Institut für Informatik
der Technischen Universität München

A Compression Engine for Multidimensional Array Database Systems

Andreas Dehmel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Bernd Radig
Prüfer der Dissertation:	
1.	Univ.-Prof. Rudolf Bayer, Ph.D.
2.	Univ.-Prof. Dr. Thomas Huckle

Die Dissertation wurde am 11.12.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.04.2002 angenommen.

Abstract:

Multidimensional array data can be found in many areas, ranging from classic applications like digital image and video data, over spatial and spatio-temporal fields commonly used in medical research, scientific computing and numerical simulation, to abstract high-dimensional data cubes in e.g. OLAP. While traditionally all these data types have been treated as entirely different entities with specialized, often file-based processing tools, the multidimensional array DBMS RasDaMan unites them under a common framework and provides typical database services like a transaction concept and a query language. One thing most multidimensional arrays have in common and which clearly sets them apart from objects typically found in relational DBMSs is their large data volume; this property makes compression very attractive to reduce storage requirements and IO times, and it also allows compression to work more efficiently than for small data units.

The topic of this thesis is the design and implementation of a compression engine for generic multidimensional array data, suitable for integration into the RasDaMan database kernel. There are no standard compression techniques for multidimensional arrays except for bytestream-oriented approaches, which however fail to exploit properties such as correlations between spatially neighbouring cells or across channels, therefore new or generalized algorithms are needed to improve compression rates. Like raster images, many types of multidimensional arrays have a spatial or spatio-temporal interpretation, so image compression techniques can be used as a design template for parts of the compression engine, leading to a standard two-layer architecture with a model layer which transforms the data to exploit correlations and a compression layer which compresses the transformed data using existing standard techniques. The resulting system has an easily extendable modular design and unites compression techniques with different levels of complexity under a common interface for lossless as well as lossy, and storage- as well as transfer compression.

The main part of this thesis deals with the development of different model layers for the compression of multidimensional arrays, including various kinds of predictors and a generic wavelet engine (mainly) for lossy compression with arbitrary quality levels. The work closes with an in-depth performance evaluation of the compression engine's major components on a selection of different test data with 2 to 4 dimensions from application areas including medicine and scientific computing, which proves that the developed model layers can help improve compression rates considerably and that lightweight compression can also improve total system performance by reducing transfer and IO cost.

Acknowledgements

Writing a thesis is a *long* process and while some aspects of the work are rather tedious it is also a tremendously satisfying experience to see it develop from the first rough draft to the consistent whole you see before you. The environment a thesis is written in plays a major role in how successful the effort will turn out to be, therefore I'd like to take the time to thank everybody who helped me directly or indirectly to finish this work, apologizing in advance to anyone I might have forgotten.

First I'd like to thank those who've known me the longest and strangely enough still put up with me, my parents Wilfried and Elfriede Dehmel, for continued support over the years at university and never pushing me one way or another; also for putting up with the occasional loud rock music and the sound of my concert guitar at 3am, of course.

Next up I'd like to thank the people I worked with at the database faculty of the Technical University of Munich while the thesis took shape, in particular my colleagues at FORWISS (past and present) who provided a wonderfully informal and supportive working environment I will truly miss. Most of all I'd like to thank the other RasDaMan developers at FORWISS who I naturally worked most closely with: those who came before me, Paula Furtado, Roland Ritsch and Norbert Widmann, as well as our youngsters Karl Hahn and Bernd Reiner who will follow in our footsteps one day. Working in this team was a real pleasure. A very warm thanks also goes to Peter Baumann, the man who gave birth to the entire RasDaMan idea and left FORWISS several years ago to turn it into a commercial product with his company *Active Knowledge*.

I'd also like to thank my friend Michael Bader for proofreading this thesis and his valuable input on document structure and fine-tuning of the layout. Furthermore I want to thank my doctoral thesis supervisor Prof. Rudolf Bayer for supporting this thesis and his sharp eye when it came to inconsistent equations; his input also proved very helpful in ironing out the remaining ambiguities and loose ends.

In closing I'd also like to thank the Open Source community for their efforts in providing free and most of all open standards compliant software, which allowed me to produce every single part of this thesis on the platforms of my choice in a straightforward way. Foremost of all I'd like to thank Donald E. Knuth in this context, whose work on TEX laid the foundation for what is still the only truly professional document processing system, which allowed me to concentrate on the important issues rather than being delayed by tedious footwork.

Contents

Chapter 1

Introduction

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say.

