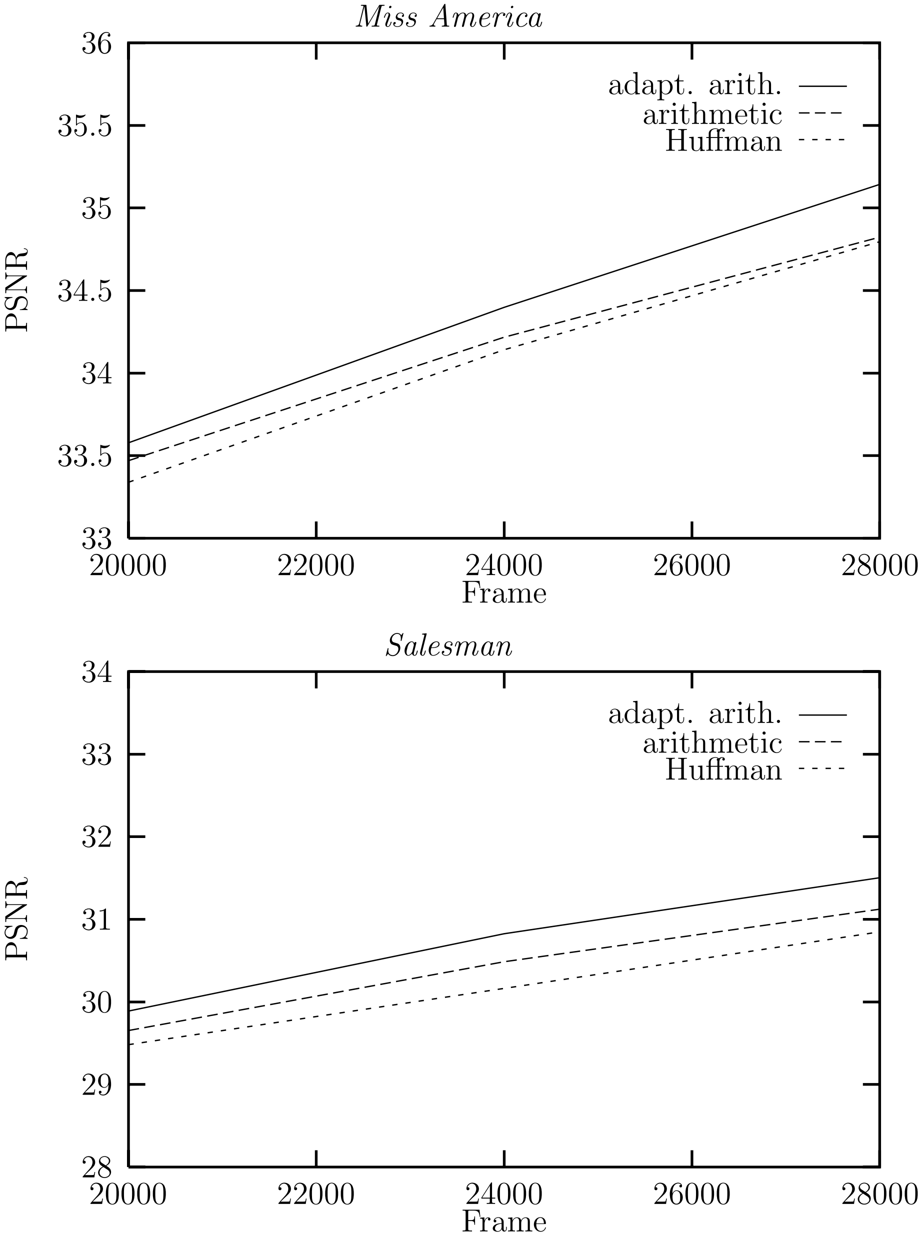
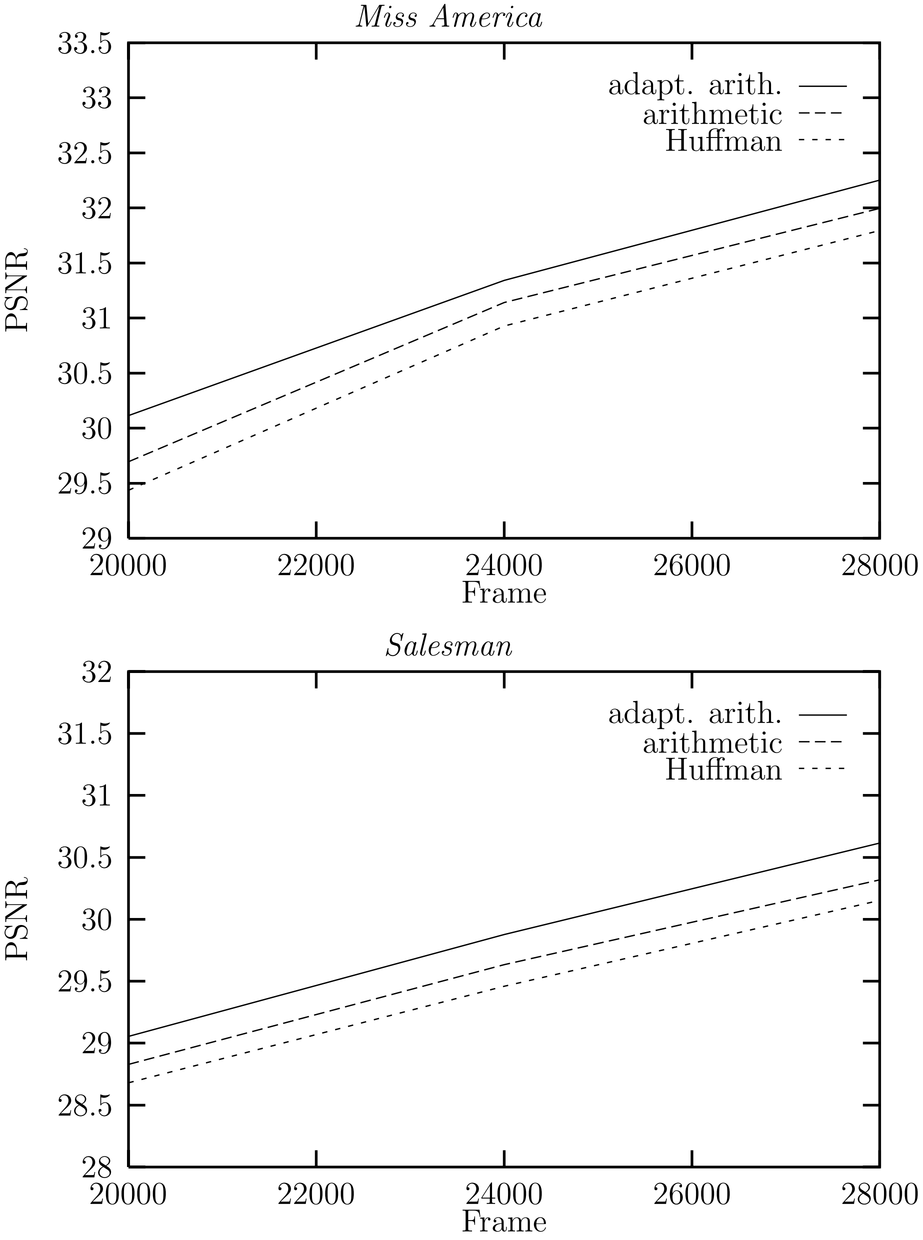


Figure 4.11: Comparison of different VLC coding methods for (a) *Miss America* and (b) *Salesman.* The corresponding figure for *Mother & Daughter* can be found in Fig. B.5 on Page 163.



Miss America

Salesman

Figure 4.12: Real-time codec. Comparison of different VLC coding methods for (a) *Miss America* and (b) *Salesman*. The corresponding figure for *Mother & Daughter* can be found in Fig. B.6 on Page 163.

quantizer, we have to transmit the codebook updates. In Sect. 4.1, this was done by DPCM encoding in the spatial domain. In the transform domain, the transmission of vectors works more efficiently.

The codec presented in this section is due to [37, 86].

4.2.1 Description of the Codec

Preprocessing. The preprocessing stage consists of a wavelet transform, grouping of transform coefficients, and subsequent scanning of these coefficients to get vectors for AVQ. The wavelet transform is done with a 2-times octave-band decomposition using 9/7-filters [2]. After the transform, the coefficients are grouped into blocks in a"zero-tree"-like fashion. The space localization property of the wavelet transform is used to create blocks corresponding to a certain region in the spatial domain. This is described in Fig. 4.13. Only highpass coefficients are organized as vectors. The lowpass coefficient is treated separately.

At the end of the preprocessing there are M 15-dimensional vectors x_0, $\dots$, x_{M-1} and lowpass coefficients $\mu_0, \dots, \mu_{M-1}$.

Encoding modes. It is assumed that the vectors and lowpass coefficients of a previously decoded transformed frame (reference frame), $x^r_0, \dots, x^r_{M-1}$ and $\mu_0, \dots, \mu_{M-1}$, are known both to the encoder and decoder. Then the encoder offers three encoding modes to approximate x_m by $\hat{x}_m$:

Mode 0. In this mode (replenishment mode), the content of the same position in the reference frame is restored, i.e., $\hat{x}_m \leftarrow x^r_m$ and $\hat{\mu}_m \leftarrow \mu^r_m$.

Mode 1. The lowpass coefficient μ_m is quantized, $\hat{\mu}_m \leftarrow Q_\mu(\mu_m)$; then x_m is represented by $\hat{x}_m \leftarrow Q(x_m)$.

Mode 2. The lowpass coefficient is quantized by Q_μ and the vector x_m is represented by a proper vector $c^*_m = Q^*(x_m) \in \mathcal{C}^*$, i.e., $\hat{x}_m \leftarrow c^*_m$ and $\hat{\mu}_m \leftarrow Q_\mu(\mu_m)$.

The resulting modes are collected in a modemap $\Omega = (\omega_0, \dots, \omega_{M-1}) : \omega_m \in \{0, 1, 2\}$. The position of the vectors encoded in mode-1 or mode-2 are transmitted by using run-length coding. For mode-1 and mode-2 encoding additional informations have to be sent. First, a flag indicating the mode is encoded. In both cases, an index for the scalar quantized lowpass coefficient $\gamma_\mu(\alpha_\mu(\mu_m))$ is sent. If x_m is encoded in mode 1, we also have to send an index of the quantized vector $\gamma(\alpha(x_m))$. For mode-2 encoding, the index $\gamma^*(\alpha^*(x_m))$ is transmitted.

Encoding parameters. All parameters that are not described explicitly in this paragraph are realized as in Sect. 4.1.3. The codebook size of the vector quantizer is $N = 512$ and the dimension of the vectors is $L = 15$. The index coder of the universal codebook works as follows. Each component of x_m is independently quantized by the scalar quantizer Q_s and separately encoded by γ_s. The binsizes of Q_s and Q_μ as well as the deadzones are 16. The codebook sizes are $N_s = 32$ and $N_\mu = 64$, respectively. These values have been experimental optimized in [37]. The index coders γ, γ^*, γ_μ, and γ_s are adaptive arithmetic coders starting with uniform cumulative frequencies.

RD-optimization For this codec, we can use the incremental computation of the convex hull method (see Alg. 6). The sets of RD-points $\mathcal{P}_m$ are created in an analogue way to Sect. 4.1.3.

The UV-component encoding is done as described in Sect. 4.1.

4.2.2 Results

In order to show the improvement by introducing the wavelet transform, we compare the current RD-optimized wavelet codec with the RD-optimized codec from Sect. 4.1.3. The PSNR course for a bitrate of 28000 bit/s is presented in Fig. 4.14. From this figure we see that the RD-optimized wavelet-based codec performs better than the RD-optimized basic codec by more than 1 dB PSNR. In order to analyze this performance gain, we compare the amount of bits spent for the different modes. This is shown in Tab. 4.1. Here, the average number

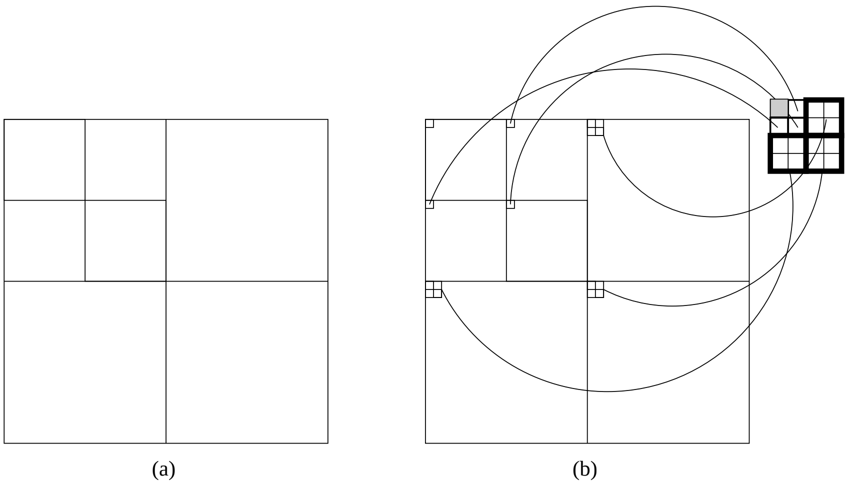


Figure 4.13: Organization of wavelet coefficients; (a) subband structure after transform, (b) grouping of blocks.

of bits per mode-1 and mode-2 encoded vector is computed over the whole sequence. As can be seen from the figure, the costs for a codebook update were reduced dramatically. The number of bits for mode-2 encoding is in the wavelet domain one-third of the mode-2 encoding in the spatial domain or less. Therefore, encoding of vectors in the transform domain is, as expected, more efficient than in the spatial domain. Moreover, the encoding cost for mode 1 is reduced by about 1 bit. Thus, we can conclude that the selected wavelet representation is suitable for AVQ.

The average PSNR values for different bitrates are provided in Fig. 4.15. As before, we excluded the first 15 frames from the average computation. We see from this figure that the above mentioned coding gain can be achieved for a wide range of bitrates.

4.3 Adaptive Partition with Quad-Trees

The codecs in the last sections used a mode map to store the different encoding modes for every vector. Then the mode map was scanned line by line and the positions of vectors encoded in mode 1 and mode 2 were transmitted with run-length coding. However, run-length coding of these positions leads to several

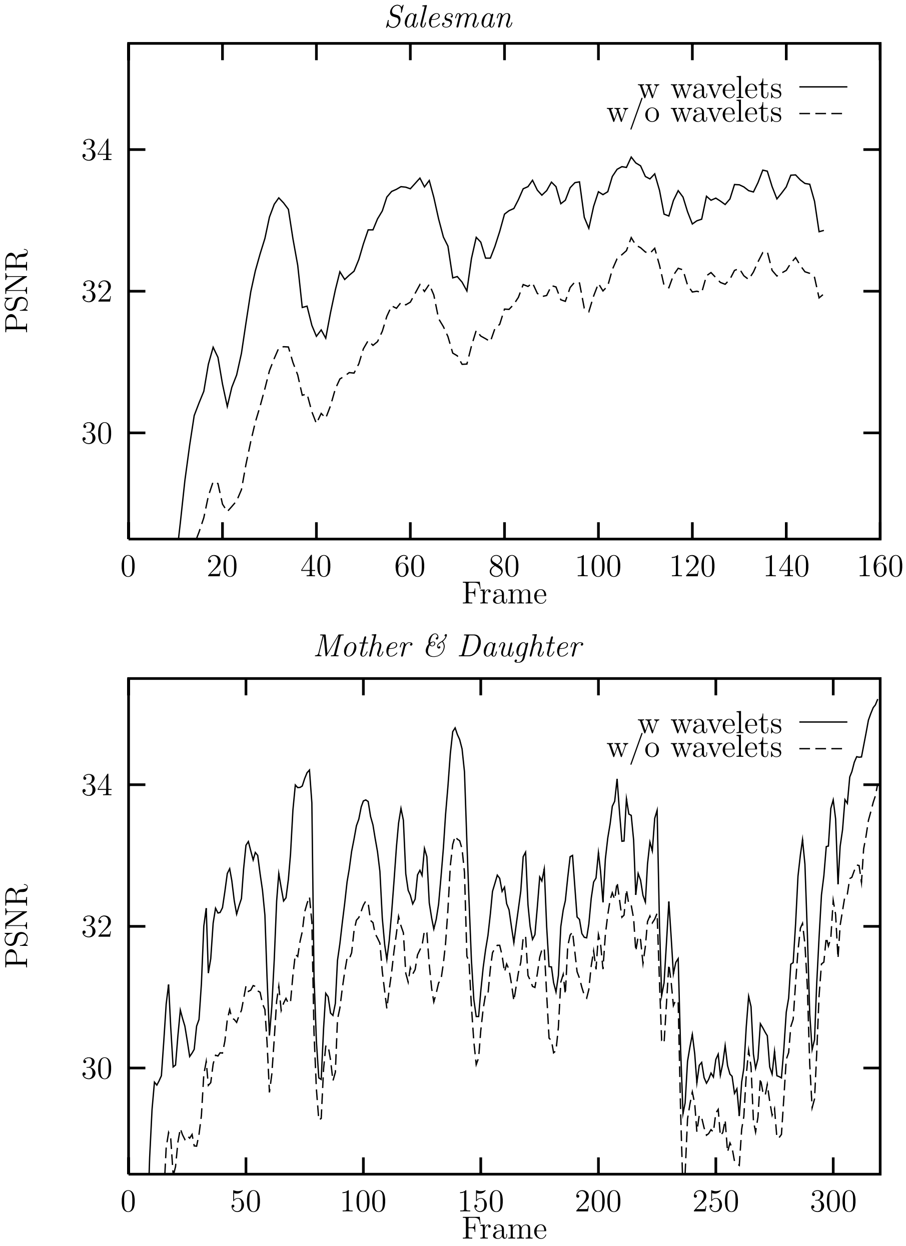


Figure 4.14: PSNR course at a bitrate of 28000 bit/s. The corresponding plot for *Miss America* can be found in Fig. B.8 on Page 164.

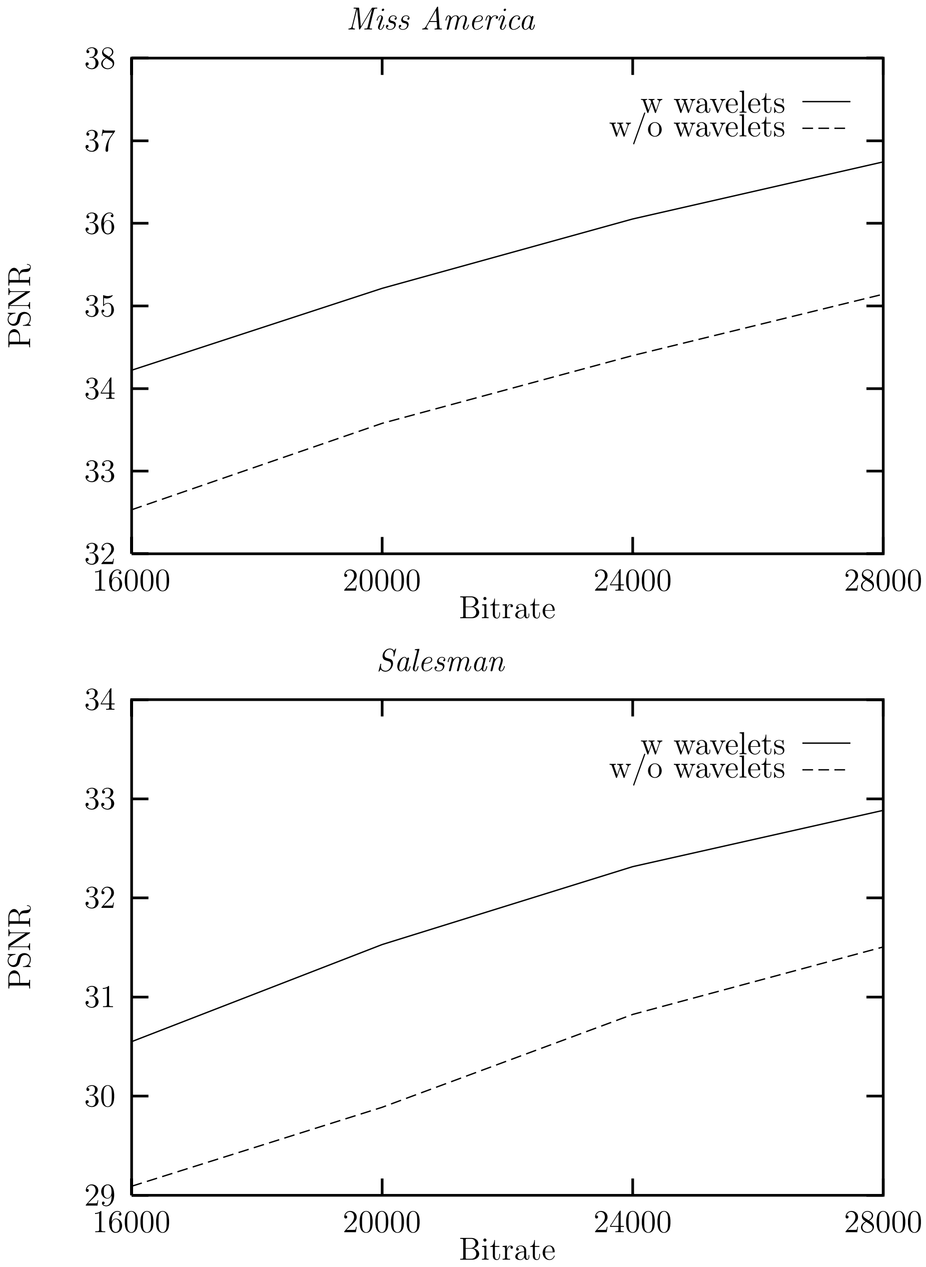


Figure 4.15: Mean PSNR of several bitrates for the wavelet codec. The corresponding figure for *Mother & Daughter* can be found in Fig. B.7 on Page B.7.