J.R.R. Tolkien, *The Lord of the Rings*

1.1 Motivation

Integrating compression in Database Management Systems (DBMS) has not received very much attention so far. The main reason for this is that the benefits of compression in traditional, relational DBMSs (RDBMS) are rather limited due to the nature of relational data which typically consists of small units like numbers or short strings, which have a size of several bytes rather than kilo- or even megabytes. Compressing these small units individually results in very poor space savings, whereas grouping units into compounds and then compressing them improves the space savings per unit considerably, but burdens the system with much higher random access times, as will be shown below. This is often unacceptable since the resulting system is usually expected not to be noticeably slower despite the compression/decompression overhead, but possibly even faster due to reduced data size which allows more information to be transfered in fewer IO operations.

Compression is always a tradeoff between savings in storage space and compression overhead, where on average complex compression algorithms save more space while taking longer, whereas simpler algorithms compress worse but are faster; it is therefore obvious that in order to minimize compression overhead, simple algorithms should be chosen. We will also see that fine access granularity, as is typically required for RDBMSs, is another point in favour of simple compression variants as well. In contrast to RDBMSs, array DBMSs (ADBMS) deal with much larger units which makes compression considerably

more efficient as well as more attractive for the space savings alone; it also encourages the use of more advanced compression techniques, even lossy ones which have never been an option in RDBMSs, in the same way that lossy compression is viable for images but not for text.

Compression aims to minimize the redundancy in data by finding a representation for the data that requires less storage. This is often an adaptive process that performs better the more data it processes, i.e. a large text compressed as a whole will usually have a higher compression ratio than any subset of the text, depending on the compression algorithm used. Analogously, while all data contained in a relational table may compress well in its entirety, the individual attribute values are typically too short to allow gathering any statistically meaningful information to aid compression. The techniques used for the compression of single attribute values are therefore very basic since more complex approaches like adaptive arithmetic coding can't accumulate enough statistical data to outperform the simpler and therefore faster techniques.

The alternative of grouping together attribute values to allow more efficient compression severely compromises execution time because accessing any value contained in such a compound structure requires uncompressing the whole, or at least all data up to the desired value, since compressed data normally does not allow random access any more. Moreover, updating data in a compressed format is not localized, i.e. changing a local part in the uncompressed data usually implies global changes of the compressed data, typically from the point corresponding to the start of the update of the uncompressed data to the end of the compressed data stream. Caching uncompressed compounds can alleviate this phenomenon in some cases, but introduces other problems like an increase of memory requirements as well as more complex IO and transaction management. Considering these points it comes as no surprise that there has been little research on compression in traditional DBMSs and the work done on the subject has focussed on relatively simple algorithms, some of which will be introduced in the *Related Work* section on page 10.

Not all DBMSs are relational, nor do all systems require as fine an access granularity, however. Array DBMSs deal with rastered data of varied dimensionality and differ considerably from traditional DBMSs both in terms of access granularity and data volume typically transfered to clients. Whereas the access granularity of RDBMs is in the area of bytes, for array DBMSs it is in the area of kilobytes or even megabytes, which makes these systems very interesting for compressed storage. The high data volume transferred to client applications also makes transfer compression a viable approach. Moreover, for multidimensional data there are often local correlations between data samples which can be exploited by compression, like for instance a rectangular area of uniform colour in an image which will compress better if the compression algorithm is aware of the 2D nature of the source data rather than being applied to a sequence of 1D data sequences. Things like this are usually resolved in a *model* (or transformation) layer which transforms the original data according to a data model into a different (possibly similar, i.e. lossy) representation which compresses better.

RDBMSs can only handle vector data efficiently, which can be translated into tuples containing coordinate attributes and value attributes. However, many kinds of data can be

modelled efficiently as multidimensional arrays, especially dense data, of course; but with ever increasing memory capacity, even modelling sparse data as multidimensional arrays becomes feasible, especially with the addition of compression, since sparse representation in the form of vectorized data is a kind of offset compression in itself. In [44], for instance, this development is quoted in the context of visualization to predict a move away from surface-oriented models (vector) to volume-oriented models (arrays) in a similar way as raster images have grown ever more popular compared to vector graphics in many cases, especially when it comes to digital representations of real world phenomena, typical examples of which are sampled analogue data such as images, volumetric data like tomograms, or simulation data (spatial or spatio-temporal, like for instance fluid flow or climate simulations). The disadvantages of dense modelling, foremost of all memory requirements, are becoming less relevant as more memory is readily available; in many cases the vector data used in e.g. visualization has to be calculated from a dense representation like a 3D data cube anyway (e.g. isosurface algorithms [34] which calculate a triangulated surface from 3D array data). At the same time the advantages of dense data modelling are becoming more and more attractive in many application areas:

- compact storage (at least for dense data), i.e. no coordinate overhead because the coordinates are implicitly given by the offset in the data and the linearization scheme;
- constant access time of arbitrary coordinates independently of the data distribution, which also includes finding neighbouring cells in constant time.

Returning to the example of visualization, volume rendering has considerable advantages compared to surface-oriented (= vector-oriented) approaches as it allows exploring internal structures [44]. In numerical simulation, using arrays rather than vectors for storage allows simpler and more efficient algorithms for the solution of partial differential equation systems. Using hierarchical grids makes it possible to use more sample points within areas of rapid change than in areas with little activity, thereby addressing the major problem of dense modelling; wavelet-based compression implicitly uses a similar approach, as will be shown later in section 3.3.

Naturally, dense modelling can't replace vector data entirely, especially when the coordinate system is not discrete but real-valued as in CAD systems or to a certain extent documents, although thanks to specialized compression techniques documents are actually becoming a borderline case [8]; it must be added, however, that rasterized documents are mostly of interest to digital libraries where the vector data is not available or never existed in the first place. Looking especially at the developments in visualization and numerical simulation, there is a clear trend away from vector data towards array data.

The object of this work was the implementation of a compression engine for the multidimensional Array DBMS $\mathcal{R}as\mathcal{D}a\mathcal{M}an$, which was originally developed at FORWISS and supports data of arbitrary dimensionality and base type. Data is modelled as multidimensional arrays, i.e. dense storage; the addition of compression capabilities to the DBMS also allows sparse data to be handled efficiently, however. There is a wealth of literature on

specialized compression, especially the compression of raster images, i.e. 2D arrays over a small number of possible base types, but there is scarcely any work on integrating these techniques into DBMSs or similar products dealing with multidimensional arrays. Furthermore, the approaches are usually restricted to a specific number of dimensions – foremost of all 2D for images – which requires generalization before they can be integrated into a truly *multidimensional* system. The purpose of this thesis is

1. the design of a generic compression framework for storage- and transfer-compression and its integration into the kernel of a multidimensional Array DBMS, $\mathcal{R}as\mathcal{D}a\mathcal{M}an$ in this case;

2. the evaluation of compression classes and the inclusion of promising candidates in the compression framework. The main focus here lies on techniques developed in image compression, because many MDD show exactly the same properties exploited in image compression, foremost of all local smoothness caused by correlations between neighbouring cells;

3. provide compression algorithms with different properties dependening on the application. For transfer compression, (de)compression overhead is usually the decisive factor, whereas for long-term storage it is the compression ratio. For mostly read-only data an algorithm with asymmetric complexity may be ideal, such as (adaptive) dictionary techniques where compression can take considerably longer than decompression.

4. performance measurements on the resulting system and evaluation of the various compression classes with respect to the data types they are applied to and the scenarios they are used in (for instance storage compression vs. transfer compression).

The goal in integrating a compression engine into the DBMS kernel is to reduce storage requirements on one hand as well as transfer times on the other. As always in compression, the trade-off between storage reduction and compression overhead plays a central role and will be evaluated in more depth in sections 3.7 and 4.4.

1.2 Compression Basics

The fundamental idea of data compression is to find shorter – and in case of lossy compression approximate – representations for given data (= a sequence of symbols). Without compression, the number of bits required for a symbol is determined by the data type of the symbol, where typically only a few different types are supported by computer systems, regardless of the actual symbol values. While this property allows very fast data access, it usually takes up more storage than strictly required, for instance when only the 26 lower case characters appear in a data stream but 8 bits have to be used for storage.

In lossless compression there is a hard limit on achievable compression, determined by the so called *entropy* of the data. This part of information theory was introduced by

Shannon [48] in 1948 where the entropy H of a sequence S of independent, identically distributed symbols over an alphabet $\mathcal{A} = \{X_1, \ldots, X_m\}$ with probabilities $P(X_i)$ was defined as

$$H(S) = -\sum_{i=0}^{m} P(X_i) \log P(X_i). \tag{1.1}$$

The entropy represents the average amount of storage space per symbol (= *rate*) required to encode S losslessly. The unit of the storage space depends on the base of the logarithm function in equation (1.1); typically $\log_2$ is used, in which case the entropy is the average number of bits per symbol. Shannon proved that it is impossible for *any* compression algorithm to encode a given symbol sequence in fewer bits than specified by the entropy, provided the assumption about the independence of the symbols holds.