	Miss America		*Salesman*		*Mthr. & Dotr.*	
Bitrate	Mode 1	Mode 2	Mode 1	Mode 2	Mode 1	Mode 2
16000	2.6	52.7	4.6	60.7	3.5	51.4
28000	4.6	47.2	6.3	49.8	5.2	51.7

(a)

	Miss America		*Salesman*		*Mthr. & Dotr.*	
Bitrate	Mode 1	Mode 2	Mode 1	Mode 2	Mode 1	Mode 2
16000	1.8	14.2	4.0	15.0	2.3	14.2
28000	3.0	10.5	5.5	14.0	3.5	13.0

(b)

Table 4.1: Average cost of the two modes in bits per vector for the (a) RD-optimized codec and the (b) wavelet codec. The first 15 frames are excluded from the average computation.

problems:

1. The RD-optimization can not take into account the positions of the vectors since the encoding method of the positions would introduce dependences between the vectors that RD-optimization algorithms are not feasible to process.

2. Since each vector corresponding to a 4 × 4 area in the spatial domain might have a different mode, the encoding of modes for such small portions of the frame could be inefficient. Better results could be achieved by combining blocks that are spatially close to each other.

In order to illustrate the first point, we consider the costs of the bitmaps of the preceding codecs. Table 4.2 shows the cost for transmitting the positions of the blocks encoded in mode 1 or 2. The costs are computed by taking the average of the costs per frame for different image sequences and the preceding AVQ-codecs. As before, in order to exclude the initialization, the first 15 frames are not considered for the computation of the average.

It can be seen that the costs are increasing as the codec performance improves. This is due to the fact that better codecs encode more blocks and hence more positions have to be encoded. This makes the run-length coding less efficient. The same argument holds true for smaller bitrates: The costs decrease with the bitrate since a smaller number of blocks must be encoded for smaller bitrates.

Table 4.2 shows that all codecs spend a reasonable amount of bits on the encoding of the positions of mode-1 or mode-2 vectors. For example, the wavelet codec demands for *Miss America* and *Mother & Daughter* at a bitrate of 28000 bit/s on the average 500 bit per frame and more. However, RD-optimization algorithms can not take these costs into account. Thus, the costs have to be estimated using the costs from the preceding frame. If we use for the positions a different encoding scheme that can be handled by RD-optimization, a coding gain might be achieved since the costs for the positions of different blocks can be an object of minimization.

The second point, mentioned in the enumeration above, is demonstrated in Fig. 4.16. This figure shows a typical mode map during the encoding of *Miss America*. One can see that the blocks that are encoded in mode 1 or mode 2 are concentrated in few small areas of the frame. We might expect to achieve a reasonable coding improvement if we offer the additional option to represent four blocks corresponding to a 4×4 pixel area by one vector corresponding to a 8×8 pixel area. In addition, we can expect to save bits for the encoding of the positions if we are able to merge four positions of small blocks by one position of larger blocks.

Such an adaptive, hierarchical organization of a frame can be managed by quad-trees (cf. Sect. 1.4).

There are a lot of conceivable organizations of different levels in the wavelet transform domain. But, in order to remain consistent with the preceding sections we organize this section as follows. First, we describe the codec for one example of quad-tree organizations. Then, in Sect. 4.3.2, we describe experiments with different quad-trees.

4.3.1 Description of the Codec

We describe the codec-structure published in [87, 88].

codec type	*Miss America*	*Salesman*	*Mother & Daughter*
real-time	307	246	295
RD-optimized	516	428	538
with wavelets	542	442	572

(a)

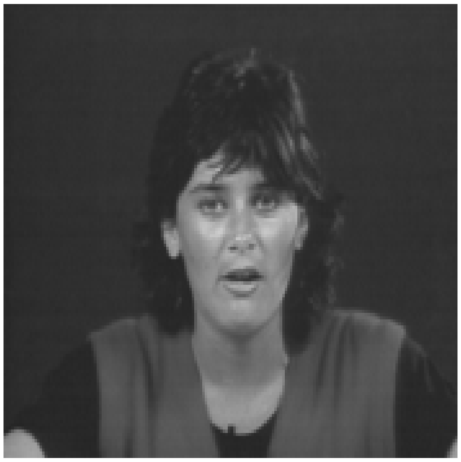
codec type	*Miss America*	*Salesman*	*Mother & Daughter*
real-time	400	328	427
RD-optimized	660	510	712
with wavelets	704	549	754

(b)

Table 4.2: Average number of bit per frame needed to encode positions of mode-1 or mode-2 vectors (a) at a bitrate of 16000 and (b) at a bitrate of 28000.

```
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000001000221000000000000000000
00000000000000000010002111000000000000000000
00000000000000000100100211021100000000000000
00000000000000000000000000022100000000000000
00000000000000000000000000021100000000000000
00000000000000000000000000021200000000000000
00000000000000000000000000021200000000000000
00000000000000000000011101111110000000000000
00000000000000000021222110211000000000000000
00000000000000000021211210000100000000000000
00000000000000000122210022100010000000000000
00000000000000002112111021211011000000000000
00000000000000001102201122222211000000000000
00000000000000021110201001102200000000000000
00000000000000022110011021102110000000000000
00000000000000022010012112111100000000000000
00000000000000022100000001022000000000000000
00000000000000001110022220001001000000000000
00000000000000000010212220010000000000000000
00000000000000000000002100000000000000000000
00000000000000000000100100010000000000000000
00000000000000000000021212010000000000200000
00000000000000011020010100010000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000120000000000000000
00000000000000000000000010000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
```

(a)



(b)

Figure 4.16: (a) Mode map of (b)

Preprocessing. As in Sect. 4.2, we start with the wavelet transform. The coefficients in the wavelet domain then are regrouped into macroblocks corresponding to 16×16 pixel areas in the spatial domain. The organization of the wavelet coefficients of a macroblock is shown in Fig. 4.17. Each macroblock consists of 16 pixels from each of the subbands 1,2,3,4 and 64 pixels from each of the subbands 5,6,7, resulting in a total of 256 coefficients.

For every macroblock, there are three possible decomposition levels ranging from coarse to fine. Level-0 encoding describes the whole macroblock with all 256 wavelet coefficients. Alternatively, the level-0 block can be decomposed into four level-1 blocks corresponding to a spatial 8×8-pixel area, containing four pixels from each of the 1,2,3,4 subbands and 16 pixels from each of the 5,6,7 subbands. Moreover, one can decompose every level-1 block into four level-2 blocks each corresponding to 4×4-pixel blocks in the spatial domain containing one pixel from each of the 1,2,3,4 subbands and four pixels from each of the 5,6,7 subbands. The grouping of the wavelet coefficients for level-1 and level-2 blocks within a macroblock is depicted in Fig. 4.18.

In order to create vectors from blocks, the blocks are scanned line by line, excluding coefficients from several subbands depending on the block level. For level-0 vectors, we take all coefficients of the level-0 blocks, that is, level-0 vectors are 256-dimensional. For level-1 vectors, only the coefficients from subbands 2,3 and 4 of level-1 blocks are taken. The coefficients of higher subbands are set to zero, whereas the four coefficients of subband 1 are treated separately. Thus, the dimension of level-1 vectors is 12. For level-2 vectors all coefficients of level-2 blocks are taken except the coefficient of subband 1 which is, similar to level-1 vectors, treated separately. The dimension of level-2 vectors is 15.

After the preprocessing, several vectors of several levels can be assigned to the M macroblocks. If we consider the mth macroblock, $0 \leq m < M$, then this macroblock contains one level-0 vector, x_m^0, 4 level-1 vectors, $x_{4m}^1, \ldots, x_{4m+3}^1$, and 16 level-2 vectors, $x_{16m}^2, \ldots, x_{16m+15}^2$. In addition, there are 4 vectors with level-1 means, $\mu_{4m}^1, \ldots, \mu_{4m+3}^1$, from subband 1 and 16 scalar level-2 means, $\mu_{16m}^2, \ldots, \mu_{16m+15}^2$. Analogously, the macroblock m of the reference frame is given by $\tilde{x}^0_m$, $\tilde{x}^1_{4m}$, $\ldots$, $\tilde{x}^1_{4m+3}$, $\tilde{x}^2_{16m}$, $\ldots$, $\tilde{x}^2_{16m+15}$ and μ^1_{4m}, $\ldots$, μ^1_{4m+3}, μ^2_{16m}, $\ldots$, μ^2_{16m+15}.

Encoding modes. For level-0 vectors, there exists only one encoding mode. Replenishment is applied, i.e., the content of the same position in the reference frame is restored. For level-1 and level-2 vectors , the AVQ-approach is used. There are three encoding modes:

Mode 0. Replenishment mode. This is identical with the replenishment mode for level-0 vectors, i.e., the content of the same position of the reference frame is restored.

Mode 1. VQ-mode. First, a flag is sent indicating mode 1. The vector x_m is then processed according to its level.

Level-1. The level-1 vector x^1_m is represented by $Q_1(x^1_m)$. Then the mean $\overline{\mu}^1_m$ of the four lowpass coefficients μ^1_m is computed and represented by $Q_\mu(\overline{\mu}^1_m)\mathbf{1}_4$. Thus, $\gamma_1(\alpha_1(x^1_m))$ and $\gamma_\mu(\alpha_\mu(\overline{\mu}^1_m))$ are transmitted.

Level-2. The level-2 vector x^2_m is represented by $Q_2(x^2_m)$, and the lowpass coefficient μ^2_m is represented by $Q_\mu(\mu^2_m)$. Thus, $\gamma_2(\alpha_2(x^2_m))$ and $\gamma_\mu(\alpha_\mu(\mu^2_m))$ are transmitted.

Mode 2. Update mode. First, a flag is sent indicating mode 2. Then the vector x_m is processed according to its level.

Level-1. The level-1 vector x^1_m is represented by $Q^*_1(x^1_m)$. Then the mean $\overline{\mu}^1_m$ of the four lowpass coefficients μ^1_m is represented by $Q_\mu(\overline{\mu}^1_m)\mathbf{1}_4$. Thus, $\gamma^*_1(\alpha^*_1(x^1_m))$ and $\gamma_\mu(\alpha_\mu(\overline{\mu}^1_m))$ are transmitted.

Level-2. The level-2 vector x^2_m is represented by $Q^*_2(x^2_m)$ and the lowpass coefficient μ^2_m is represented by $Q_\mu(\mu^2_m)$. Thus, $\gamma^*_2(\alpha^*_2(x^2_m))$ and $\gamma_\mu(\alpha_\mu(\mu^2_m))$ are transmitted.

The resulting positions of vectors that are encoded in mode 1 or mode 2 are managed by a quad-tree structure.

Encoding parameters. Since we have different vector dimensions, we need two vector quantizers, namely one vector quantizer for level-1 vectors ($N_1, \mathcal{C}_1$, $L_1, \mathcal{I}_1$, $\mathcal{V}_1, \alpha_1$, β_1, γ_1, $\mathcal{C}_1^*, \alpha_1^*$, β_1^*, γ_1^*) and one for level-2 vectors ($N_2, \mathcal{C}_2$, $L_2, \mathcal{I}_2$, $\mathcal{V}_2, \alpha_2$, β_2, γ_2, $\mathcal{C}_2^*, \alpha_2^*$, β_2^*, γ_2^*). The vector dimensions are $L_1 = 12$ and $L_2 = 15$. The codebook sizes are $N_1 = 64$ and $N_2 = 512$. The vector encoders α_1 and α_2 are realized by $\alpha_1(\lambda, x) = \arg\min_{i\in\mathcal{I}}[d(x, \beta_1(i)) + \lambda \cdot \gamma_1(i)]$ and $\alpha_2(\lambda, x) = \arg\min_{i\in\mathcal{I}}[d(x, \beta_2(i)) + \lambda \cdot \gamma_2(i)]$. The codebooks, $\mathcal{C}_1$ and $\mathcal{C}_2$, and the vector decoders, β_1 and β_2, are organized as described in Sect. 4.1.1, i.e., each vector in the codebooks has a frequency count of its own and each vector decoder, β_1 and β_2, satisfies (4.1).

The universal quantizers, Q_1^* and Q_2^*, and index coders, γ_1^* and γ_2^*, work as follows. The coefficients of x are independently scalar quantized and encoded by Q_s and γ_s, respectively. Q_f has binsize 16, deadzone 32 and $N_f = 32$.

Q_μ has binsize 16, deadzone 32, and $N_\mu = 64$.

The index coders γ_1, γ_2, γ_s and γ_μ are realized by adaptive arithmetic coders with an initially uniform cumulative frequency.

RD-optimization. Since the different levels are managed by quad-trees, we can use the incremental computation of the convex hull for hierarchical dependences (see Sect. 3.3.3).

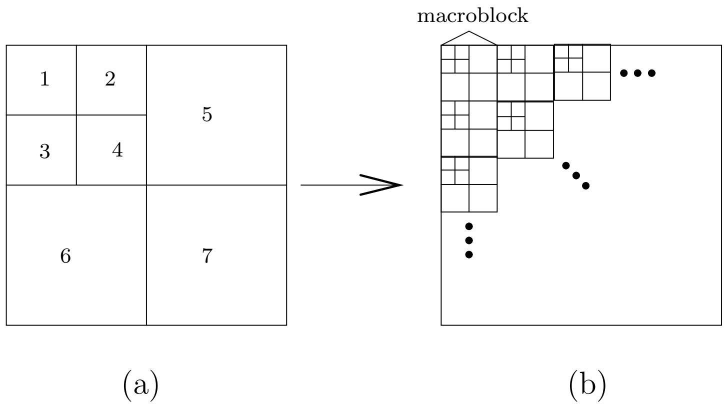
We assign a tree $\mathcal{T}_m = \{t_0, \ldots, t_{20}\}$ to the macroblock m as shown in Fig. 4.19. Then, $\{t_0\}$, $\{t_1, \ldots, t_4\}$ and $\{t_5, \ldots, t_{20}\}$ describe level 0, level 1 and level 2 of the macroblock. We define for each node t_n, $0 \leq n < 21$, a set of RD-points, $\mathcal{T}_{t_n}$. In the following we set $d(x, y) = \|x - y\|_2^2$.

Level 0. $\mathcal{P}_{t_0} = \{(R_{t_0}^0 = 0, D_{t_0}^0 = d(x_m^0, \tilde{x}_m^0))\}$

Level 1. The sets $\mathcal{P}_{t_{n+1}}$, $0 \leq n < 4$, contain one point for mode-0 encoding, N_1 points for mode-1 encoding, and one point for mode-2 encoding.

Mode 0 $(R_{t_{n+1}}^0 = 0, D_{t_{n+1}}^0 = d(x_{4m+n}^1, \tilde{x}_{4m+n}^1) + d(\mu_{4m+n}^1, \tilde{\mu}_{4m+n}^1))$,

Mode 1 $(R_{t_{n+1}}^{i+1} = |\gamma_1(i)| + |\gamma_\mu(\alpha_\mu(\overline{\mu}_{4m+n}^1))|$, $D_{t_{n+1}}^{i+1} = d(x_{4m+n}^1, \beta_1(i)) + d(\mu_{4m+n}^1, Q_\mu(\overline{\mu}_{4m+n}^1)\mathbf{1}_4))$, $0 \leq i < N_1$



(a)

(b)

Figure 4.17: Regrouping of macroblocks in the wavelet domain. (a) Subband grouping. Coefficients are grouped corresponding to their type of subband membership. (b) Macroblock grouping. Coefficients are grouped corresponding to their spatial position.

Mode 2 $(R^{N_1+1}_{t_{n+1}} = |\gamma^*_1(\alpha^*_1(x_{4m+n}))| + |\gamma_\mu(\alpha_\mu(\overline{\mu}^1_{4m+n}))|,\ D^{N_1+1}_{t_{n+1}} = d(x^1_{4m+n},\ Q^*_1(x^1_{4m+n})) + d(\mu^1_{4m+n},\ Q_\mu(\overline{\mu}^1_{4m+n})\mathbf{1}_4))$

Level 2. The sets $\mathcal{P}_{t_{n+5}}$, $0 \leq n < 16$, contain one point for mode-0 encoding,N_2 points for mode-1 encoding, and one point for mode-2 encoding.

Mode 0 $(R^0_{t_{n+5}} = 0, D^0_{t_{n+5}} = d(x^2_{16m+n}, \tilde{x}^2_{16m+n}) + d(\mu^2_{16m+n}, \tilde{\mu}^2_{16m+n}))$

Mode 1 $(R^{i+1}_{t_{n+5}} = |\gamma_2(i)| + |\gamma_\mu(\alpha_\mu(\mu^2_{16m+n}))|, D^{i+1}_{t_{n+5}} = d(x^2_{16m+n}, \beta_2(i)) + d(\mu^2_{16m+n}, Q_\mu(\mu^2_{16m+n})))$, $0 \leq i < N_2$

Mode 2 $(R^{N_2+1}_{t_{n+5}} = |\gamma^*_2(\alpha^*_2(x^2_{16m+n}))| + |\gamma_\mu(\alpha_\mu(\mu^2_{16m+n}))|,\ D^{N_2+1}_{t_{n+5}} = d(x^2_{16m+n},\ Q^*_2(x^2_{16m+n})) + d(\mu^2_{16m+n},\ Q_\mu(\mu^2_{16m+n})))$

The UV-component encoding is done as described in Sect. 4.1.

4.3.2 Experiments with Quad-Tree Organizations

The quad-tree based codec in Sect. 4.3.1 has been developed by evaluating various quad-tree organizations. In this section, we describe these experiments. All experiments are governed by the following basic structure. The coefficients are organized in macroblocks (see Fig. 4.17). For the coding of the macroblocks

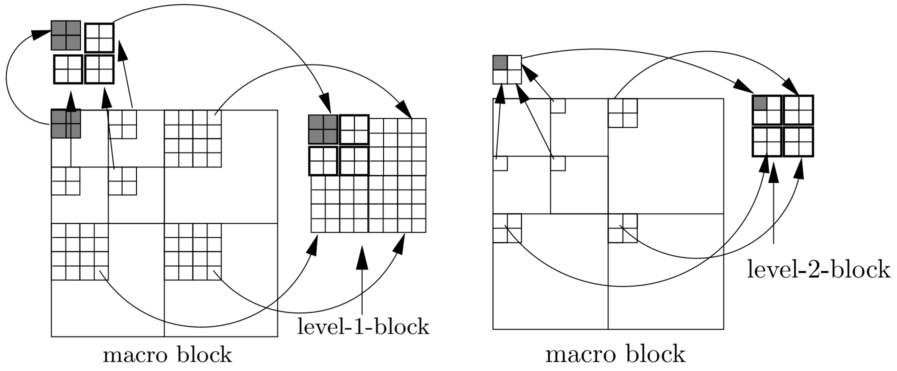


Figure 4.18: Grouping of macroblock coefficients for different block levels.

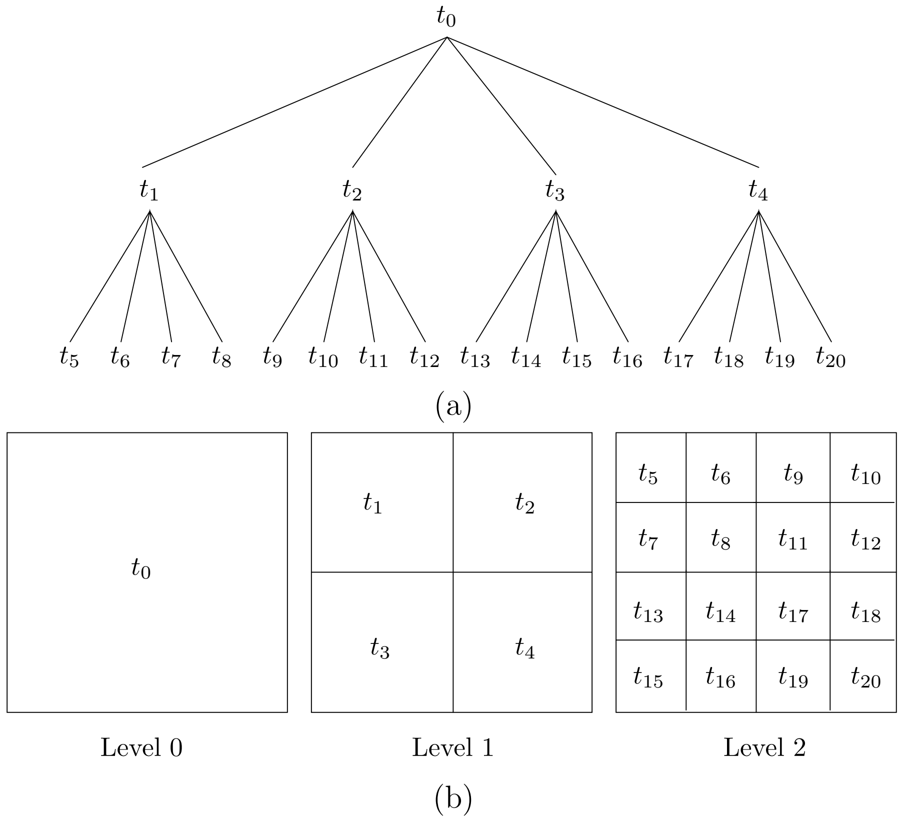

t_0
t_1
t_2
t_3
t_4
t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{15} t_{16} t_{17} t_{18} t_{19} t_{20}
(a)
t_0
t_1 t_2
t_3 t_4
t_5 t_6 t_9 t_{10}
t_7 t_8 t_{11} t_{12}
t_{13} t_{14} t_{17} t_{18}
t_{15} t_{16} t_{19} t_{20}
Level 0
Level 1
Level 2
(b)

Figure 4.19: Tree $\mathcal{T}_m = \{t_0, \ldots, t_{20}\}$(a) corresponding to macroblock m (b).

we have three possible encoding levels: Level-0 blocks describe the whole macroblock corresponding to a 16 × 16-pixel area in the spatial domain. Level-0 blocks can be decomposed into four level-1 blocks, each corresponding to a 8 × 8 pixel area in the spatial domain. Analogously, each level-1 block can be subdivided into four level-2 blocks corresponding to a 4 × 4 pixel spatial area. The block structure is depicted in Fig. 4.18. In the following we enumerate the quad-trees (QT) structures by describing the encoding options provided for every level. The vectors and lowpass coefficients of the different levels are denoted as in Sect. 4.3.1.