In many cases, the symbols are dependent, however. A typical example of this is of course text, where preceding symbols severely restrict the possible values of following symbols, for example the probability of a vowel following a "th" in an english text is considerably higher than that of another consonant. Symbols are often correlated in other cases as well, such as in "smooth" signals where the next symbol only differs from the preceding one by a small amount which can be coded in fewer bits than the actual symbol value. These dependencies are usually resolved via a *data model*, for instance "english text" or "smooth values" or "values lying approximately on a straight line" etc. The better a model matches a symbol sequence the better this sequence can be compressed, even well below the entropy based on the assumption of independent symbols. Therefore most modern compression techniques consist of two layers, a top layer which transforms the data according to a data model, and a bottom layer which actually compresses the transformed data; this architecture will be discussed in more depth in chapter 3.

There is a very limited number of traditional techniques in the bottom layer and even modern compression techniques rely on one of these for actual data compression. In contrast, there is a large number of data models in the top layer and most "new" compression techniques introduce new algorithms in the top layer only, this thesis being no exception. Traditional data compression techniques can be divided into four elementary techniques and two basic classes which may also be combined to improve the compression ratio (and often are so); I will only give a short overview here since they will be covered in much more depth in chapter 3.

Pattern-oriented techniques:

RLE: (*Run-Length Encoding*) compresses consecutive symbols of the same value. Very low complexity, but also relatively poor compression ratio for dense data;

Dictionary Techniques: find patterns in the symbol sequence and replace these literal patterns with references into a dictionary. Very high complexity during compression, but usually a good compression rate;

Variable-length coding:

Huffmann Coding: represents symbols with high probability with fewer bits than those with lower probability, thereby achieving data reduction. Because every symbol must be represented by an integral number of bits, only symbol probabilities which are (negative) powers of 2 can be modelled exactly;

Arithmetic Coding: the current state-of-the-art in variable length coding which recently superceded Huffmann coding as the de-facto standard. Arithmetic coding represents a symbol sequence of arbitrary length by a number in the unit interval $[0, 1[$ with arbitrary precision. By representing symbol probabilities as subintervals of the unit interval with a width proportional to their probability, arbitrary symbol probabilities can be modelled exactly;

An important property of these compression techniques is whether they are static or whether they can adapt to the data they are applied to. For example a dictionary technique could use a static dictionary or build the dictionary during operation; or the variable-length coding techniques may need to know the probability distribution in advance or they may adapt the probabilities according to the data automatically. In some cases, like text in a natural language, static variants are usually sufficient or actually better than adaptive ones because the sub-optimal "learning" phase of the adaptive process can be disposed of and the dictionary doesn't have to be stored with the compressed data. But in the majority of cases – especially when compressing binary data – adaptivity is an important criterion because no generally applicable dictionary or probability distribution exists. Regarding adaptivity, all dictionary techniques that are in common use today are adaptive. Huffman coding is very hard to use adaptively because the Huffman tree has to be fully materialized (and therefore constantly recalculated for adaptivity), making adaptivity very expensive. Arithmetic coding can be made adaptive with much less overhead, which is another substantial advantage over Huffman coding: even one of the first implementations of arithmetic coding was adaptive [63].

Typically, compression in the bottom layer is a combination of a pattern-oriented technique followed by variable-length coding, e.g. for a compact representation of references in dictionary coding like in the *ZLib* compression library [67] where dictionary offsets and lengths are Huffmann-coded.

1.3 Related Work

There are not many publications on the subject of compression in databases and the existing ones are based on mostly text-based databases like RDBMSs. As noted in section 1.1, the situation regarding compression in traditional, text based DBMSs differs considerably from that in an Array DBMS, where typically large blocks of contiguous data are processed and transferred in one go rather than single cell values. However, the basic goals of using compression in a DBMS are the same for all kinds: reduction of storage and in addition a potential IO speedup due to the reduced data volume. I will therefore discuss some of

the literature on compression in RDBMSs here. It must be noted that the older the literature gets the less relevant its conclusions usually are from today's perspective because what used to be an expensive compression technique at that time may well be considered light-weight compression today.

The newest of the publications introduced here is [16] and is actually more about database normalization than compression in the classical sense. The idea introduced there is to reduce the data volume by adding an indirection level for attributes which can have a limited number of different values, i.e. rather than encode the attribute value inside the table, the attribute becomes a reference into an additional dictionary table. The example given there is a table encoding computer chip specifications where one field of characters can take on the values `CMOS` and `TTL` only. These can be "compressed" by using a dictionary table containing these two possible values and transforming the original table by storing the offset into this dictionary table in place of the attribute value, which can be done in just 1 bit in this example. While some of the most widely used compression algorithms like LZ77 [64] and LZ78 [65] are based on the idea of a dictionary, it still seems odd to categorize transformations like the above as compression rather than database normalization. There is common ground between schema design and compression in that good schema design strives to minimize redundancy (at least when ignoring preaggregation), just like compression; nonetheless what this paper actually describes is much closer to schema design or even plain programming style than to (database) compression.

A paper covering actual compression in a RDBMS is [62] which introduces several compression techniques, discusses the ones chosen in more depth and concludes with measurements after integrating the compression engine into their AODB system. Low compression overhead and fine access granularity are stressed as points of primary importance from the beginning. Compressed tuples are divided into five fields with different properties, namely

1. values of fields compressed to constant length (no address calculations), e.g. using dictionary techniques with a known dictionary size;

2. lengths of all fields compressed to variable length (note that the length of this section is also of constant length);

3. values of uncompressed fields of constant length (this section is of constant length too);

4. values of fields compressed to variable length;

5. string values of `VARCHAR` fields; `CHAR` fields of fixed (maximum) length are converted to `VARCHAR` fields to ensure only the data actually required for the string is stored.

This separation aims to allow constant random access time by putting all fields of constant size in a block at the beginning. The actual compression techniques used are rather simple, e.g. integer numbers are compressed by storing the minimum number of bytes needed to represent the integer as length information, plus the actual bytes as values.

Since all length information is packed in one block, it can be packed into bytes without unused gaps, i.e. at most 7 bits are wasted per block. Applying the TPC-D benchmark to a database using this compression approach was quite successful with the size of the compressed database reduced to 64% and the total query time on the compressed database reduced to 62%, both compared to the uncompressed database. However, the time for bulkloading the compressed database went up to 146%.

A more advanced compression method based on Huffman coding is suggested in [17] and implemented in the IMS DBMS as a segment store/retrieve filter. A segment in IMS is a concatenation of tuple values of various types, typically considerably shorter than a database page. Because the access granularity of the filter is on segment level, the decompression algorithm can only use (statistical) data stored in the compressed segment and due to the size of the segments this means only very little compression meta data can be stored. Furthermore, the exact boundaries of the tuple values are not known to the filter, although they would aid the handling of compression considerably. The solution suggested in [17] was to use a fixed set of contexts, each with its own set of Huffman codes optimal for the context, and switching the context adaptively whenever a symbol is encountered that is improbable in the current context but probable in another. Thus, in a context for alphabetic symbols, a number is relatively improbable, so after encountering a number the context would be switched to one for numbers. The Huffman codes for the various contexts are generated once for a given database by gathering statistical information about its contents. That is probably the biggest shortcoming of this approach because it works only on (at least statistically) static databases, as changing the Huffman codes to compensate changes in the distribution requires recompressing the entire database. When applied to entire, existing databases, the approach worked quite well, however: the authors compressed a database consisting of student records to 58% with only 17% CPU overhead due to compression.

The above publications share little common ground with this thesis because of the considerable differences between text and array data. Of course basic techniques like variable length or dictionary coding are used in some parts of the compression engine described in this thesis as well, as in most sophisticated compression algorithms, but the model layer differs completely. There is a large body of work using similar model layers in image compression, however, which was used as a design template for parts of the compression engine. The related literature on image compression is too numerous to quote at this point, but a good overview on existing techniques can be found in [47].

1.4 Structure

The thesis starts with an overview on the terminology and data types used as well as the $\mathcal{R}as\mathcal{D}a\mathcal{M}an$ architecture and its implementation of these data types in chapter 2. This is followed by the design of the compression engine as an object-oriented two-layer architecture (model- and compression layer) supporting a large range of compression techniques in chapter 3. This includes wavelets in theory and their applications in data compression,

predictors, dynamic parameter system, and closes with an analysis of transfer compression. This will be followed by an evaluation of the engine on different data types in chapter 4, and the thesis will close with some comments on the current state of the system and future work in chapter 5.