QT 1. For level 0, there is only a replenishment mode. For level 1, first the mean of the vector of lowpass coefficient μ_m^1 is computed, $\overline{\mu}_m^1$, and represented by $Q_\mu(\overline{\mu}_m^1)$. The vectors x_m^1 are encoded either in mode 1 or mode 2, i.e., x_m^1 can be represented by $Q_1(x_m^1)$ or $Q_1^*(x_m^1)$ and is encoded by γ_1 or γ_1^*, respectively. For level 2, the lowpass coefficient μ_m^2 is quantized, $Q_\mu(\mu_m^2)$, and represented by the quantized value. The vectors x_m^2 are encoded either with mode 1 or mode 2, i.e., x_m^2 can be represented by $Q_2(x_m^2)$ or $Q_2^*(x_m^2)$ and is encoded by γ_2 or γ_2^*.

QT 2. Same as QT 1, but with an additional replenishment mode (mode 0) for level-1 blocks.

QT 3. Same as QT 2, but with an additional replenishment mode (mode 0) for level-2 blocks.

QT 4. Same as QT 3, but the encoding of μ_m^1 is different. The vector of lowpass coefficients μ_m^1 is quantized with DPCM, using the corresponding scalar quantizer Q_d, and encoded with γ_d.

QT 5. Same as QT 3, but the encoding of μ_m^1 and x_m^1 is different. In addition to the encoding of the mean $\overline{\mu}_m^1$, the shape vector $\mu_m^1 - \overline{\mu}_m^1 \mathbf{1}_4$ is appended to the level-1 vector. Thus, the resulting vector $\hat{x}_m^1$ is 16-dimensional. For mode-1 and mode-2 encoding, $Q_3(\hat{x}_m^1)$ and $Q_3^*(\hat{x}_m^1)$ are used together with the index coders γ_3 and γ_3^*, respectively.

QT 6. Same as QT 5, but the shape vector $\mu_m^1 - \overline{\mu}_m^1 \mathbf{1}_4$ is vector quantized separately. Thus, the shape vector and x_m^1 are encoded with product code VQ or, to be more specific, with partitioned VQ (see Sect. 2.3.3). The shape vector is quantized by Q_4 and Q_4^* and is encoded by γ_4 and γ_4^* for mode-1 and mode-2 coding of level-1 blocks.

The parameters of the experiments described above are as follows. The vector quantizers Q_1 and Q_2 are described in Sect. 4.3. The scalar quantizer Q_d has binsize 16 and deadzone 32 and $N_d = 64$. The quantizers Q_3 and Q_4 are organized analogously to Q_1 and Q_2 with different codebook sizes and vector dimensions, namely, $L_3 = 16$, $L_4 = 4$, $N_3 = 256$, and $N_4 = 128$. As before, all index coders, γ, are realized by an adaptive arithmetic coder with an initially uniform cumulative frequency.

Presently, we come to the results of the experiments. Figure 4.20 and 4.21 show the PSNR courses of the experiments compared with the PSNR course of the wavelet codec for bitrates 16000 and 28000 bit/s.

We observe that all quad-tree codecs finish the initialization phase much faster than the wavelet codec. This is due to the fact that the quad-tree codec is able to encode a mean for a block corresponding to a 8×8-pixel area. Thus, the rough approximation of the frame is faster than for the wavelet codec.

Now we consider QT 1. After the initialization phase, the PSNR courses of the two test sequences behave differently. While for *Miss America* the codec with quad-trees maintains good performance, the quad-tree encoding of *Salesman* performs worse than the wavelet codec. For QT 2 we observe the same. For QT 3 the quad-tree codec for both sequences is better than the wavelet codec. This upward trend for *Salesman* is preserved in QT 4, that is, the gap between the quad-tree codec and the wavelet codec is getting wider. On the other hand, this gap becomes smaller for the encoding of *Miss America*. The performance of the subsequent experiment, QT 5, is similar to QT 3. The same holds true for QT 6.

First, we have to consider the question why QT 1 performs that badly. Apparently, the performance depends on the test sequence. For *Miss America* it turns out to be better for *Salesman* worse than the wavelet codec. But if we take a look at the implicit assumption of QT 1, it becomes clearer why the codec fails for *Salesman*. The assumption is that replenishment is needed only for level-0 blocks. This means, however, that if a block is not encoded with replenishment then it must be completely represented with mode-1 and mode-2 encoding for level 1 and level 2. This encoding strategy is efficient for sequences whose image blocks encoded in mode 1 and mode 2 are locally concentrated and do not spread over the whole frame. This holds true for *Miss America* (as one can see, for example, in Fig. 4.16), but not always for *Salesman*. Thus, replenishment is needed for level-1 and level-2 encoding. This hypothesis is verified by QT 2 and 3. Even though the encoding costs increase by introducing additional replenishment modes, the performance of the salesman sequence noticeably improves. For QT 3 the coding performance is even better than the wavelet codec. On the other hand, the performance of *Miss America* does not change significantly.

Now let us consider level-1 encoding for QT 3. It seems that this scheme is not well-balanced. On the one hand, a lot of bits are spent to encode some bandpass coefficients with VQ (see Fig. 4.18). On the other hand, the lowpass coefficients are roughly approximated by only one mean. This aspect is evaluated by QTs 4–6. Here, we can observe what happens, if we spend more bits on the encoding of the 4 lowpass coefficients. However, in all experiments we trade a performance increase in the case of *Salesman* for a decrease in the case of *Miss America*.

The average PSNR values for QTs 3–6 are presented in Fig. 4.22. The figure shows that QT 3,5,6 perform very similar. The results of QT 4 depend on the test sequence. For *Miss America*, this QT performs worse than QT 3,5,6. The opposite holds true for *Salesman*; here, QT 4 is better than the other QTs.

We conclude that QT 3, 5, and 6 perform best over all test sequences. Thus, for further comparisons with other codecs, we select QT 3 since this has the lowest complexity.

Finally, we would like to continue the discussion from the beginning of Sect. 4.3. We asserted then that the use of a quad-tree structure could, among

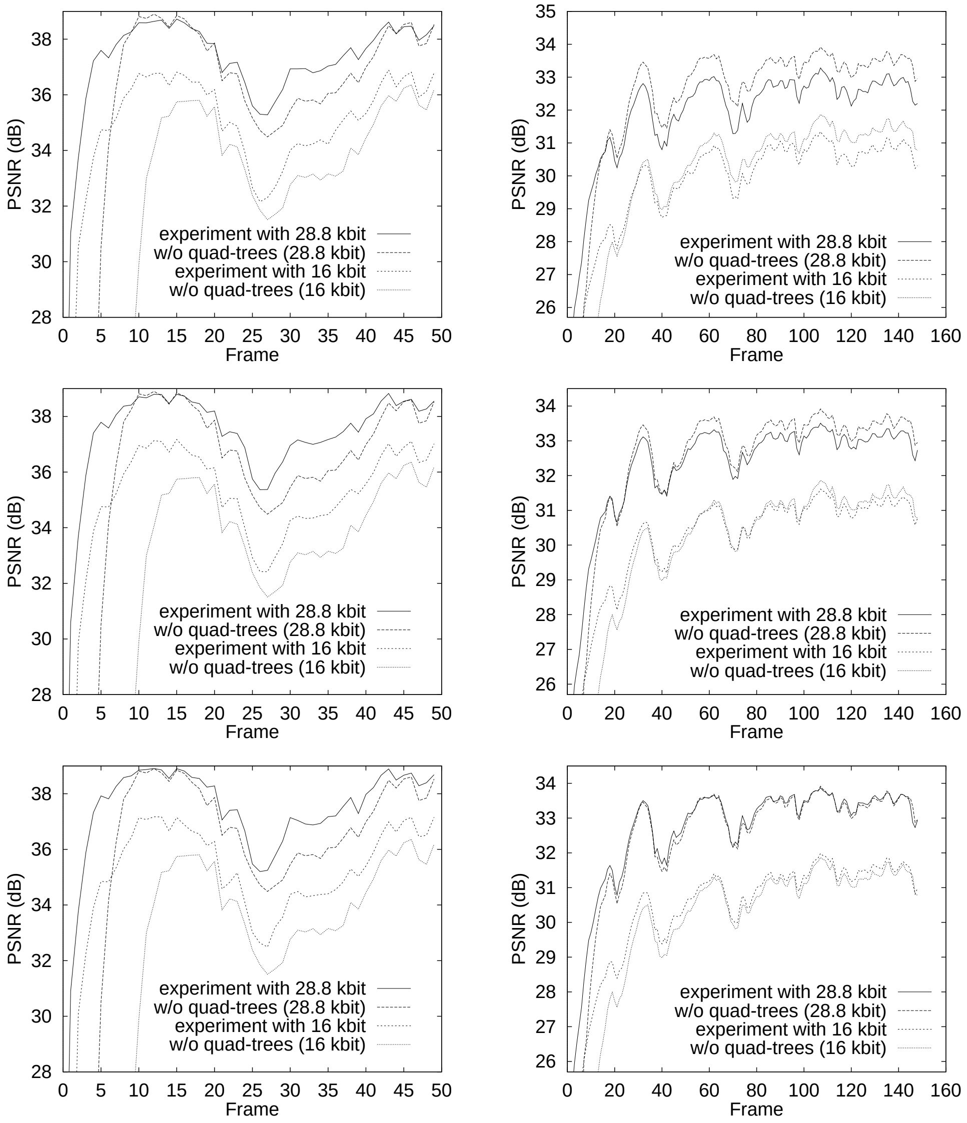


Figure 4.20: PSNR coursed of (row by row) QT 1, 2 and 3 for (column by column) *Miss America* and *Salesman*.

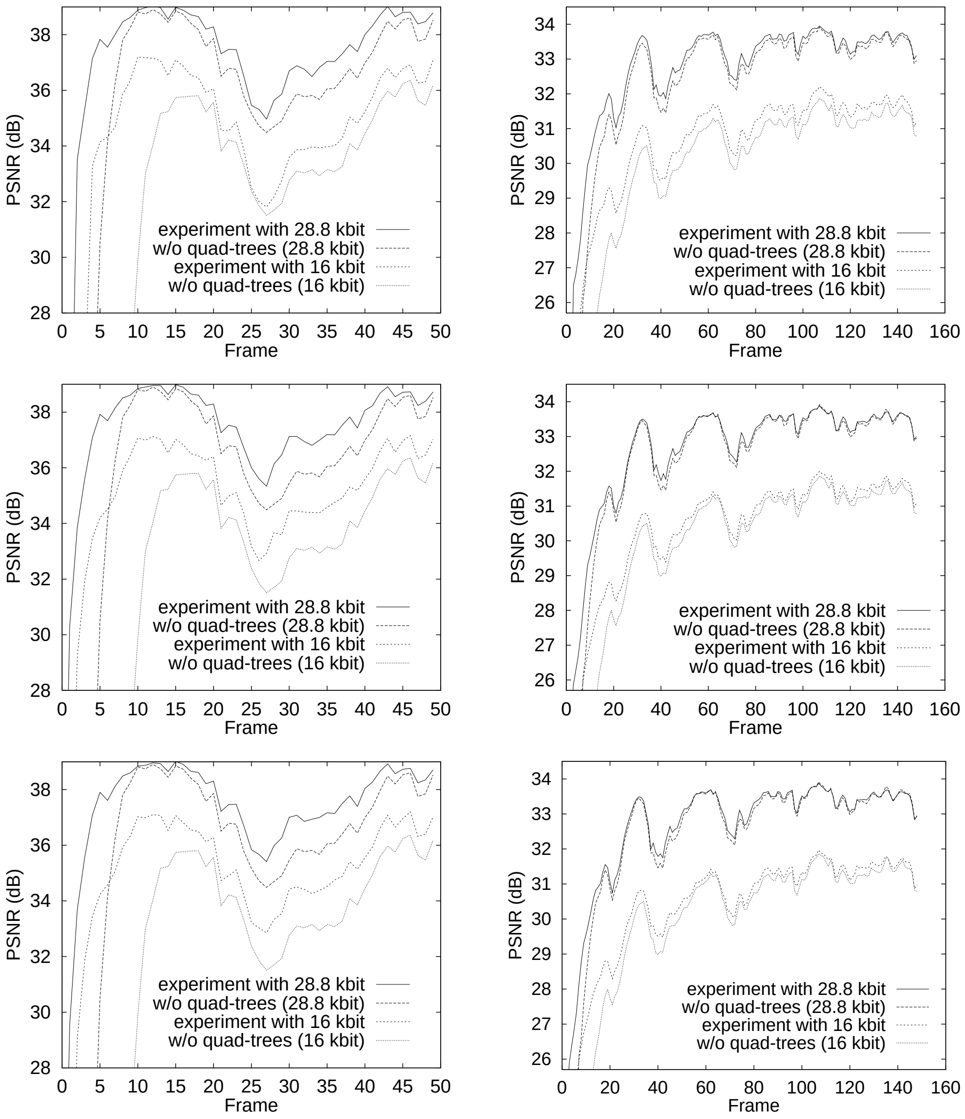
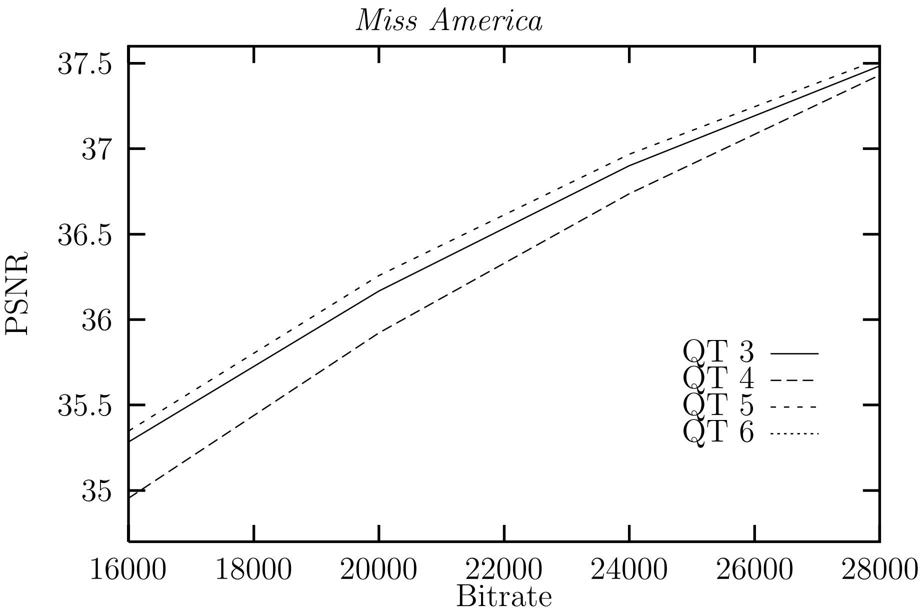


Figure 4.21: PSNR courses of (row by row) QT 4, 5 and 6 for (column by column) *Miss America* and *Salesman*.



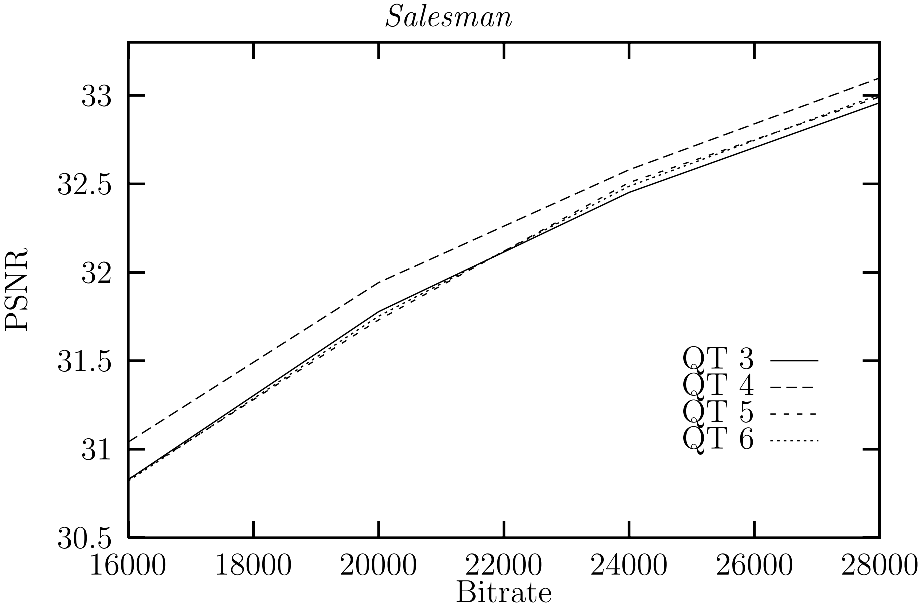


Figure 4.22: Average PSNR values for the quad-tree experiments. The corresponding plot for *Mother & Daughter* can be found in Fig. B.9 on Page 165.

Bitrates	*Miss America*	*Salesman*	*Mother & Daughter*
16000	219	180	248
28000	279	221	303

Table 4.3: Average number of bits needed to encode the position of mode-1 and mode-2 vectors with a quad-tree structure.

other things, improve the encoding of the positions of the mode-1 and mode-2 vectors. This is now proven by Tab. 4.3. This table shows, corresponding to Tab. 4.2, the costs for the quad-tree codec. The costs are more than halved. Thus, the encoding of the position of mode-1 and mode-2 vectors can be made more efficient as previously asserted.

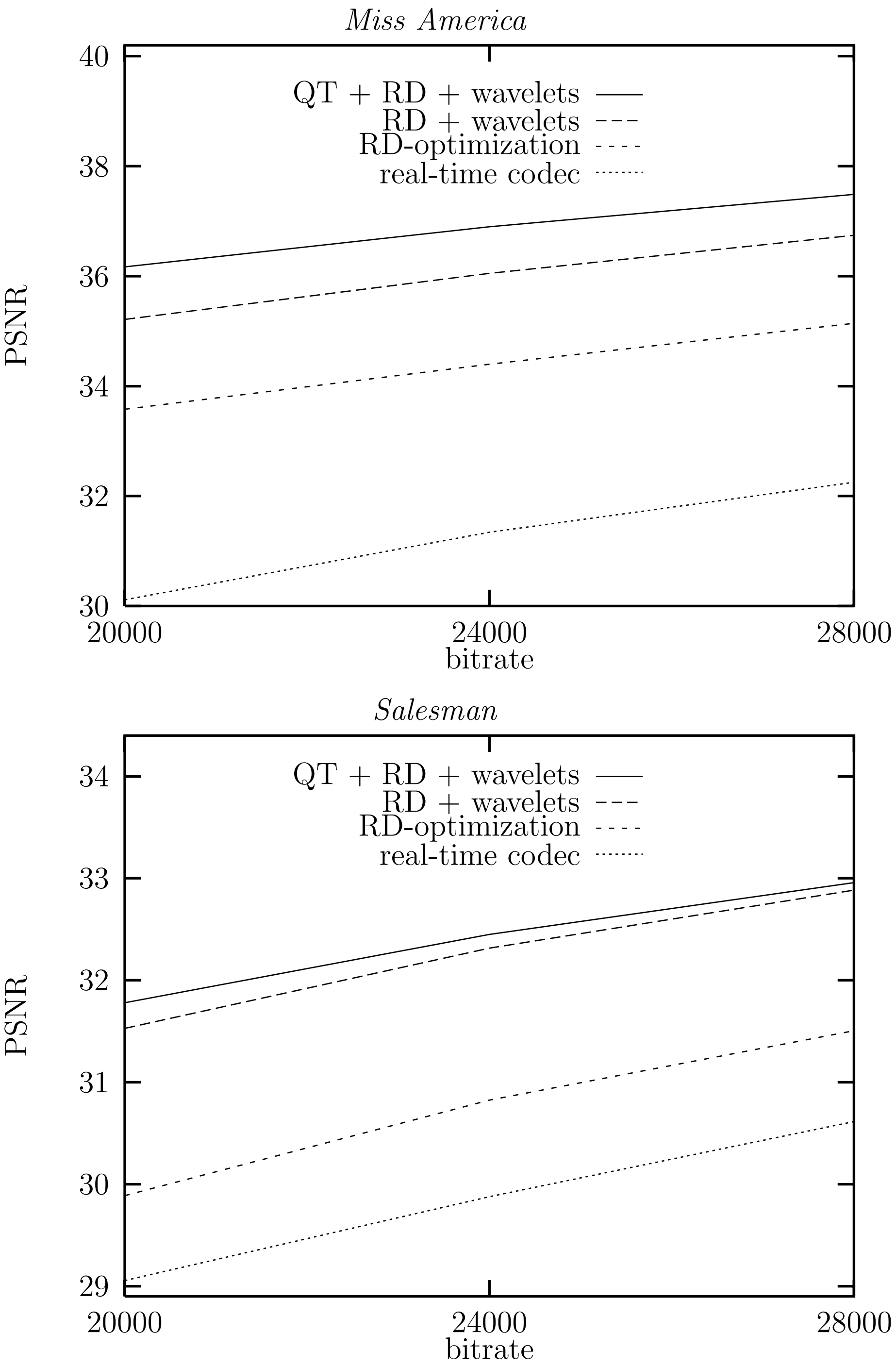
4.3.3 Comparison

We now compare the quad-tree based codec with our previously developed AVQ codecs.

In Fig. 4.23, we present the average PSNR values of the test sequences for different bitrates. The real-time codec from Sect. 4.1.1, its RD-optimization (Sect. 4.1.3), the application of wavelets (see Sect. 4.2) and, finally, the quad-tree based codec developed in Sect. 4.3.1 are compared.

The figure shows that we achieved an improvement ranging between 3 dB (*Salesman*) and 6 dB (*Miss America*). Different techniques lead to different improvements. The RD-optimization of the real-time codec yields the largest improvement for *Miss America* and *Mother & Daughter*, whereas, the largest gain for *Salesman* results from the wavelet transform. Some frames of *Salesman* are presented in Fig. 4.24 for the real-time and the quad-tree based AVQ-codec.

Unfortunately, only a small number of results have been reported in the literature for VQ coding of video sequences that are suitable for a comparison. To give the reader some sort of measure at hand to assess the performance of our codec, we provide a comparison with the tmn codec (H.263) [1, 42]. The tmn codec is used with syntax based arithmetic coding at a bitrate of 8000 bit/s and a frame rate of 8.3 frames/s. The target bitrate is achieved by the


Miss America

Salesman

Figure 4.23: Comparison of all AVQ-codecs. The corresponding figure for *Mother & Daughter* can be found in Fig. B.10 on Page 165.

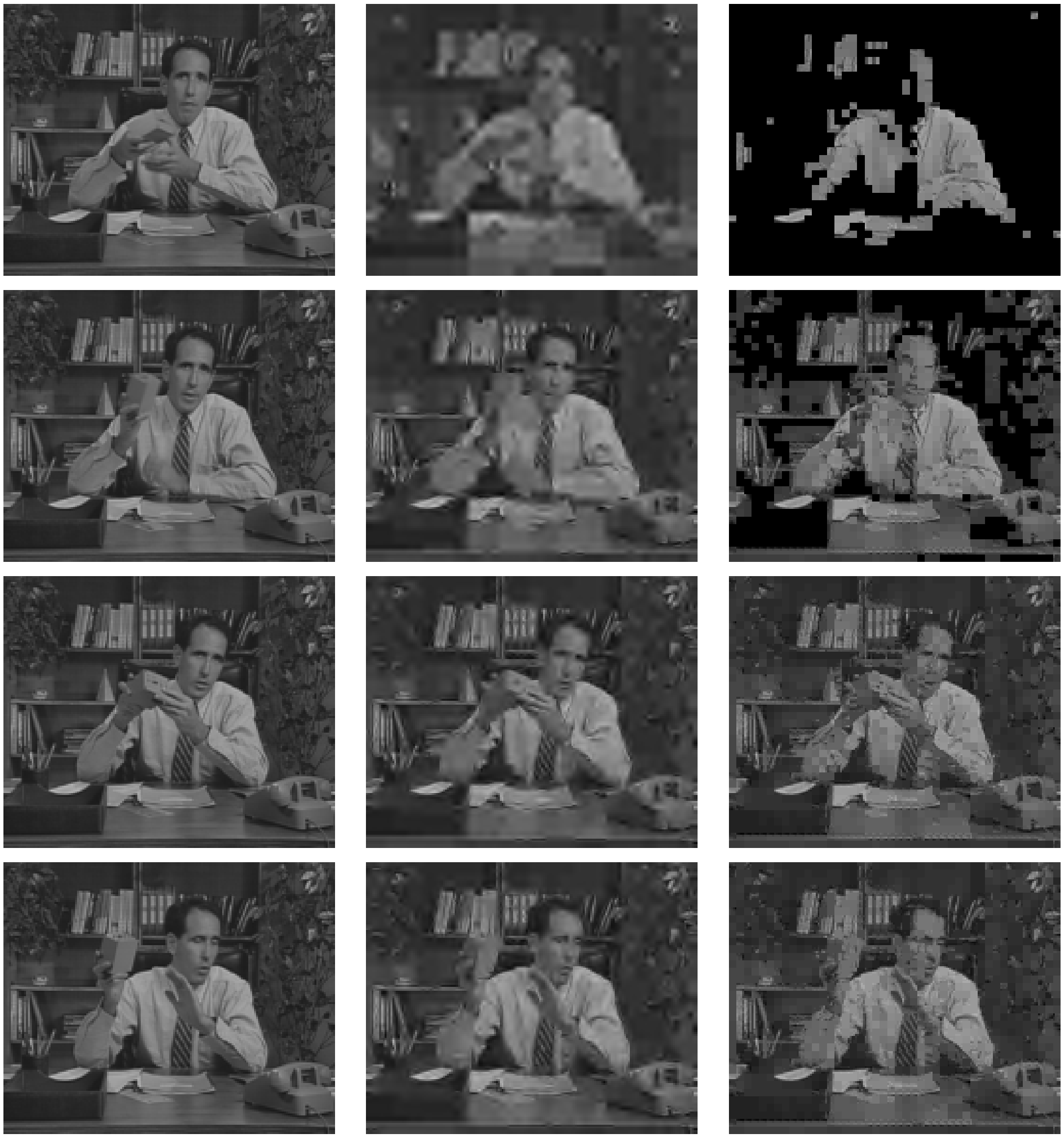


Figure 4.24: Line by line (top down): The 5th, 20th, 50th, and 100th frame of the *Salesman* sequence. Column by column (left to right): The original sequence, the decoded sequence of the quad-tree codec, and the decoded sequence of the real-time version at a bitrate of 16 kbit/s.

off-line rate control method of the tmn codec, i.e., the codec tries to adjust an average bitrate of $\frac{8000}{8.3}$ bit/frame over the whole sequence. The result is compared with the quad-tree based AVQ-codec (see Fig. 4.25). For *Mother & Daughter* one can see that at the beginning the performance gap between these two encoding schemes is very large. The gap is decreasing during the encoding of the first 100 frames. After that, an average difference of about 1.2 dB PSNR between the tmn and the quad-tree codec is maintained over the rest of the sequence. After subsequent 100 frames the average performance gap is about 0.9 dB PSNR. A similar observation can be made for *Salesman*. This shows the adaptability of our approach that needs about 100 frames to evolve. Note that, unlike H.263, we are not using motion compensation.

4.4 Motion Compensation

In the last sections we successively improved the AVQ based codecs. Up to now, the codecs did not apply block based motion compensation (MC). Therefore, in this section we describe experiments with this technique. However, the concept of macroblocks consisting of wavelet transform coefficients is not suitable for a block based MC scenario. Since the classical wavelet transform is not shift-invariant and advanced methods trade shift invariance in the wavelet domain for an overcomplete wavelet representation [77, 69], which is not desirable for image and image sequence compression, we cannot apply MC in the wavelet domain. On the other hand, using MC in the spatial domain makes it impossible to decide for each macroblock in the wavelet domain separately whether it should be motion compensated or not. Thus, motion information cannot be encoded efficiently.

For that reason, we focus on experiments exploring the performance of the AVQ-method that encodes motion compensated residual frames in the spatial domain. In addition, we compare this scheme with the performance of discrete cosine transform (DCT) coding.

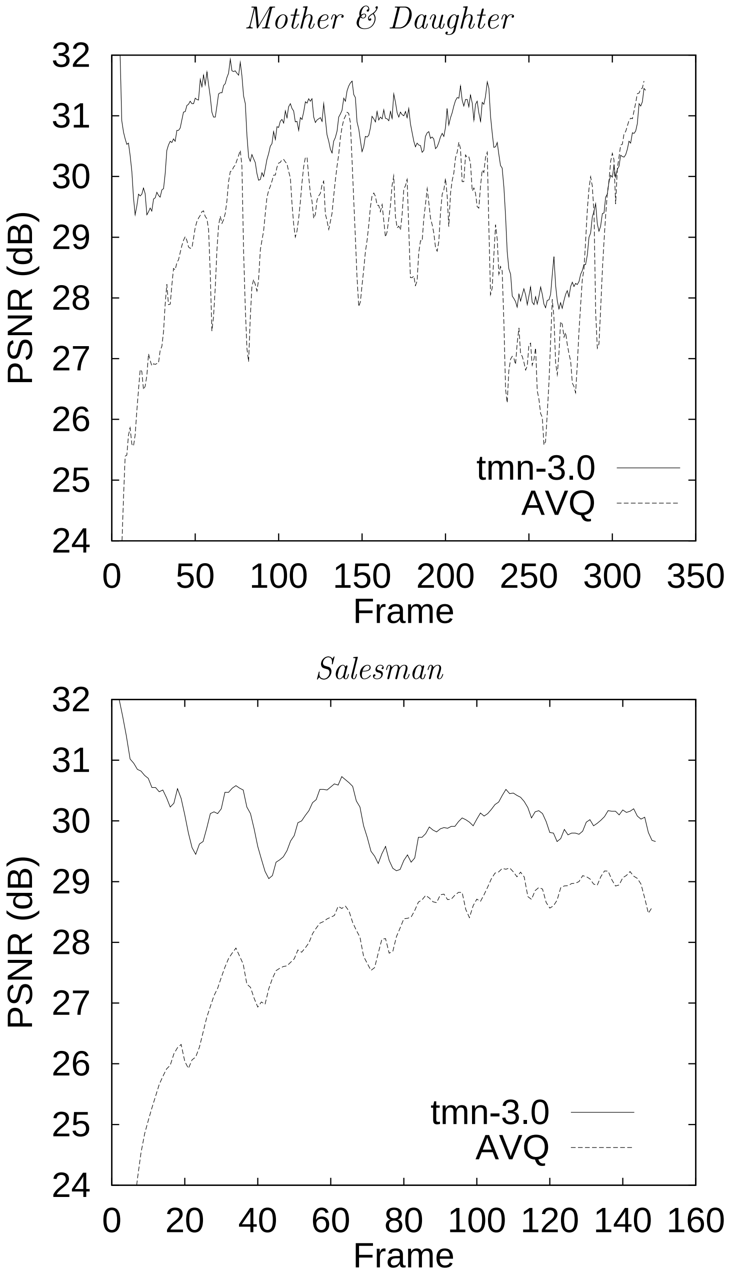


Figure 4.25: PSNR course for *Mother & Daughter* and *Salesman* for the tmn and the AVQ-codec at 8000 bit/s.

4.4.1 AVQ Codec with Motion Compensation

In this section, we describe the combination of AVQ and motion compensation. In order to exclude the influence of other parameters, we essentially use the H.263 encoding scheme substituting only the DCT encoding of the motion compensated residual by AVQ. The results of our first experiments have been published in [88].

Preprocessing. In the preprocessing stage, we divide the frame into M 16×16-pixel blocks, called macroblocks (MB). Then the basic block motion estimation scheme of the H.263 standard is applied on each MB. The motion estimation determines for the mth MB, $0 \leq m < M$, a motion vector (MV) $(v_x, v_y)_m$. Thereafter, we create for each macroblock three different types of blocks:

Intrablock. The macroblock without prediction.

Differenceblock. The residual block created by motion compensation with MV $(0, 0)$.

Interblock. The residual block created by motion compensation with MV $(v_x, v_y)_m$.

For every difference- and interblock, there are three possible decomposition levels. Level-0 describes the difference- and interblock as a whole. Alternatively, a level-0 block can be decomposed into four level-1 blocks of 8×8 pixel size. Each level-1 block, again, can be subdivided into four 4×4-pixel blocks, called level-2 blocks.

In order to create vectors, the blocks are scanned line by line. Thus, level-0 vectors are 256-dimensional, level-1 vectors are 64-dimensional, and level-2 vectors are 16-dimensional.

After the preprocessing, several vectors of several levels can be assigned to the M macroblocks. For the mth macroblock there are two level-0 vectors e_m^0 and $\tilde{e}_m^0$ corresponding to the difference- and interblock, respectively, and, in an analogous way, we have 4 level-1 vectors, $e_{4m}^1, \ldots, e_{4m+3}^1$ and $\tilde{e}_{4m}^1, \ldots, \tilde{e}_{4m+3}^1$, and 16 level-2 vectors, $e^2_{16m}, \ldots, e^2_{16m+15}$ and $\tilde{e}^2_{16m}, \ldots, \tilde{e}^2_{16m+15}$. In addition, there are the vector x_m, $0 \leq m < M$, scanned from the M intrablocks.

Encoding modes. For each macroblock, there are basically three encoding modes according to the the block types mentioned above: intrablock-mode, differenceblock-mode, and interblock-mode. In intrablock-mode the MB encoding scheme for intrablocks of H.263 is used. For difference- and interblocks the AVQ-approach combined with a quad-tree structure is applied. Hence, we have to specify the encoding modes for each of the different levels.

For level-2 encoding, there is only a replenishment mode. Note that replenishment in this case means padding the residual block with zeros. This can be interpreted as taking the motion prediction from the previously decoded frame as reference block. This justifies the term "replenishment" since it can be considered as a generalization of the replenishment of the last sections. In fact, for blocks predicted with MV $(0, 0)$, padding the residual block with zeros means exactly "taking the contents of the block at the same position in the previously decoded frame".

Level-1 vectors can be represented either by replenishment or by the quantized mean $\hat{\mu}_m$. The μ_m is quantized and encoded by Q_μ and γ_μ, respectively.

For level-2 vectors there are three encoding modes:

Mode 0. Replenishment mode as described above.

Mode 1. VQ mode. The prediction error is vector quantized, $Q(e^2_m)$ and $Q(\tilde{e}^2_m)$, and encoded with $\gamma(\alpha(e^2_m))$ and $\gamma(\alpha(\tilde{e}^2_m))$, respectively.

Mode 2. Update mode. The prediction error is transmitted as side information and inserted in the codebook. The vector is represented by $Q^*(e^2_m)$ and $Q^*(\tilde{e}^2_m)$ and is transmitted by $\gamma^*(\alpha^*(e^2_m))$ and $\gamma^*(\alpha^*(\tilde{e}^2_m))$, respectively.

The encoding of the modes of the macroblocks is done in the H.263 fashion.

Encoding parameters. The codebook organization of the vector quantizer as well as the vector decoder β is the same as described for the previous codecs.

The vector encoder α is realized by $\alpha(\lambda, x) = \arg\min_{i \in \mathcal{I}}[d(x, \beta(i)) + \lambda \cdot \gamma(i)]$. The rate distortion trade-off λ is determined by a RD-optimization. The universal vector encoder α^* uses a scalar quantizer Q_s to quantize the vector components separately. The binsize of Q_s is 11 and the deadzone is 23.

RD-Optimization. The RD-optimization can be achieved by an RD-algorithm for hierarchical dependences (cf. Sect. 3.3). The details of the quad-tree structure are analogous to that of Sect. 4.3.1.

The UV-component encoding is done in the H.263 fashion.

4.4.2 Results

In this section, we describe the results of the combination of the AVQ-approach with MC (AVQ-MC-codec).

The following experiment is carried out. First, we run the pure tmn-codec with a fixed quantization parameter q and the *syntax based arithmetic coder* option. The bits used for every frame are stored. After that, we apply the new AVQ-MC-codec on the same frames, using for every frame the same number of bits that the pure tmn-codec consumed. The frame format is QCIF and every third frame is taken.

Figure 4.26 shows the result of the experiment for *Mother & Daughter* and *Salesman* at 8000 bit/s. We see that at the beginning the AVQ-MC-codec performs worse than the tmn codec for both sequences. But after 50 frames, the performance is similar, and after 100 frames, the tmn codec is outperformed. The coding gain is 0.4 dB PSNR and 0.2 dB PSNR for *Mother & Daughter* and *Salesman.* The same experiment for 10000 bit/s is presented in Fig. 4.27. In this figure, the AVQ-MC-codec performs similar to the tmn codec for *Salesman. Mother & Daughter* performs slightly better for the AVQ-MC-codec. Both figures show, as in Sect. 4.3.3, the adaptability of the AVQ-MC-codec.

The results of our experiments show that, after an initialization phase, the AVQ performs better or at least as well as the DCT on motion compensated residual frames. Note that the AVQ-MC-codec uses RD-optimization for the mode-decision. In order to provide fair comparison the pure tmn codec should also apply RD-optimization for mode decision. However, applying RD-optimization solely on mode decision without optimizing motion compensation only leads to a very small improvement of less than 0.2 dB PSNR [83]. But optimizing motion compensation would also improve the results of our AVQ-MC-codec.

4.5 Summary and Discussion

In this chapter, we investigated the application of AVQ on image sequences. First, we described a fast AVQ codec that is able to encode image sequences in real-time using current PC technology. Then, we applied successively state-of-the-art techniques to improve the codec. The RD-optimization results in a smaller overall distortion for a given bit budget. The wavelet transform helps to encode the codebook update more efficiently and enables a better vector representation. Finally, the quad-tree structure leads to a better encoding of the vector positions in the frame and enables hierarchical macroblock coding.

Many details had to be taken into account. The update strategy of the codebook turned out to be an unproblematic design issue if carefully selected. The use of an adaptive arithmetic coder significantly improves the entropy coding. And, finally, an efficient quad-tree structure was determined.

However, comparison of the quad-tree codec with standard transform coding shows that inspite of the significant improvement over the previous versions it still leads to a performance gap of about 1 dB PSNR. We conclude that motion compensation is essential also for codecs based on AVQ.

Therefore, we studied the performance of AVQ applied on motion compensated residual frames. The experiments we made show that AVQ improves at least to the level of DCT coding and beyond.

We now discuss the relation of our AVQ to the schemes previously appeared in the literature (see Sect. 2.4). The real-time codec applies a distortion threshold *tol* to decide whether a block is encoded in mode 1 or mode 2. This strategy at first sight is very similar to the Paul algorithm. On the other hand,

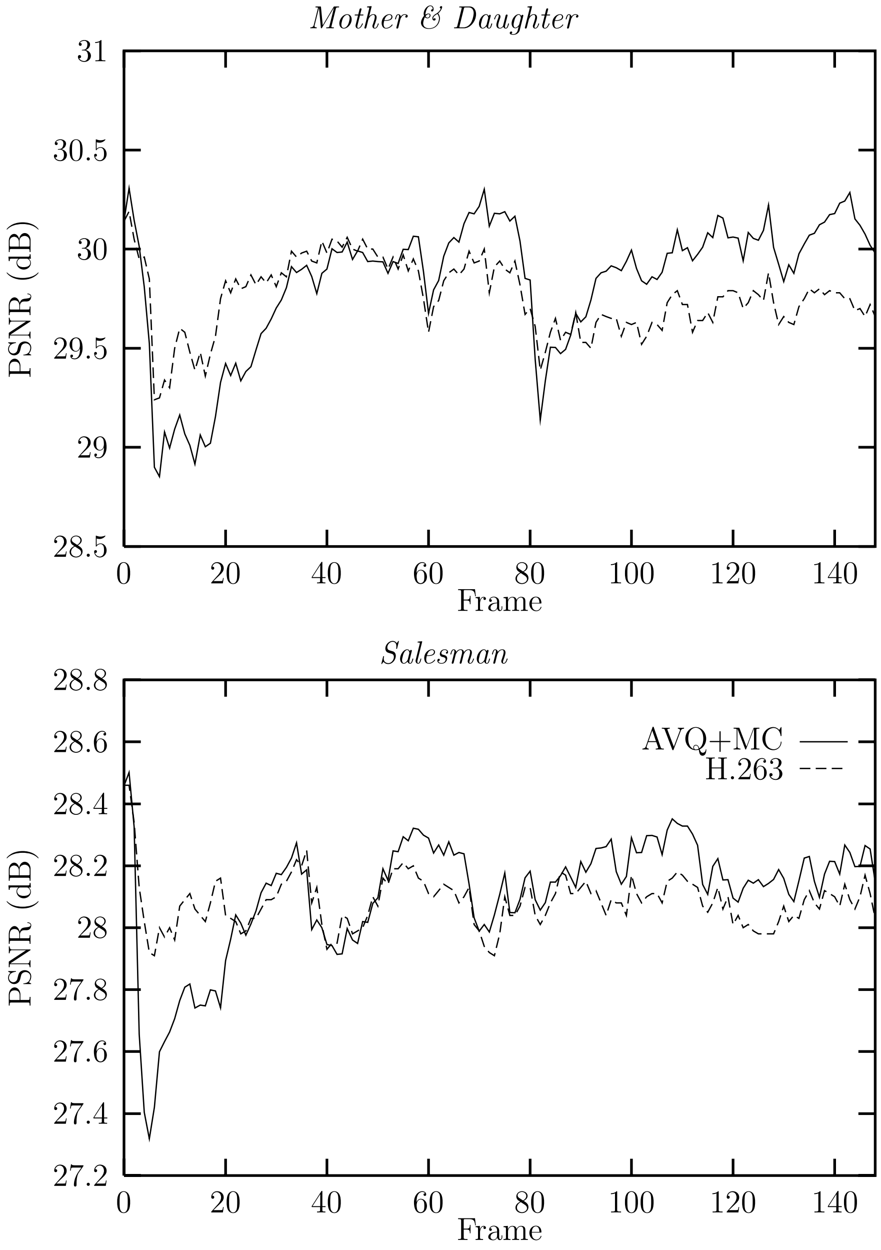


Figure 4.26: PSNR course for the tmn and the AVQ-MC-codec for a bitrate 8 kbit/s.

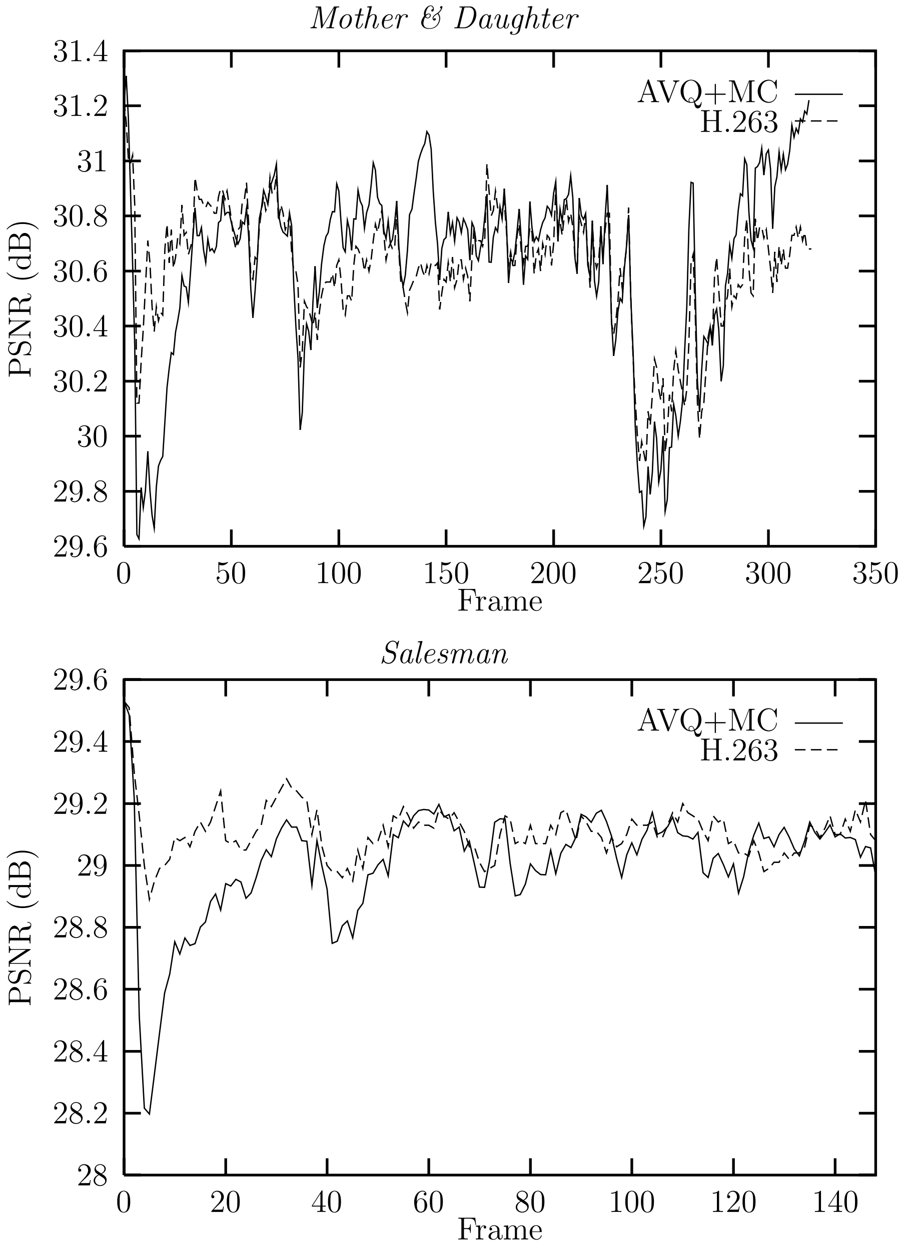


Figure 4.27: PSNR course for the tmn and the AVQ-MC-codec for a bitrate 10 kbit/s.

the real-time codec also observes the bitrate and breaks off the encoding if a predetermined bit budget is exhausted. Moreover, unlike the Paul algorithm the blocks are processed according to a priority list, and, thus, the blocks of the whole frame have to be known in advance. Therefore, the AVQ-scheme of the real-time codec should rather be considered as a constrained-rate AVQ algorithm.

The AVQ-scheme of the other codecs presented in this chapter applies a rate-distortion trade-off to decide in which mode a block has to be encoded. Therefore, it can be considered as rate-distortion-based. If we neglect the search for the appropriate value of λ, the AVQ-scheme in this case is very similar to the online algorithm of Fowler. The main differences are the additional mode-0 encoding, the fact that vectors selected for the insertion in the codebook are not inserted immediately but after the encoding of a frame has been completed, and the different codebook update strategies. Moreover, taking into account the RD-optimization, our AVQ-scheme is not even an online algorithm since all blocks in a frame are analyzed prior to the encoding.

Chapter 5

Complexity Evaluation of RD-Techniques for AVQ Video Coding

In this chapter, we will evaluate the complexity of rate-distortion (RD) optimization techniques of Chapter 3 for the AVQ-codecs described in Chapter 4. In particular, we compare the complexity of the Lagrangian multiplier (LM) algorithm with the incremental computation of the convex hull (ICCH) approach. To do so, we introduce an implementation independent complexity measure and investigate the two approaches accordingly (Sect. 5.1). In Sect. 5.2 we then show how to speed up the LM-algorithm.

5.1 Comparison of the LM- and the ICCH-Algorithm

In our study, we consider the codec of Sect. 4.3. The frame size is QCIF; therefore we have 99 macroblocks and, counting only level-1 and level-2 blocks, $M = 20 \cdot 99 = 1980$ blocks. For all nodes we assume the same number of RD-points, $N = 514$, the same vector dimension, $L = 15$, and the structure of level-2 vectors.

First, we analyze the complexity of the computation of the local sets, $\mathcal{P}_t$, of

RD-points. Obviously, the complexity is dominated by the computation of the distortions for each encoding option. The encoding options derive from the different codebook vectors that could represent a certain part of the macroblock. To compute the sum of squared errors for level-2 blocks, 15 subtractions, 15 multiplications and 14 additions are necessary. One further addition is needed to take into account the distortion of the lowpass coefficient. Thus, 45 operations[1] are needed to compute one RD-point for one subblock. The total number of operations to compute the RD-points is $45 \cdot N \cdot M$.

Now we consider the LM-algorithm (Alg. 7) for hierarchical dependences. Here, the complexity is dominated by the $\arg\min$ computation. This computation is made by one multiplication, one addition, and one comparison per RD-point. Let A be the number of iterations that are necessary to find an appropriate value of λ. Then the LM-algorithms needs $3 \cdot A \cdot N \cdot M$ operations.

For the ICCH-algorithm (Sect. 3.3.3), we analyze the complexity of the different procedures FIND_NEXT_LAMBDA, INIT, SEARCH, and UPDATE. The search for the next λ, FIND_NEXT_LAMBDA (Alg. 10), is dominated by the for-loop and the $\arg\min$ computation. Notice that the comparison $(D_t[i] <= D_t[f_c(t)]))$ serves the purpose to speed up the computation since it excludes many RD-points from the subsequent computation of λ_h. However, if we consider the worst case complexity, this computation is not essential for the correctness of the algorithm and, thus, may be omitted. Therefore, for every RD-point we have one comparison, two additions, and one division within the for-loop and one comparison for the computation of the $\arg\min$ expression, a total of 5 operations per RD-point. The initialization, INIT (Alg. 9), needs the most operations for the $\arg\min$ computation and in FIND_NEXT_LAMBDA. Thus, we have 6 operations per RD-point. The algorithms SEARCH (Alg. 11) and UPDATE (Alg. 12) use the most operations in the procedure FIND_NEXT_LAMBDA. Let B be the number of calls of FIND_NEXT_LAMBDA from SEARCH during the optimization. Then the algorithm requires $M \cdot N \cdot 6$ operations for the initialization and $5 \cdot B \cdot N$

[1] We assume equal complexity of the addition, multiplication, division and comparison operation. This is, however, platform dependent.

bitr.(bit/s)	8000	16000	28800
Miss Am.	12.2	12.5	12.5
Salesm.	12.4	12.8	12.9
Mthr.	12.5	12.8	13.1

(a)

bitr.(bit/s)	8000	16000	28800
Miss Am.	9.1	9.8	10.3
Salesm.	8.8	9.8	9.8
Mthr.	9.1	9.8	10.3

(b)

Table 5.1: Number of LM-iterations (a) without prediction of λ from the last frame and (b) with prediction of λ from the last frame.

bitr.(bit/s)	8000	16000	28800
Miss Am.	6.2	6.1	5.2
Salesm.	7.0	7.1	6.0
Mthr.	7.4	6.8	6.6

(a)

bitr.(bit/s)	8000	16000	28800
Miss Am.	2.5	2.7	1.9
Salesm.	3.6	3.5	2.8
Mthr.	3.6	3.4	3.0

(b)

Table 5.2: Number of LM-iterations for an approximate solution (a) with 1% tolerance and (b), with 10% tolerance.

operations for the determination of the optimal LCH solution. Since the level-2 blocks are dominating, the case that FIND_NEXT_LAMBDA is called in the UPDATE procedure is of no significance. Thus, we assume that each call of FIND_NEXT_LAMBDA after the initialization by INIT is invoked by SEARCH. We denote one call of SEARCH by one *search step*.

Our first experiments concern the LM-algorithm. We run the codec with several bitrates on the test sequences *Miss America*, *Salesman* and *Mother & Daughter*. As initial interval, we take $[0, \infty]$ and use the fast convex search to find the value of λ that yields the closest LCH solution to the target rate, called the *optimal LCH solution*. The results of the average number of LM-iterations

bitrate (bit/s)	8000	16000	28800
Miss America	200	351	572
Salesman	232	344	457
Mthr. & Dotr.	247	386	554

Table 5.3: Average number of search steps for Alg. 11.

is shown in Tab. 5.1a.

The next experiment examines predictability of the value of λ using the value of λ from the previous frame (see Sect. 3.2.2). As in the last experiment, the codec is run for several bitrates and sequences. The average number of LM-iterations needed in this case can be seen in Tab. 5.1b.

The LM-algorithm can also be applied to find an approximation of the optimal LCH solution. A tolerance is defined and if the bitrate that was computed for a value of λ deviates from the target rate less than the tolerance does, the search is stopped. The results of the average number of iterations are depicted in Tab. 5.2a for a tolerance of 1% and in Tab. 5.2b for a tolerance of 10%. Note that in this case the initial search interval is predicted from the last frame, too.

From the tables it can be gathered that the LM-algorithm needs 12.2–13.1 iterations without prediction of λ, 8.8–10.3 iterations with prediction of λ, 5.2–7.4 iterations with a 1% tolerance approximation, and 1.9–3.6 iterations with a 10% tolerance. In addition, we can see that the number of iterations to a slight degree is dependent on the bitrate. In Tab. 5.1 the number of iterations increases with increasing bitrate and, vice versa, in Tab 5.2 the number of iterations decreases with increasing bitrate.

The most iterations are needed by the algorithm without prediction. The algorithm with prediction uses about 3 less iterations. This means that the λ can be predicted from the preceding frame, but the prediction is not much better than starting with the $[0, \infty]$ interval. The approximating algorithms behave as expected. Stopping within a tolerance is more efficient that searching for the optimal LCH solution since in the worst case the approximating algorithm stops if the optimal LCH solution has been found. Thus, the approximating algorithm needs less iterations. Apparently, the algorithm with a larger tolerance needs less or at most the same number of iterations compared to the algorithm with a smaller tolerance. Therefore, the difference between the algorithm with 10% tolerance and 1% tolerance is not surprising.

Only the dependence on the bitrates is not that obvious at first sight. In order to find an explanation, we must take a look at the lower convex hull

(LCH) of the global set of RD-points for the different bitrates. An example for this can be seen in Fig. 5.1. We count the average number of RD-points on the LCH of the global set $\mathcal{P}_{\mathcal{I}}$ within the rate interval $[500, 4000]$. This value is averaged for the different bitrates over all frames of one sequence excluding the first 15 frames from the computation. The result is presented in Tab. 5.5. From this table it can be seen that the LCH of the global set of RD-points is denser "populated" for higher rates. This explains the rate dependence in Tab. 5.1. The search algorithm for higher rates is employed with "fine-tuning" of the solution whereas the lower rates determine faster the optimal LCH solution with a larger gap to the target rate. The rate dependence in Tab. 5.2 can be explained as follows. The absolute value of the tolerance becomes larger with increasing target rates. Thus, for larger target rates a satisfying solution can be found faster, provided that the LCHs of the global sets have similar dense "population" of RD-points for different bitrates. The fact that the number of the RD-points on the LCH increases with the rate further increases the probability to determine faster a tolerable solution.

Now we consider the complexity of the ICCH-algorithm. The average number of search steps needed to find the optimal LCH solution is presented in Tab. 5.3. Note that one LM-iteration has about $\frac{3}{5} \cdot M = 1188$ times the complexity of one ICCH-step. Thus, without the initialization, the complexity of the ICCH-algorithm is much less than one LM-iteration. The complexity of the ICCH-initialization is two times of the complexity of one LM-iteration, however, in contrast to the LM-iteration, the estimation of the complexity of one ICCH-step as well as of the ICCH-initialization is a worst case estimation. Altogether, the ICCH-algorithm needs about one-fourth of the complexity of the LM-algorithm to find the optimal LCH solution. Even if the LM-algorithm seeks a solution with tolerance 10%, the ICCH algorithm performs better.

In Tab. 5.4, we provide run-time experiments made on a Silicon Graphics O2 with a 150 MHz R10000 processor. This table contains the run-time for computation of the local RD-sets, the direct computation of the LCH for all local RD-sets with *Graham's scan* (cf. [53, p. 275ff]), the LM-algorithm that predicts the initial search interval from the previous frame, ICCH initialization, and ICCH search steps. Furthermore, the codec was compiled with and without compiler optimization. Without optimization, the ICCH-algorithm, including initialization and search steps, is around 7 times faster than the LM-approach. The time needed for the LM-approach is about the same order of magnitude as the computation of RD-sets. Actually, the LM-algorithm performs as expected compared with the computation of the RD-sets since, e.g., for *Salesman*, we have $\frac{3.65}{2.27} \approx \frac{45 \cdot N \cdot M}{3 \cdot A \cdot N \cdot M}$. Note that the LM-algorithm needs significantly more time than the direct computation of the LCH of the local sets $\mathcal{P}_t$ by Graham's scan. Surely, the computation of the local LCHs is not equivalent to the computation of the LCH of the global set $\mathcal{P}_{\mathcal{T}}$, but computing the global LCH from the known local LCHs has a smaller complexity than the ICCH search steps and, thus, may be neglected. Also notice that in contrast to the implementation-independent estimation the ICCH-init procedure needs roughly the time of one LM-iteration. This is due to the fact that, as mentioned above, the estimation for the ICCH-init procedure is a worst case estimation since many operations are applied only conditionally.

Apparently, the performance of the two RD-optimization algorithms depends on the statistical characteristic of the RD-points. In Fig. 5.2 we present typical local RD-sets, $\mathcal{P}_t$. Figure 5.2a shows the RD-points of a block encoded in mode 1 and Fig. 5.2b shows the RD-points of a block encoded in mode 0. From these figures it can be seen that only a small number of points are located on the LCH of the local sets. Thus, only a few search steps are needed to traverse the LCH. In addition, the ICCH algorithm excludes the blocks with low activities, i.e., blocks that will be encoded in mode 0 after RD-optimization, from further investigations since the optimization procedure automatically knows after the initialization that the first slope in Fig. 5.2d is very flat and needs only to be reconsidered if the steeper slopes of all active blocks, like in Fig. 5.2c, have been processed. Therefore, since a large number of blocks shows low activity due to their image background membership, this feature of the LCHs of the local RD-sets seems to be suitable for the ICCH-algorithm.

From Tab. 5.4a, it can be seen that our AVQ-codec based on the ICCH

	Miss America	*Salesman*	*Mthr. & Dotr.*
computing RD-sets	3.63s	3.65s	3.63s
direct comp. of LCH	1.08s	1.03s	1.16s
LM-algorithm	2.19s	2.27s	2.39s
ICCH init	0.28s	0.28s	0.29s
ICCH search steps	0.06s	0.05s	0.06s

(a)

	Miss America	*Salesman*	*Mthr. & Dotr.*
computing RD-sets	1.02s	1.02s	1.02s
direct comp. of LCH	0.50s	0.46s	0.53s
LM-algorithm	1.67s	1.73s	1.83s
ICCH init	0.10s	0.10s	0.10s
ICCH search steps	0.04s	0.03s	0.04s

(b)

Table 5.4: Run-time per frame of the AVQ-codec at 28800 bit/s compiled (a) without (b) with additional compiler optimization.

approach needs less than 1.2 seconds per frame for vector quantization and RD-optimization. Together with the wavelet transform (not optimized) one frame is encoded in less than 1.5 seconds.

bitrate (bit/s)	8000	16000	28800
Miss America	259	321	357
Salesman	147	191	224
Mother & Daughter	201	222	240

Table 5.5: Average number of RD-points ont the global LCH within $[500, 4000]$ for several bitrates and sequences.

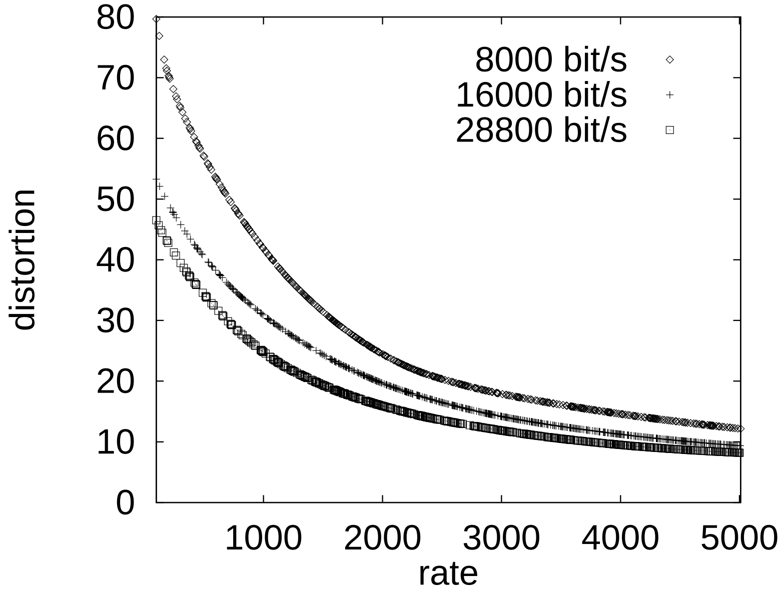


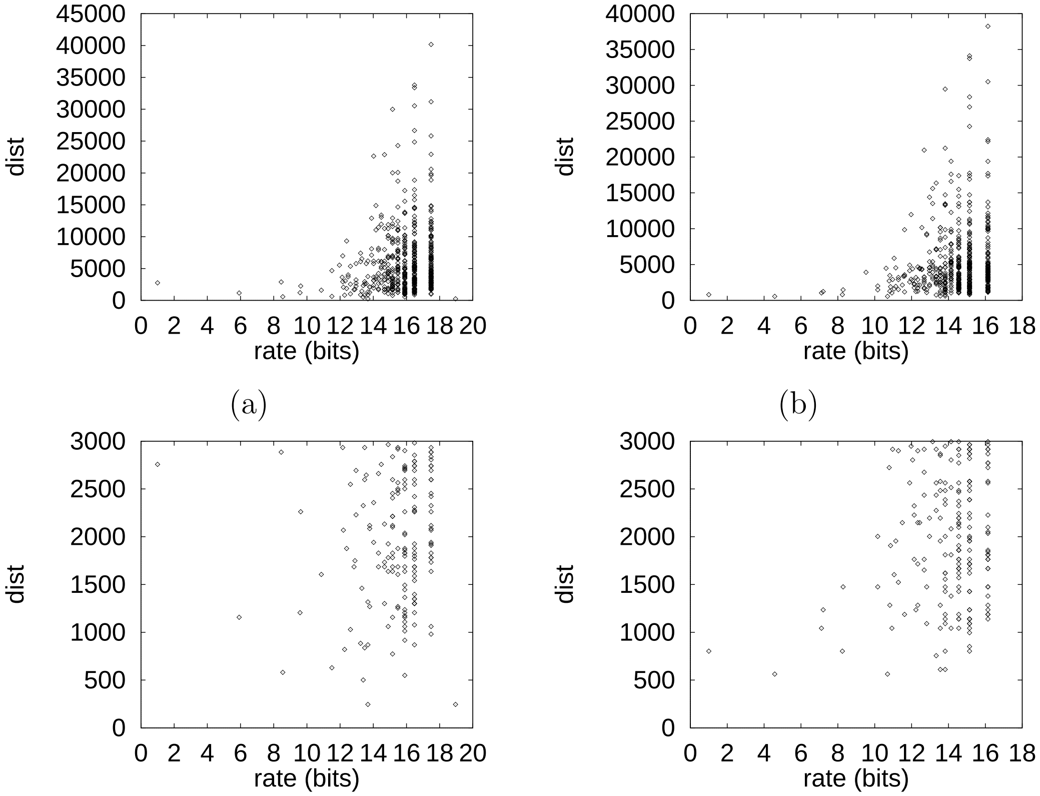
Figure 5.1: Points on the LCH in the global set of RD-points for the 30th frame of *Miss America* for different bitrates. The distortion is measured in MSE per pixel.

5.2 Speed up Techniques for the LM-Algorithm

In this section, we develop a technique to speed up the hierarchical LM-algorithm (Alg. 7). The main idea is to reduce the complexity of the arg min computations. This can be done with an approach based on Proposition 6 (Page 61). This proposition shows that the rates $R^*(\lambda)$ of the solution of (3.13) are monotonically decreasing with λ. From Corollary 1 (Page 52) we know that the rate $\tilde{R}^*_t(\lambda)$ of the optimal solution of

$$\min_{i\in\mathcal{I}} \tilde{D}^i_t + \lambda \cdot \tilde{R}^i_t$$

is also monotonically decreasing with λ, where $(\tilde{R}^i_t, \tilde{D}^i_t) \in \tilde{\mathcal{P}}_t$ and $\tilde{\mathcal{P}}_t \overset{\text{def}}{=} \{(R^n_t, D^n_t) : 0 \leq n < N\}$ consists of all local points $(R^i_t, D^i_t) \in \mathcal{P}_t$ except the point for the case t is an internal node. This means that the range of the arg min computation in Alg. 7 can be bounded using informations from previous LM-iterations. For example, let us assume that the preceding LM-iteration



(a) (b)

(c) (d)

Figure 5.2: Local sets of RD-points of blocks in the 30th frame of *Miss America*: The block was encoded in (a) mode 1 and (b) mode 0. The segments of (a) and (b) containing the LCH are depicted in (c) and (d).

used λ_i and found a rate $R^*(\lambda_i)$ that is smaller than the target rate, R_T. Then every reasonable algorithm that seeks a λ_T yielding an optimal LCH solution will apply a λ_j that satisfies $\lambda_j < \lambda_i$ in further iterations since $R^*(\lambda_j) > R^*(\lambda_i)$. Therefore, for every node $t \in \mathcal{T}$, we can safely assume that $\tilde{R}^*_t(\lambda_j) > \tilde{R}^*_t(\lambda_i)$ for all further iterations. Thus, each LM-iteration determines either a lower or an upper bound for the arg min search in every $t \in \mathcal{T}$. If the RD-points, $\tilde{\mathcal{P}}_t$, of all local sets, are sorted with respect to the rate, the upper and lower bound for the rate are corresponding to an index interval $[i^l_t, i^u_t]$. To find $\tilde{R}^*_t$ for the nodes $t \in \mathcal{T}$, only this interval has to be searched. Unfortunately, we cannot expect to have sorted RD-points at all and it is doubtful whether a sorting as preprocessing would improve the performance, since the lower bound of the complexity of sorting of all local sets is $\Omega(M \cdot N \cdot \log N)$. But if a sorting of N rates is sufficient to know the sorting of all sets, this technique can be promising. Apparently, this is the case for our RD-sets $\tilde{\mathcal{P}}_t$ since the rates are mainly determined by the index coders of VQ (cf. Sect. 4.3.1). Thus, we need only to sort the N VQ indices once to know the order of the RD-points for all $\tilde{\mathcal{P}}_t$. The additional points for mode 0 and mode 2 do not significantly affect the complexity. Furthermore, the rate from the lowpass coefficient that is added to the VQ index rates does not change the order of the index rates, since the rate of the lowpass coefficient is fixed for a given node t. Therefore, it can be expected that this technique speeds up the LM-algorithm.

However, it is even possible to improve the LM-iteration without restrictions to the rate order. We consider the case if $i^l_t = i^u_t$. Then the set $\tilde{\mathcal{P}}_t$ can be excluded from searches in further iterations. Otherwise the search must be performed within the full index range since the indices are not ordered with respect to the rate. In the following, we describe this generally applicable algorithm. The new LM-iteration is called *LM_iteration_bound*. We introduce an additional data structure to manage the indices of the upper and lower bound for each local set $\tilde{\mathcal{P}}_t$. The indices are denoted by i^l_t for the lower bound and by i^u_t for the upper bound. However, the two bounds can not be updated during the iteration. The problem is that, during the iteration for a fixed λ_i, it is not possible to decide in every node $t \in \mathcal{T}$ whether the index that was found by the $\arg\min$ computation describes the upper or the lower bound until the rate $R_t^*(\lambda_i)$ has been computed and is compared with R_T. Thus, in iteration i, only the type of bound of the index determined in the previous iteration, $i-1$, is known. Therefore, we have to store the result of the $\arg\min$ operation to make it accessible in the next iteration. This is made by i_t^{last}. The variables have to be initialized for all local sets of RD-points by $i_t^u = N+1$, $i_t^l = -1$ and $i_t^{\text{last}} = 0$. The new LM-iteration is depicted in Alg. 13. The parameter *ulstat* indicates if the last iteration was a lower bound $(\text{ulstat} = 0)$, upper bound $(\text{ulstat} = 1)$ or if it is not known (e.g. $\text{ulstat} = -1$).

In the following we compare the complexity of the two LM-algorithms, the LM-algorithm that applies the original LM-iteration as described in Alg. 7, called *LMO-algorithm*, and the LM-algorithm that uses the new LM-iteration (Alg. 13), called *LMN-algorithm*. An implementation-independent performance measure is given by the number a of $\arg\min$ computations. Apparently, in the LMO-algorithm this is directly dependent on the number of iterations A, $a = A \cdot M$. For the LMN-algorithm, the number of $\arg\min$ computations must be counted during the optimization. Table 5.6a shows the average number of $\arg\min$ computations needed by the LMN-algorithm to find the optimal LCH solution for several bitrates and sequences. The initial search interval is predicted from the preceding frame. The $\arg\min$ operations for the LMN-algorithm with prediction of the initial search interval and the 10% tolerance version are provided in Tab. 5.6a and 5.6b.

It can be seen that the complexity of the LMN-algorithm is about four to five times lower than the LMO-algorithm. In addition, the LMN-algorithm needs even less $\arg\min$ computations than the 10% tolerance version of the LMO-algorithm even though the LMN-algorithm computes the optimal LCH solution.

The run-time experiments presented in Tab. 5.7 show similar results. The new LM-algorithm without an optimized compilation is about 3.5 times faster than the old LM-algorithm. For the version that has been compiled with an optimizing flag, the new LM-algorithm is even about 4.5 times faster.

The average number of the $\arg\min$ computations in the different LM-

bitrate (bit/s)	8000	16000	28800
Miss America	4205	4292	4450
Salesman	4360	4474	4348
Mthr. & Dotr.	4352	4442	4494

(a)

bitrate (bit/s)	8000	16000	28800
Miss America	18018	19404	20394
Salesman	17424	19404	19404
Mthr. & Dotr.	18018	19404	20394

(b)

bitrate (bit/s)	8000	16000	28800
Miss America	4950	5346	3762
Salesman	7128	6930	5544
Mthr. & Dotr.	7128	6732	5940

(c)

Table 5.6: Number of $\arg\min$ computations for (a) the LMN-algorithm, (b) the LMO-algorithm with prediction of the initial search interval, and (c) the LMO-algorithm with 10% tolerance.

iterations of the LMN-algorithm is depicted in Fig. 5.3. This figure shows that in the first two iterations there are M $\arg\min$ computations. This is always the case since only in the third iterations exists a defined upper and lower bound for the sets $\tilde{\mathcal{P}}_t$. Then the number of $\arg\min$ operations declines fast. After the eighth iteration, the number of $\arg\min$ operations is near to zero. The most local sets, $\tilde{\mathcal{P}}_t$, are excluded from the $\arg\min$ computation.

The implementation-independent complexity measure suggests that the LMN-algorithm is as fast as the ICCH-algorithm since the LMN-algorithm has the complexity of about 2.5 LM-iterations of the LMO-algorithm. Yet, the run-time experiments show that the ICCH-algorithm is more than two times faster. This can be explained by the fact that the complexity estimation for the ICCH-algorithms is, unlike the LM-algorithm, a worst case estimation (cf. Sect. 5.1).

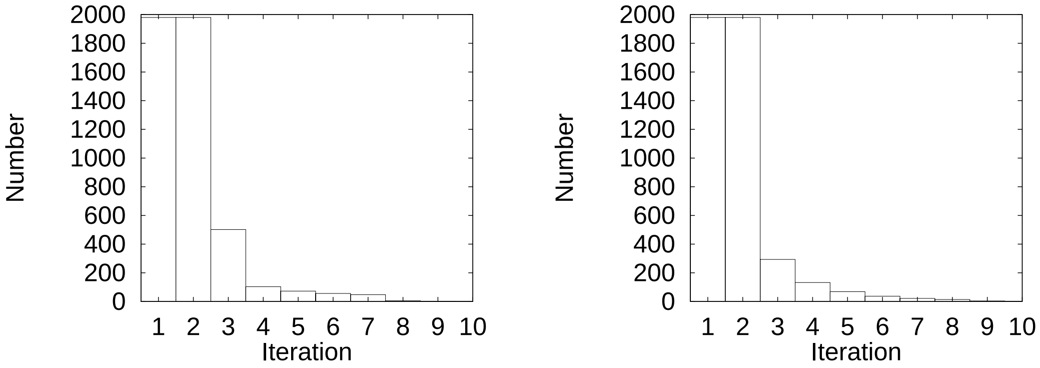
bitrate (bit/s)	8000	16000	28800
Miss America	0.62s	0.64s	0.65s
Salesman	0.64s	0.67s	0.65s
Mthr. & Dotr.	0.64s	0.66s	0.67s

(a)

bitrate (bit/s)	8000	16000	28800
Miss America	0.38s	0.39s	0.42s
Salesman	0.41s	0.42s	0.41s
Mthr. & Dotr.	0.40s	0.41s	0.42s

(b)

Table 5.7: Run-time experiments for the LMN-algorithm: compiled (a) without and (b) with compiler optimization flag.



(a) (b)

Figure 5.3: Average number of arg min computations in the different iterations of the LMN-algorithm: for (a)*Miss America* and (b)*Salesman* at 28800 bit/s.

Algorithm 13 Improved LM-iteration.

```
Given: RD-parameter $\lambda$
LM_iteration_bound(node $t$,int ulstat)
{
if $t \in \mathcal{T} \setminus \tilde{\mathcal{T}}$ then
    LM_iteration_bound(left($t$),ulstat)
    LM_iteration_bound(right($t$),ulstat)
else
    $J^*_{\text{left}(t)} \leftarrow \infty$
    $J^*_{\text{right}(t)} \leftarrow \infty$
end if
if  ulstat=0 then
    $i^l_t \leftarrow i^{\text{last}}_t$
end if
if  ulstat=1 then
    $i^u_t \leftarrow i^{\text{last}}_t$
end if
if  $i^l_t = i^u_t$ then
    equal($t$) $\leftarrow 1$
end if
$J^{\sum}_t \leftarrow J^*_{\text{left}(t)} + J^*_{\text{right}(t)} + (D^N_t + \lambda \cdot R^N_t)$
if  not equal($t$) then
    $i_t \leftarrow \arg\min_{0 \leq i < N}[D^i_t + \lambda \cdot R^i_t]$
else
    $i_t \leftarrow i^l_t$
end if
$i^{\text{last}}_t \leftarrow i_t$
$J_t \leftarrow D^{i_t}_t + \lambda \cdot R^{i_t}_t$
if $J^{\sum}_t < J_t$ then
    $i_t \leftarrow N$
    $J^*_t \leftarrow J^{\sum}_t$
else
    $J^*_t \leftarrow J_t$
end if
}
```

5.3 Summary

In this chapter, we probed into complexity aspects of two kinds of RD-optimization algorithm, the Lagrangian multiplier (LM) algorithm and the algorithm for the incremental computation of the convex hull (ICCH). First, we defined implementation-independent complexity measures. We proceeded to estimate the complexity of RD-optimization algorithms using these measures. The comparison reveals that the ICCH-algorithm outperforms the LM-algorithm if the optimal LCH solution is sought. However, unlike the ICCH-algorithm, the LM-algorithm enables to find a tolerable solution fast. But even in this case, the ICCH-algorithm computes the optimal LCH solution faster. A speed up technique for the LM-algorithm was developed. It was shown that this new LM-algorithm determines the optimal LCH solution faster than the old LM-algorithm even if the old LM-algorithm computes only a tolerable solution.

Finally, run-time experiments showed that the ICCH-algorithm is still the fastest RD-optimization algorithm even though the implementation-independent measure suggests that the new LM-algorithm has the same run-time. This is due to the fact that the implementation-independent complexity measure of the ICCH-algorithm is, unlike the LM-algorithm, a worst case estimation.

Chapter 6

Summary and Conclusions

In this thesis, we inquired into two aspects of video coding. First, we dealt with *rate distortion (RD) optimization*. Secondly, we contributed experiments with a compression technique called *adaptive vector quantization (AVQ)*.

At the start of of Chapter 3 we described rate distortion optimization for independent continuous random variables. We stated a central problem of this theory, the *rate allocation problem*. We then presented the *discrete* rate allocation problem and showed similarities with the continuous case. Three types of dependences of the random variables can be determined, the *independent*, the *hierarchical dependent*, and the *general dependent* type. We perceived that the discrete rate allocation problem for the independent type can be optimally solved by a complex dynamic programming approach and be approximately solved by two fast algorithms, the Lagrangian multiplier (LM) algorithm and the incremental computation of the convex hull (ICCH) algorithm. For the second type of dependences, the hierarchical dependence, we presented the LM-algorithm which used dynamic programming to find the solution and the generalized BFOS algorithm for ICCH. The generalized BFOS, however, imposes restrictions on the hierarchical structure, and, thus, we contributed an extended generalized BFOS algorithm that solves the problem without restrictions. As it is not obvious that the extended algorithm should determine the same solution as the LM-algorithm, we proved the correctness of this new approach. Finally, general dependence is considered. This type of dependence can be solved with the LM-algorithm using dynamic programming. To the best of the author's knowledge, solutions applying other techniques are not found in the literature. To summarize, we gave a detailed review of the RD-optimization techniques and contributed a new technique in this chapter. In addition, we developed a new terminology that permits a concise and consistent description of the algorithms.

Chapter 4 described the experiments performed with *adaptive vector quantization (AVQ)* and video coding. First, we presented a codec that is able to encode and decode in real-time. The concept of mean-removed vector quantization was applied where the vector quantization was made by AVQ. We then carried out experiments with the codebook update strategy showing that the update strategy, if carefully designed, has little influence on the coding performance. It was demonstrated that the real-time codec can be improved significantly by more than 1 dB PSNR using a RD-optimization technique even though the RD-optimized codec is too complex to retain the real-time ability. Furthermore, we provided experiments with variable-length coding. With respect to our AVQ-codec, the adaptive arithmetic coder proved to perform best. Transform coding was considered next. We transfered the RD-optimized mean-removed AVQ scheme into the transform domain of a wavelet transform. Our experiments revealed that this led not only to a striking improvement of the AVQ update cost, as we had anticipated, but the pure vector quantization cost also yielded a better result. The overall improvement was more than 1 dB PSNR. This approach was refined by application of an adaptive partitioning technique, called *quad-trees* in the wavelet domain. We described how we found an appropriate partitioning structure. We used the RD-optimization for hierarchical dependences to optimize this structure. Again, this led to a significant performance gain over the previous codec. Comparison with standard transform coding, however, reveals that in spite of these improvements and apparent adaptability of our AVQ-codec it still shows a performance gap of about 1 dB PSNR. We conclude that this is due to the lack of motion compensation in our quad-tree based AVQ-scheme. Finally, we presented a study combining AVQ and motion compensation. Unlike the previous quad-tree codec, AVQ was combined with quad-tree coding in the spatial domain. We substituted the discrete cosine transform coding of a H.263 codec by AVQ. These experiments demonstrated that AVQ coding of the motion prediction error frame for low bitrates performs at least as well as DCT coding after an adaption period, if not better.

In Chapter 5 we used the AVQ-codec from Sect. 4.3 to analyze the complexity of the two RD-optimization techniques. We first defined implementation-independent complexity measures for the two algorithms. These measures were used to estimate the complexity of the LM- and the ICCH-algorithm. It was observed that the estimated complexity of the LM-algorithm is in the same order of magnitude as the estimated complexity of the computation of the sets of RD-points. What is more, the estimated complexity of the ICCH-algorithm is one-fourth of the estimated complexity of the LM-algorithm. Run-time experiments were provided which support the implementation-independent results. It was demonstrated that the characteristic of the specific RD-sets derived from AVQ is suitable for the ICCH-algorithm. Then we proposed an improvement of the LM-algorithm. We exploited a monotonicity property of the operational RD-function to exclude some RD-sets from the optimization procedure. This technique makes the LM-algorithm four times faster for both implementation-independent complexity and run-time experiments. With this technique RD-optimization can be performed mostly in three LM-iterations. Therefore, provided that the RD-sets have the same characteristics as in our experiments, it is not to be expected that techniques which reduce the number of LM-iterations would further speed up the improved LM-algorithm significantly.

Without doubt, the above results are dependent on the characteristic of RD-sets. In future research these complexity considerations should therefore be made for abstract sources of RD-points. In addition, the number of LM-iterations should be analyzed theoretically in order to find a tighter worst case complexity bound than the one presented in this thesis. What is more, the number of points on the lower convex hull of abstract RD-sets should be analyzed in order to find a lower bound for the complexity of the ICCH-algorithm.

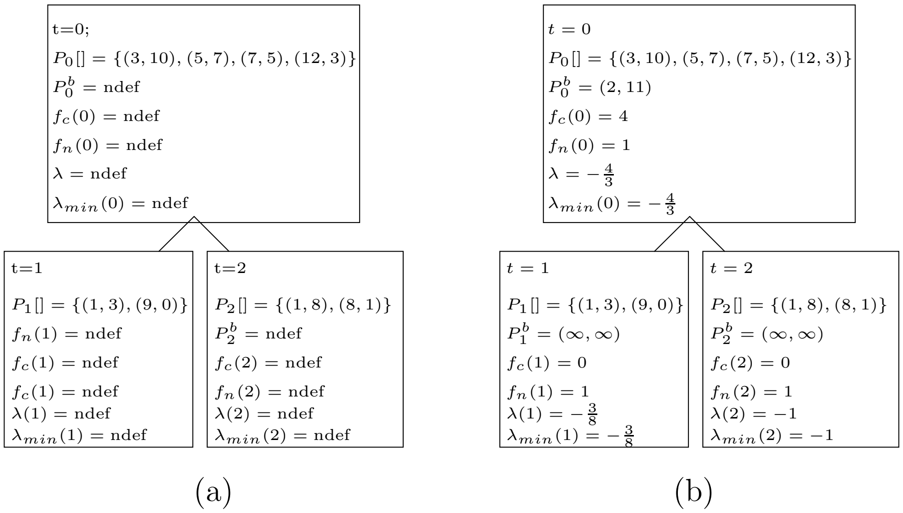
Further research is also desirable for the combination of AVQ and motion compensation. Even though the results presented in this thesis are promising, this concept needs a more thorough consideration. For example, the codebook update might be transmitted with transform coding, and AVQ could be applied for level-1 vectors, too. In addition, an investigation into the application of AVQ on DCT coefficients might improve the encoding results.

Appendix A

Example for the Proposed Algorithm

This section provides an example of the extended GBFOS algorithm presented in section 3.3.3. The tree structure of $\mathcal{T}$ is shown in Fig. A.1a. Node 0 has four RD-points and the additional option to be decomposed into node 1 and 2. Each of node 1 and 2 has 2 RD-points. Note that the branch point P_m^b is not contained in $P_m[]$ but appears separately. In addition, we have no decomposition cost, i.e., $(R_0^N, D_0^N) = (0, 0)$ with N=4.

At the beginning, values are undefined except the values referring to the tree structure, e.g. the local points $P_m[]$ and left(t) and right(t). The set, $\mathcal{P}_{\mathcal{T}}$, of all points representable by $\mathcal{T}$ is $\mathcal{P}_{\mathcal{T}} = \{(2, 11)$, $(3, 10)$, $(5, 7)$, $(7, 5)$, $(9, 4)$, $(10, 8)$, $(12, 3)$, $(17, 1)\}$. This set is depicted in Fig. A.2. The points $\{(2, 11), (5, 7), (7, 5), (9, 4), (17, 1)\}$ belong to the LCH. These points will be computed by our proposed algorithm. Figure A.3 shows the result if the procedure SEARCH is applied to $\mathcal{T}$.


t=0;
$P_0[] = \{(3,10),(5,7),(7,5),(12,3)\}$
$P_0^b = \text{ndef}$
$f_c(0) = \text{ndef}$
$f_n(0) = \text{ndef}$
$\lambda = \text{ndef}$
$\lambda_{min}(0) = \text{ndef}$
t=1
$P_1[] = \{(1,3),(9,0)\}$
$f_n(1) = \text{ndef}$
$f_c(1) = \text{ndef}$
$f_c(1) = \text{ndef}$
$\lambda(1) = \text{ndef}$
$\lambda_{min}(1) = \text{ndef}$
t=2
$P_2[] = \{(1,8),(8,1)\}$
$P_2^b = \text{ndef}$
$f_c(2) = \text{ndef}$
$f_n(2) = \text{ndef}$
$\lambda(2) = \text{ndef}$
$\lambda_{min}(2) = \text{ndef}$
$t = 0$
$P_0[] = \{(3,10),(5,7),(7,5),(12,3)\}$
$P_0^b = (2,11)$
$f_c(0) = 4$
$f_n(0) = 1$
$\lambda = -\frac{4}{3}$
$\lambda_{min}(0) = -\frac{4}{3}$
$t = 1$
$P_1[] = \{(1,3),(9,0)\}$
$P_1^b = (\infty,\infty)$
$f_c(1) = 0$
$f_n(1) = 1$
$\lambda(1) = -\frac{3}{8}$
$\lambda_{min}(1) = -\frac{3}{8}$
$t = 2$
$P_2[] = \{(1,8),(8,1)\}$
$P_2^b = (\infty,\infty)$
$f_c(2) = 0$
$f_n(2) = 1$
$\lambda(2) = -1$
$\lambda_{min}(2) = -1$

(a) (b)

Figure A.1: Example of the proposed algorithm, (a) before, (b) after initialization.

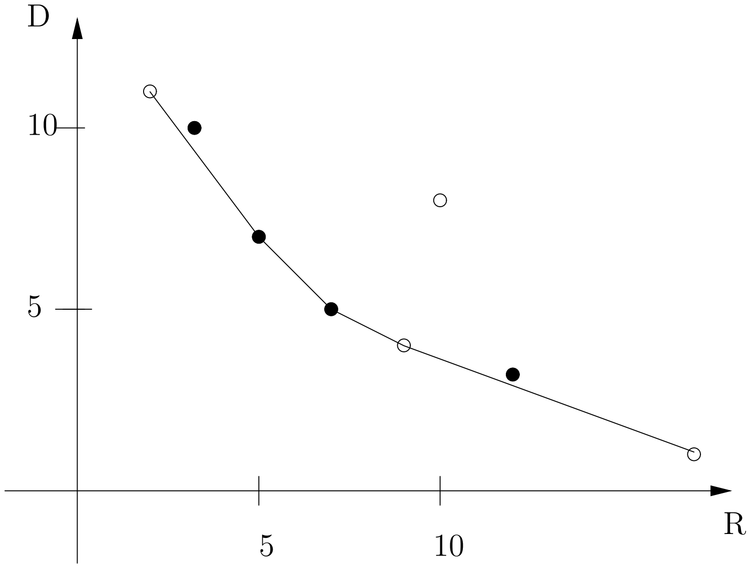


Figure A.2: Example of $\mathcal{P}_{\mathcal{T}}$. The points $\{(2,11),(5,7),(7,5),(9,4),(17,1)\}$ belong the the LCH.

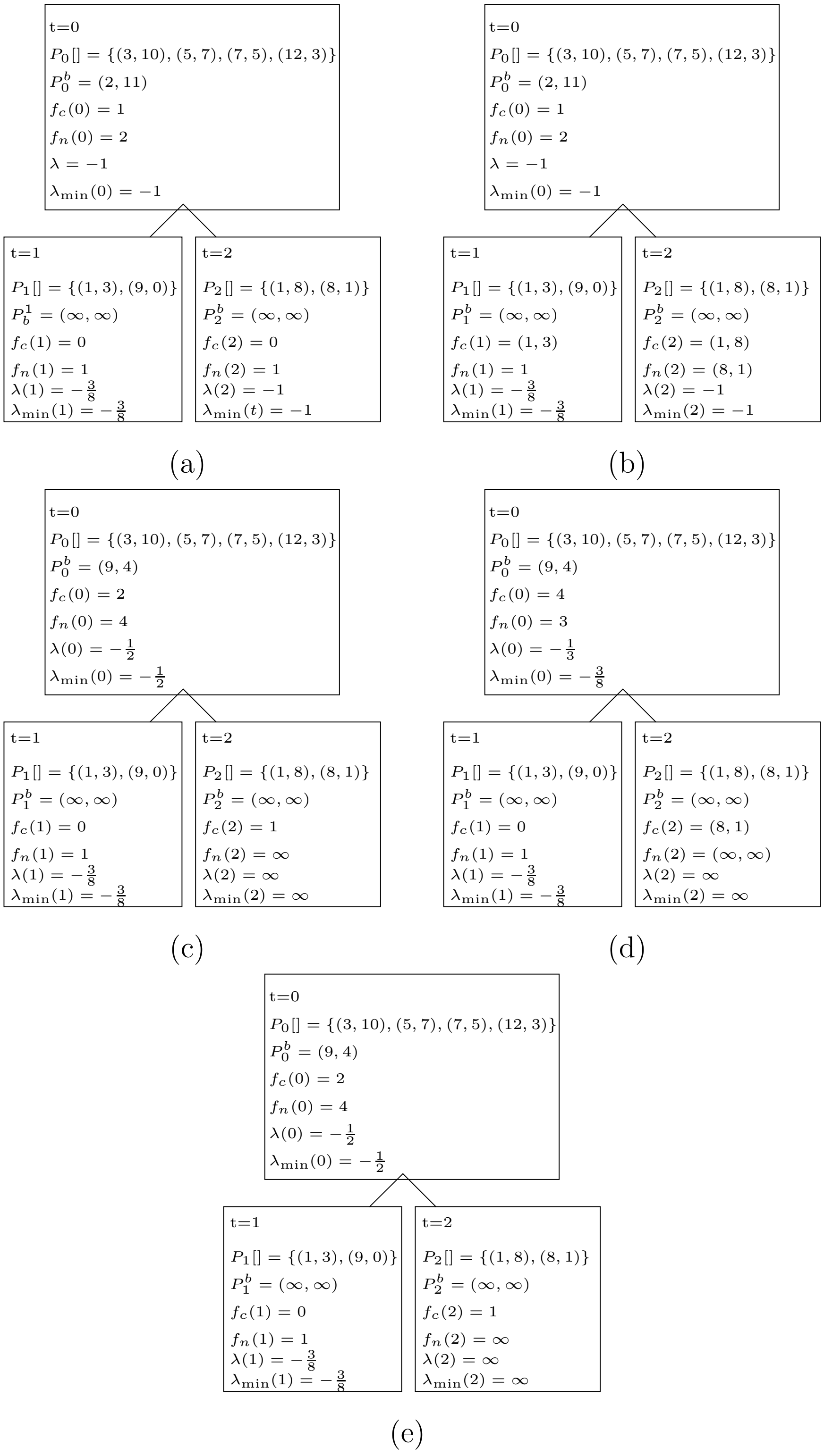

t=0
$P_0[] = \{(3,10),(5,7),(7,5),(12,3)\}$
$P_0^b = (2,11)$
$f_c(0) = 1$
$f_n(0) = 2$
$\lambda = -1$
$\lambda_{\min}(0) = -1$
t=1
$P_1[] = \{(1,3),(9,0)\}$
$P_b^1 = (\infty,\infty)$
$f_c(1) = 0$
$f_n(1) = 1$
$\lambda(1) = -\frac{3}{8}$
$\lambda_{\min}(1) = -\frac{3}{8}$
t=2
$P_2[] = \{(1,8),(8,1)\}$
$P_2^b = (\infty,\infty)$
$f_c(2) = 0$
$f_n(2) = 1$
$\lambda(2) = -1$
$\lambda_{\min}(t) = -1$
(a)
t=0
$P_0[] = \{(3,10),(5,7),(7,5),(12,3)\}$
$P_0^b = (2,11)$
$f_c(0) = 1$
$f_n(0) = 2$
$\lambda = -1$
$\lambda_{\min}(0) = -1$
t=1
$P_1[] = \{(1,3),(9,0)\}$
$P_1^b = (\infty,\infty)$
$f_c(1) = (1,3)$
$f_n(1) = 1$
$\lambda(1) = -\frac{3}{8}$
$\lambda_{\min}(1) = -\frac{3}{8}$
t=2
$P_2[] = \{(1,8),(8,1)\}$
$P_2^b = (\infty,\infty)$
$f_c(2) = (1,8)$
$f_n(2) = (8,1)$
$\lambda(2) = -1$
$\lambda_{\min}(2) = -1$
(b)
t=0
$P_0[] = \{(3,10),(5,7),(7,5),(12,3)\}$
$P_0^b = (9,4)$
$f_c(0) = 2$
$f_n(0) = 4$
$\lambda(0) = -\frac{1}{2}$
$\lambda_{\min}(0) = -\frac{1}{2}$
t=1
$P_1[] = \{(1,3),(9,0)\}$
$P_1^b = (\infty,\infty)$
$f_c(1) = 0$
$f_n(1) = 1$
$\lambda(1) = -\frac{3}{8}$
$\lambda_{\min}(1) = -\frac{3}{8}$
t=2
$P_2[] = \{(1,8),(8,1)\}$
$P_2^b = (\infty,\infty)$
$f_c(2) = 1$
$f_n(2) = \infty$
$\lambda(2) = \infty$
$\lambda_{\min}(2) = \infty$
(c)
t=0
$P_0[] = \{(3,10),(5,7),(7,5),(12,3)\}$
$P_0^b = (9,4)$
$f_c(0) = 4$
$f_n(0) = 3$
$\lambda(0) = -\frac{1}{3}$
$\lambda_{\min}(0) = -\frac{3}{8}$
t=1
$P_1[] = \{(1,3),(9,0)\}$
$P_1^b = (\infty,\infty)$
$f_c(1) = 0$
$f_n(1) = 1$
$\lambda(1) = -\frac{3}{8}$
$\lambda_{\min}(1) = -\frac{3}{8}$
t=2
$P_2[] = \{(1,8),(8,1)\}$
$P_2^b = (\infty,\infty)$
$f_c(2) = (8,1)$
$f_n(2) = (\infty,\infty)$
$\lambda(2) = \infty$
$\lambda_{\min}(2) = \infty$
(d)
t=0
$P_0[] = \{(3,10),(5,7),(7,5),(12,3)\}$
$P_0^b = (9,4)$
$f_c(0) = 2$
$f_n(0) = 4$
$\lambda(0) = -\frac{1}{2}$
$\lambda_{\min}(0) = -\frac{1}{2}$
t=1
$P_1[] = \{(1,3),(9,0)\}$
$P_1^b = (\infty,\infty)$
$f_c(1) = 0$
$f_n(1) = 1$
$\lambda(1) = -\frac{3}{8}$
$\lambda_{\min}(1) = -\frac{3}{8}$
t=2
$P_2[] = \{(1,8),(8,1)\}$
$P_2^b = (\infty,\infty)$
$f_c(2) = 1$
$f_n(2) = \infty$
$\lambda(2) = \infty$
$\lambda_{\min}(2) = \infty$
(e)

Figure A.3: Example of the application of the procedure SEARCH.

Appendix B

Additional Experiments

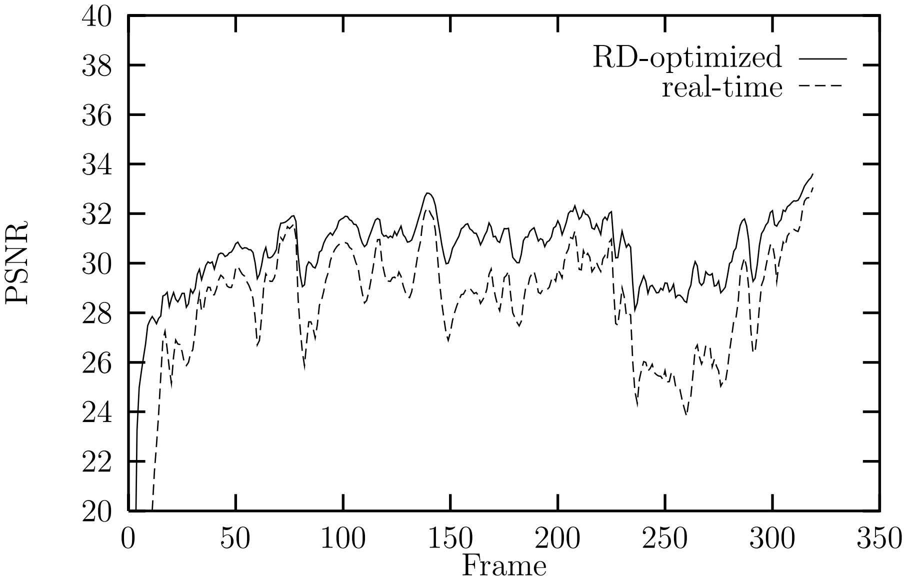


Figure B.1: PSNR course for the real-time codec compared with its RD-optimization for the sequence *Mother & Daughter* at bitrate 28000 bit/s.

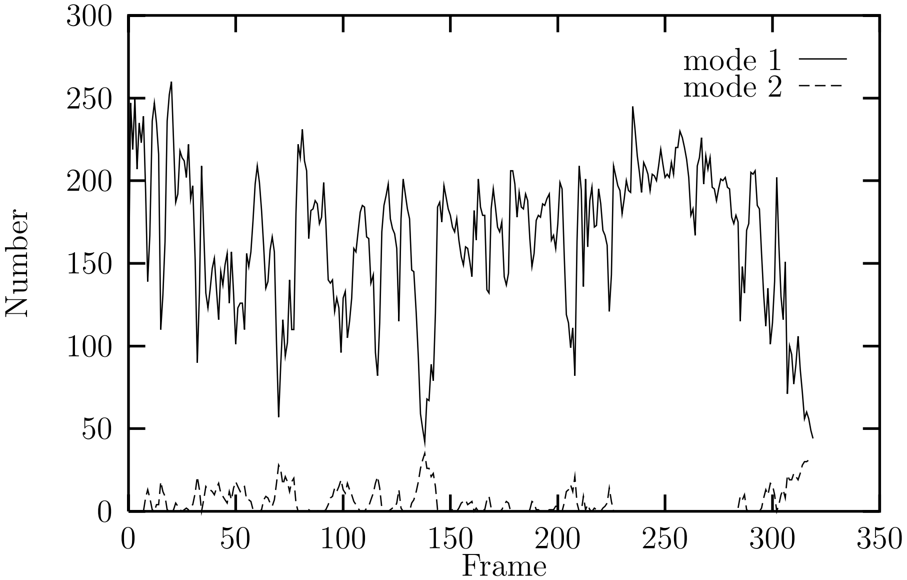


Figure B.2: Number of vectors per frame encoded in mode 1 or mode 2 for *Mother & Daughter* (RD-codec).

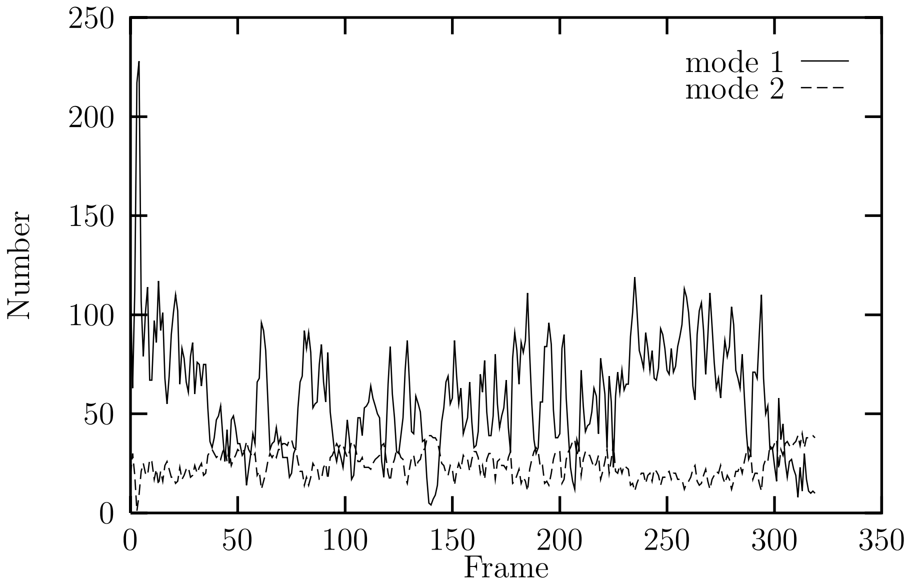


Figure B.3: Number of vectors per frame encoded in mode 1 or mode 2 for *Mother & Daughter* (real-time codec).

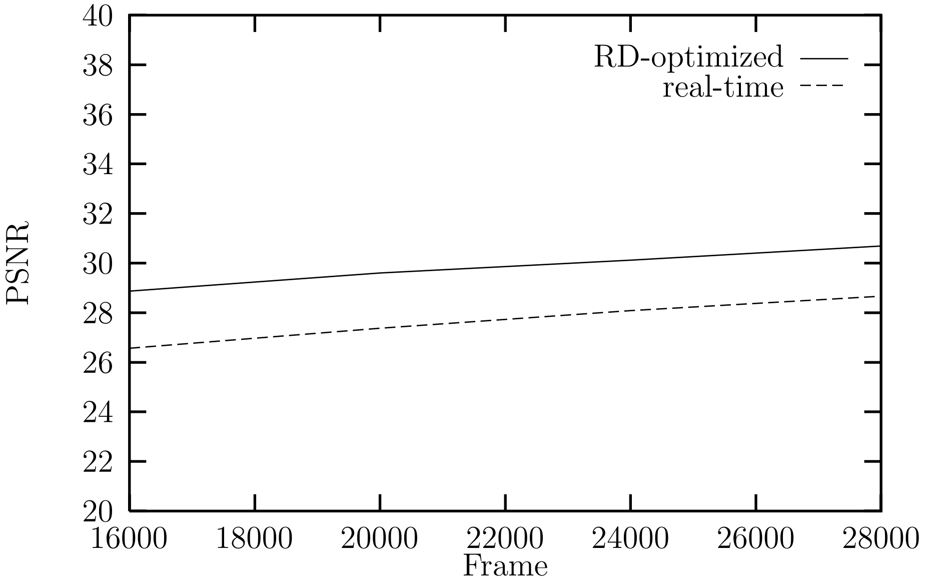


Figure B.4: Average PSNR values for several bitrates for the real-time codec compared with its RD-optimization for the sequence *Mother & Daughter* at bitrate 28000 bit/s.

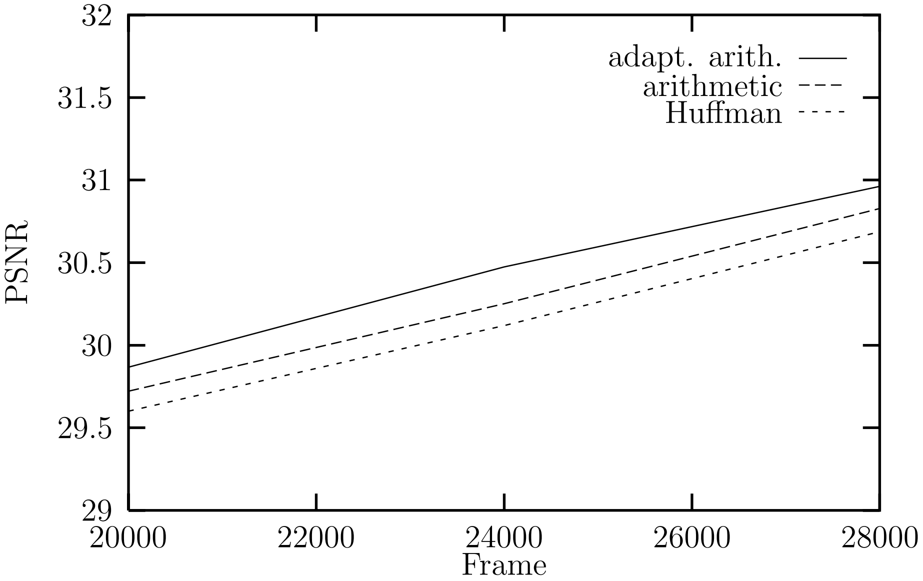


Figure B.5: Comparison of different VLC methods for *Mother & Daughter*.

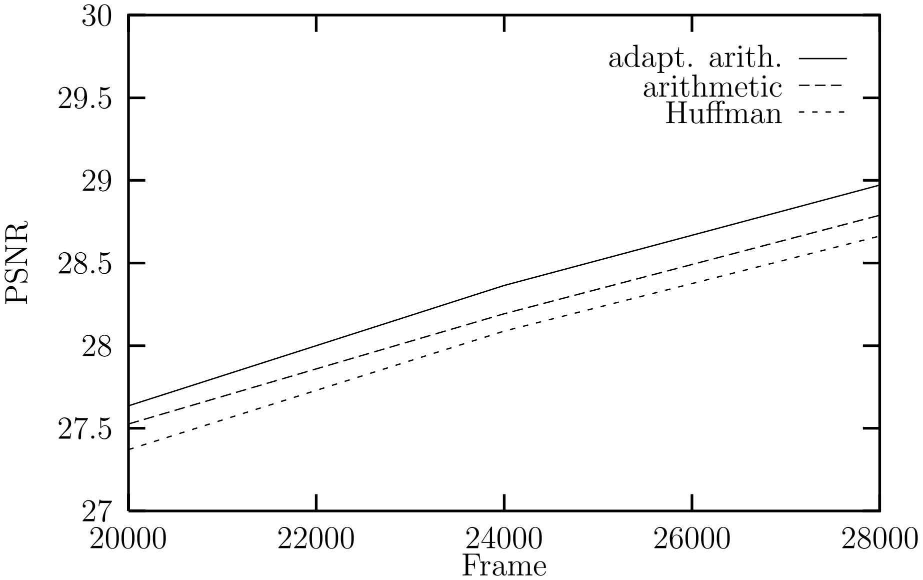


Figure B.6: Real time codec. Comparison of different VLC methods for *Mother & Daughter*.

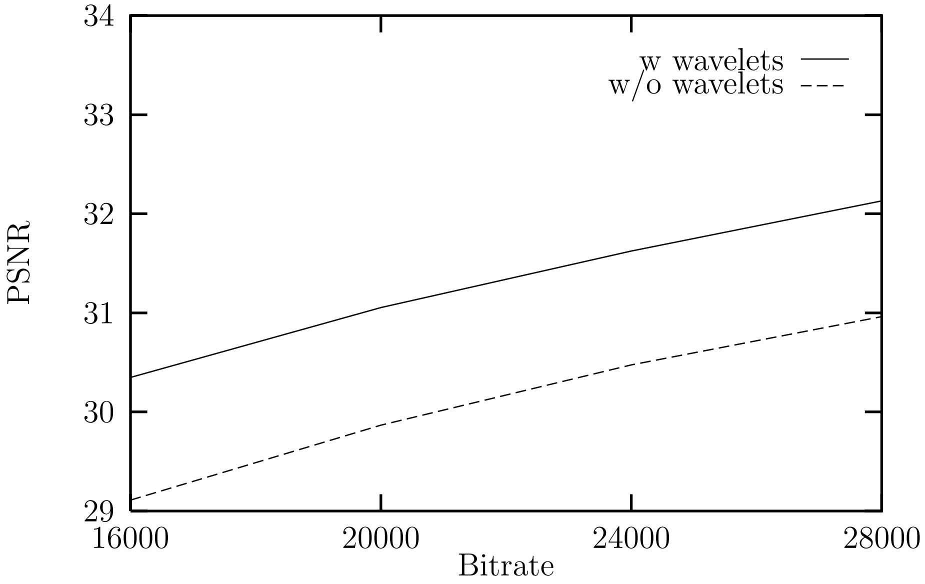


Figure B.7: Mean PSNR of several bitrates for *Mother & Daughter*.

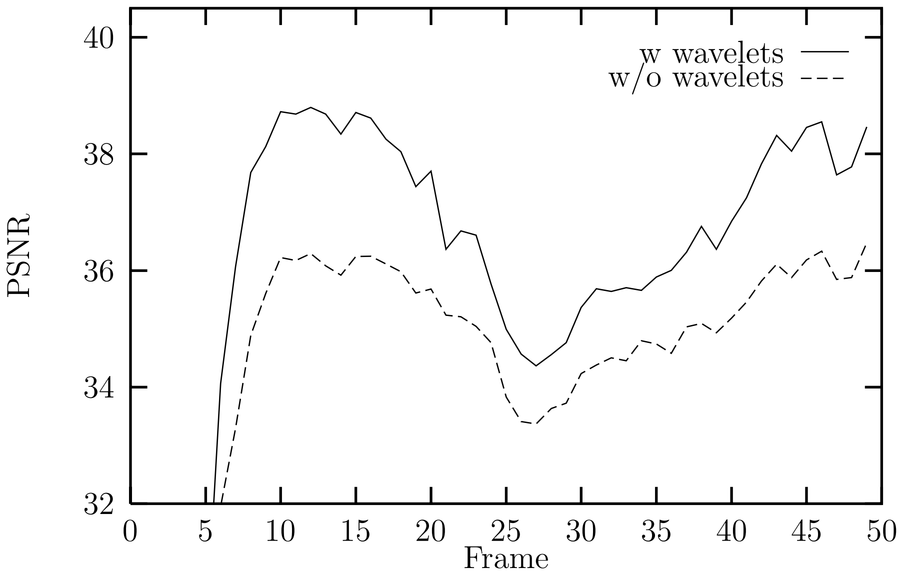


Figure B.8: PSNR course at a bitrate of 28000 bits/s for *Miss America*

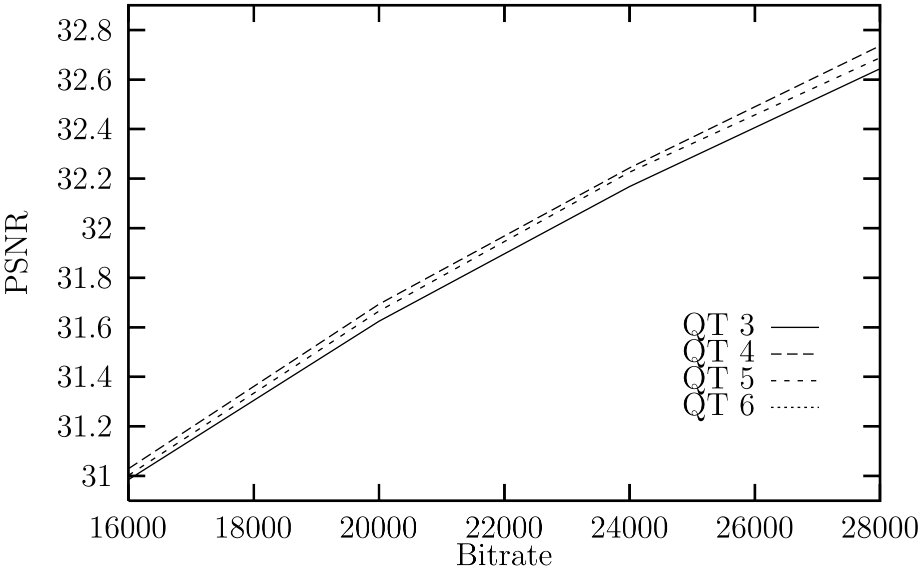


Figure B.9: Average PSNR values for the quad-tree experiments with *Mother & Daughter*.

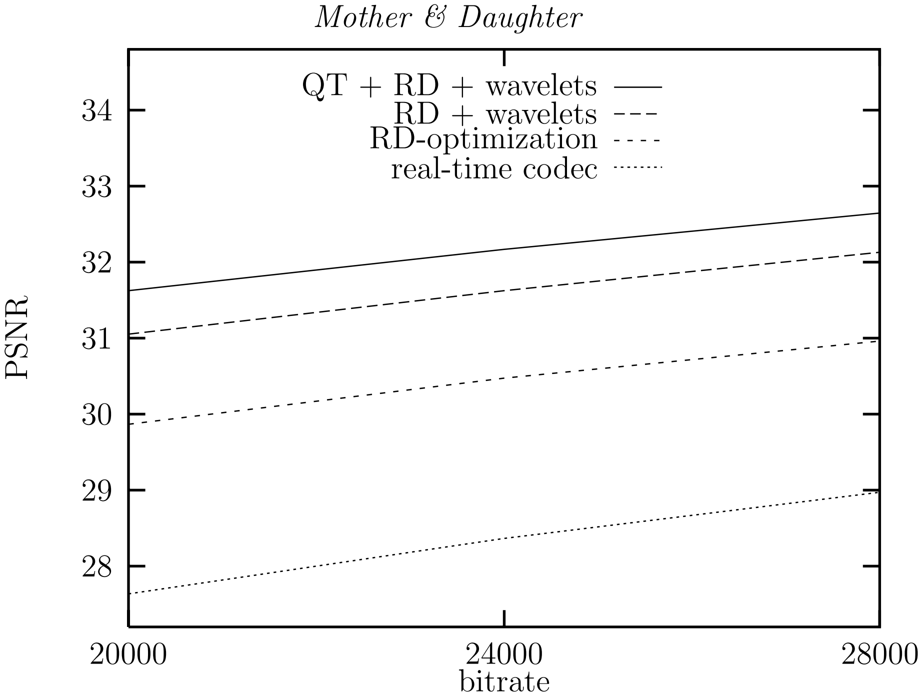


Figure B.10: Comparison of all AVQ-codecs.

Bibliography

[1] TMN 3.0 (H.263) codec. Released by the Signal Processing and Multimedia Group, University of British Columbia, `http://spmg.ece.ubc.ca`.

[2] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies. Image coding using wavelet transform. *IEEE Transactions on Image Processing*, 1(2):205–220, April 1992.

[3] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[4] T. Berger. *Rate Distortion Theory: A Mathematical Basis for Data Compression.* Prentice-Hall, Englewood Cliffs, NJ, 1971.

[5] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards.* Kluwer Academic Publishing, 1995.

[6] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Belmont, CA: Wadsworth, 1984.

[7] Bernd Butz. Echtzeitkodierung, -dekodierung, von Bildsequenzen mit adaptiver Vektorquantisierung. Master's thesis, Universität Freiburg, Insitut für Informatik, 1996.

[8] D.T.S. Chen. On two or more dimensional optimum quantizers. In *Proc. of ICASSP*, pages 640–643, 1977.

[9] Oscal T.-C. Chen, Bing J. Sheu, and Zhen Zhang. An adaptive vector quantizer based on the gold-washing method for image compression. *Trans. on Circuits and Systems for Video Technology*, 4(2):143–157, April 1994.

[10] P.A. Chou, T. Lookabaugh, and R.M. Gray. Entropy-contstrained vector quantization. *IEEE Trans. Accoustic, Speech and Signal Processing*, 37(1):31–42, January 1989.

[11] P.A. Chou, T. Lookabaugh, and R.M. Gray. Optimal pruning with applications to tree-structured source coding and modeling. *IEEE Trans. on Inform. Theory*, IT-35:299–315, March 1989.

[12] R.R. Coifman and M.V. Wickerhauser. Entropy-based algorithms for best basis selection. *Trans. Inf. Theory*, 38(2), March 1992.

[13] J.H. Conway and N.J.A. Sloane. Fast quantizing and decoding algorithm for lattice quantizers and codes. *IEEE Transactions on Information Theory*, 28:227–232, March 1982.

[14] P.C. Cosman, R.M. Gray, and M. Vetterli. Vector quantization of image subbands: A survey. *IEEE Transaction on Image Processing*, february 1996.

[15] Guy Côté, Berna Erol, Michael Gallant, and Faouzi Kossentini. H.263+: Video coding at low bit rates. *IEEE Transactions on Circ. and Syst. for Video Tech.*, 8(7), 1998 1998.

[16] T.M. Cover and J.A. Thomas. *Elements of Information Theory.* Toronto, Canada: J. Wiley & Sons, 1991.

[17] I. Daubechies. *Ten Lectures on Wavelets.* SIAM Publ., Philadelphia, 1992.

[18] T. Ebrahimi and C. Horne. "mpeg-4 natural video coding – an overview". *Signal Processing: Image Communication*, 15(4–5), Jan. 2000.

[19] W.H. Equitz. A new vector quantization clustering algorithm. *IEEE Trans. Acoust. Speech Signal Process.*, pages 1568–1575, October 1989.

[20] A. Moffat et al. Arithmetic coding source code, version 2, URL: http://www.cs.mu.oz.au/ alistair/arith_coder/, 1996.

[21] Markus Flierl, Thomas Wiegand, and Bernd Girod. A locally optimal design algorithm for block-based multi-hypothesis motion-compensated prediction. *Data Compression Conference, Snowbird, USA*, April 1998.

[22] J. E. Fowler. *Adaptive Vector Quantization for the Coding of Nonstationary Sources*. PhD thesis, Graduate School of The Ohio State University, 1996.

[23] J. E. Fowler. Generalized threshold replenishment: An adaptive vector quantization algorithm for the coding of nonstationary sources. *IEEE Transactions on Image Processing*, 7:1410–1424, Oct. 1998.

[24] J. E. Fowler and S. C. Ahalt. Adaptive vector quantization using generalized threshold replenishment. In *Proceedings of the 1997 IEEE Data Compression Conference*, 1997.

[25] M.R. Garey, D.S. Johnson, and H.S. Witsenhausen. The complexity of the generalized Lloyd-Max problem. *IEEE Trans. on Inf. Th.*, 28:255–256, march 1982.

[26] A. Gersho. On the structure of vector quantizers. *IEEE Trans. Inf. Th.*, IT-28:157–166, March 1982.

[27] A. Gersho and V. Cuperman. Vector quantization: A pattern-matching technique for speech coding. *IEEE Communications Magazine*, 21:15–21, December 1983.

[28] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.

[29] A. Gersho and B. Ramamurthi. Image coding using vector quantization. In *ICASSP'82*, volume 1, pages 428–431, May 1982.

[30] A. Gersho and M. Yano. Adaptive vector quantization by progressive codevector replacement. In *Proc. of ICASSP*, pages 4.6.1–4.6.4, 1985.

[31] A. Gersho and M. Yano. Adaptive vector quantization by progressive codevector replacement. In *Proc. Int. Conf. Acoustics, Speech and Signal Processing*, volume 1, Tampa, FL, Mar. 1986.

[32] M. Goldberg, P.R. Boucher, and S. Shlien. Image compression using adaptive vector quantization. *IEEE Trans. Commun.*, COMM-34(2):180–187, Feb. 1986.

[33] M. Goldberg and H. Sun. Image sequence coding using vector quantization. *IEEE Trans. Commun.*, COMM-34(7):703–710, July 1986.

[34] M. Goldberg and H. Sun. Frame adaptive vector quantization for image sequence coding. *IEEE Trans. Comm.*, 36:629–635, May 1988.

[35] ISO/IEC 13818 ITU-T Rec. H.262. Generic coding of moving pictures and associate audio information: Video, July 1995.

[36] R. Hamzaoui, D. Saupe, and M. Wagner. Rate-distortion based video coding with adaptive mean-removed vector quantization. In *Proc. of IEEE Intern. Conf. on Image Processing*, Chicago, USA, October 1998.

[37] Ralf Herz. Videokodierung mit adaptiver Vektorquantisierung und Wavelet-Transformation. Master's thesis, Universität Freiburg, Dec. 1998.

[38] J.J. Huang and P.M. Schultheiss. Block quantization of correlated gaussian random variables. *IEEE Trans. Commun. Sys.*, CS-11:289–296, Sept. 1963.

[39] D.A. Huffman. A method for the construction of minimum redundancy coding. *Proc. IRE*, 40:1098–1101, 1952.

[40] H. Everett III. Generalized Lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 11:399–417, 1963.

[41] ITU-T. Video codec for audiovisual services at $p \times 64$ kbit/s. ITU-T Recommendation H.261, March 1993.

[42] ITU-T. Video coding for low bitrate communication. ITU-T Recommendation H.263, Jan 1995.

[43] Jiro Katto, Jun ichi Ohki, Satoshi Nogaki, and Mutsumi Ohta. A wavelet codec with overlapped motion. *IEEE Trans. Circ. and syst. for video Technology*, 4(3):328–338, June 1994.

[44] T. Kohonen. *Self-Organization and Associative Memory.* Berlin: Springer-Verlag, 1984.

[45] R. Lancini, F. Perego, and S. Tubaro. Neural network approach for adaptive vector quantization of images. In *Proc. of ICASSP*, pages 389–392, March 1992.

[46] M. Lightstone and S. K. Mitra. Image-adaptive vector quantization in an entropy-constrained framework. *IEEE Transaction on Image Processing*, 6(3), March 1997.

[47] Michael Lightstone and Sanjit K. Mitra. Adaptive vector quantization for image coding in an entropy-constrained framework. In *Proc. of ICIP'94*, Austin, TX, November 1994.

[48] Y. Linde, A. Buzo, and R. M. Gray. An algorithm for vector quantizer design. *IEEE Trans. Commun.*, 28:84–95, Jan. 1980.

[49] S.P. Lloyd. Least squares quantization in PCM. Unpublished Bell Laboratories Technical Note. Published in IEEE Trans. on Inf. Th. 1982, 1957.

[50] S.P. Lloyd. Least squares quantization in PCM. *Trans. on Inf. Th.*, IT-28:127–135, March 1982.

[51] A.K. Louis, P. Maaß, and A. Rieder. *Wavelets: Theorie und Anwendungen.* B.G. Teubner Stuttgart, 1994.

[52] David G. Luenberger. *Linear and nonlinear programming.* Addison-Wesley Publishing Company, second edition, 1989.

[53] Udi Manber. *Introduction to algorithms.* Addison-Wesley, 1989.

[54] A. Moffat, R. Neal, and I.H. Witten. Arithmetic coding revisited. In *Proc. IEEE Data Compression Conference, Snowbird, Utah*, pages 202–211, March 1995.

[55] Mark Nelson and J.-L. Gailly. *The Data Compression Book*. M&T Books, 1996.

[56] D. L. Neuhoff. Why vector quantizers outperform scalar quantizers on stationary memoryless sources. In *Proc. 1995 IEEE Int'l Symp. on Information Thy.*, Whistler, B.C., Sept. 1995.

[57] M. Orchard and G. Sullivan. Overlapped block motion compensation: An estimation-theoretic approach. *IEEE Trans. Image Processing*, 3(5):693–699, September orchard.

[58] A. Ortega and K. Ramchandran. Forward-adaptive quantization with optimal overhead cost for image and video coding with applications to MPEG video coders. In *IS&T/SPIE, Digital Video Compression*, Feb. 1995.

[59] A. Ortega and K. Ramchandran. Rate-distortion methods for image and video compression. *Signal Processing Magazine*, 15(6), November 1998.

[60] A. Ortega, K. Ramchandran, and M. Vetterli. Optimal trellis-based buffered compression and fast approximations. *Trans. Image Proc.*, 3(1), Jan. 1994.

[61] T. Ottmann and P. Widmayer. *Algorithms and Data Structures*. Spektrum-Verlag, 2nd edition, 1993.

[62] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C – The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[63] K. Ramchandran and M. Vetterli. Best wavelet packet bases in a rate-distortion sense. *IEEE Trans. Image Processing*, 2(2):160–175, April 1993.

[64] K.R. Rao and J.J. Hwang. *Techniques & Standards for Image, Video & Audio Coding.* Prentice Hall, 1996.

[65] K.R. Rao and P. Yip. *Discrete Cosine transform.* Academic Press, New York, 1990.

[66] M.J. Sabin and R.M. Gray. Global convergence and empirical consistency of the generalized Lloyd algorithm. *IEEE Trans. Inform. Theory*, IT-32:148–155, March 1986.

[67] A. Said and W.A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Trans. on Circuits and Systems for Video Tech.*, 6(3):243–250, June 1996.

[68] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):188–216, June 1984.

[69] Hamed Sari-Sarraf and Dragana Brzakovic. A shift-invariant discrete wavelet transform. *IEEE Transactions on Signal Processing*, 45(10):2621–2626, October 1997.

[70] D. Saupe and B. Butz. Real-time very low bit rate video coding with adaptive mean-removed vector quantization. In *Proc. of IEEE Intern. Conf. on Image Processing*, Santa Barbara, 1997.

[71] G. M. Schuster and A. K. Katsaggelos. *Rate-Distortion Based Video Compression.* Kluwer Academic Publishers, 1997.

[72] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[73] C.E. Shannon. Coding theorems for a discrete source with a fidelity criterion. *IRE National Convention Record*, pages 142–163, 1959.

[74] Jerome M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993.

[75] Y. Shoham and A. Gersho. Efficient bit allocation for an arbitrary set of quantizers. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 36(9):1445–1453, September 1988.

[76] Eli Shusterman and Meir Feder. Image compression via improved quadtree decomposition algorithms. *IEEE Trans. on Image Processing*, 3(2):207–215, March 1994.

[77] E. P. Simoncelli, W.T. Freeman, E.H. Adelson, and D.J. Heeger. Shiftable multiscale tranforms. *Trans. Inf. Theory*, 38(2), March 1992.

[78] P. Strobach. Tree-structured scene adaptive coder. *IEEE Trans. Commun.*, 38(4):477–486, Apr. 1990.

[79] P. Strobach. Quadtree-structured recursive plane decomposition coding of images. *IEEE Trans. Signal Processing*, 39(4):1380–1397, June 1991.

[80] G. J. Sullivan. *Low-Rate Coding of Moving Images Using Motion Compensation, Vector Quantizatin and Quadtree Decomposition.* PhD thesis, University of California, Los Angeles, 1991.

[81] G. J. Sullivan. Multi-hypothesis motion compensation for low bit-rate video coding. In *Proc. ICASSP'93*, volume 5, Apr. 1993.

[82] G. J. Sullivan and R. L. Baker. Efficient quadtree coding of images and video. *Trans. image proc.*, 3(3):327–331, May 1994.

[83] G.J. Sullivan and T. Wiegand. Rate-distortion optimization for video compression. *IEEE Signal Processing Magazine*, pages 74–90, 1998.

[84] M. Vetterli and J. Kovacevic. *Wavelets and subband coding.* Prentice Hall, 1995.

[85] John D. Villasenor, Benjamin Belzer, and Judy Liao. Wavelet filter evaluation for image compression. *IEEE Transactions on Image Processing*, August 1995.

[86] M. Wagner, R. Herz, H. Hartenstein, R. Hamzaoui, and D. Saupe. A video codec based on R/D-optimized adaptive vector quantization. In *Proc. of Data Compression Conference, Snowbird, Utah*, page 556, 1999. Full paper available as Technical Report 119, Institut für Informatik, Universität Freiburg.

[87] M. Wagner and D. Saupe. RD-optimization of hierarchical structured adaptive vector quantization for video coding. In *Proc. of Data Compression Conference, Snowbird, Utah*, page 576, 2000. Full paper available as Technical Report 139, Institut für Informatik, Universität Freiburg.

[88] M. Wagner and D. Saupe. Video coding with quad-trees and adaptive vector quantization. In *Proc. of European Signal Processing Conference*, 2000.

[89] X. Wang, S. Shende, and K. Sayood. Online compression of video sequences using adaptive VQ codebooks. In *Proc. DCC'94 Data Compression Conference*, Snowbird, Utah, March 1994.

[90] P.H. Westerink, J. Biemond, and D.E. Boekee. An optimal bit allocation algorithm for subband coding. In *Proc. ICASSP'88*, pages 757–760, 1988.

[91] P.H. Westerink, D.E. Boekee, J. Biemond, and J.W. Woods. Subband coding of images using vector quantization. *IEEE Trans. Commun.*, 36:713–719, 1988.

[92] Thomas Wiegand, Michael Lightstone, and Debargha Mukherjee et al. Rate-distortion optimized mode selection for very low bit rate video coding and the emerging H.263 standard. *IEEE Transactions on Circuits and Systems for Video Technology*, December 1995.

[93] Thomas Wiegand, Xiaozheng Zhang, and Bernd Girod. Motion compensating long-term memory prediction. In *Proceedings of ICIP*, October 1997.

[94] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6), june 1987.

[95] Zhan Zhang and Victor K. Wei. An on-line universal lossy data compression algorithm via continuous codebook refinement–Part I: Basic results. *IEEE Transactions on Information Theory*, 42(3), may 1996.