Chapter 2

Data Model and Terminology

In this chapter, the basic data model of an array DBMS will be introduced, starting with multidimensional data in general in section 2.1 and continuing with an evaluation of multidimensional arrays as a means to model MDD (*Multidimensional Discrete Data*) in section 2.2. The chapter will continue with an overview over some typical operations performed on MDD in section 2.3 and close with the implementation of MDD in $\mathcal{R}as\mathcal{D}a\mathcal{M}an$.
The following font conventions will be used in this work:

- sans-serif fonts for classes
- `fixed-width` fonts for code literals
- *`fixed-width slanted`* fonts for code variables
- *slanted* fonts for named MDD

2.1 Multidimensional Data

In the discrete case, multidimensional data is modelled as a list of sample values (*cell* values) at specific sample points, i.e. in the nD case tuples of the form $(v_i, x_{i,1}, \ldots, x_{i,n})$ where v_i are the sample values and $x_{i,1}, \ldots, x_{i,n}$ are the nD coordinates of the sample points. *Cells* are the building blocks of a multidimensional object: each cell has a unique coordinate, a base type and a value; the union of all cells forms a multidimensional object. Normally all cells are of the same type, i.e. there is a homogeneous base type for the entire multidimensional object. Cell coordinates can be stored explicitly, resulting in vector data which is particularily efficient for sparse data, or implicitly by a specific linearization order in arrays, an approach which has traditionally been used for dense data like raster images. In this work I will concentrate on the latter case, which is the format of choice of an array DBMS. The next section will introduce the concept of multidimensional arrays, their strengths and weaknesses.

2.2 Multidimensional Arrays

Arrays are a fundamental data type in programming languages. Their main advantages are constant random access time[1] and that no storage is required for the cell coordinates, because these are given implicitly by a linearization order, i.e. the cell size is independent from the MDD's dimensionality, properties vector-oriented storage in tuples does not have. The use of sophisticated index-structures can improve access times of vectored data considerably compared to the worst case linear scan (e.g. [7]), but the increasing storage requirements per sample proportional to the number of dimensions is intrinsic to the vector model. The disadvantage of array modelling is the memory consumption for sparse data, because storage is required for all points within the array, irrespective of fill level. Therefore vector modelling gains over array modelling when large portions of the data space are empty – or in a related case when adequate representation of the data requires array resolution too high for practical purposes, e.g. when local singularities are sampled at considerably higher resolution than the surrounding smooth data. It must be noted that vector storage requires considerable storage space for coordinates alone, however: a typical 2D greyscale image (8 bits per pixel) stored that way would require five bytes per sample, assuming 16 bit types for the coordinates, in contrast to array storage which requires just one byte per sample, i.e. vector storage only begins to consume less space than array storage when at least four out of five cells are empty. High dimensionality is also to the disadvantage of vector modelling because the number of bytes per sample increases with dimensionality in that case, whereas it is constant in arrays. For example, a 3D tomogram with cell values one byte in length would already require seven bytes per sample, again assuming 16 bit types for coordinate representation. This is a considerable argument against vector representation in many application areas.

When using arrays, coordinates are implicitly given by a bijective linearization function $o_\omega : \mathbb{Z}^n \to \mathbb{Z}$, which translates multidimensional coordinates $x = (x_1, \ldots, x_n)$ within an array over the nD interval $\omega = [l_1, u_1] \times \cdots \times [l_n, u_n]$, $l_i, u_i \in \mathbb{Z}$, $1 \leq i \leq n$ into (1D) offsets. With the extents of the dimensions $d_i = u_i - l_i + 1, d_i \in \mathbb{N}, 1 \leq i \leq n$, the standard linearization technique used in high-level languages like C++ ("Scanline") is given by

$$\begin{aligned} o_\omega(x) &= \sum_{i=1}^{n}(x_i - l_i) \prod_{j=i+1}^{n} d_j \\ &= ((\cdots((x_1 - l_1) \cdot d_2 + x_2 - l_2) \cdot d_3 + \ldots + x_{n-1} - l_{n-1}) \cdot d_n + x_n - l_n) \end{aligned} \tag{2.1}$$

and the matching inverse function $o_\omega^{-1} : \mathbb{Z} \to \mathbb{Z}^n$ which translates offsets back into multidimensional coordinates is

$$o_\omega^{-1}(z) = x$$

[1] Or more precisely access time is constant for all positions within an array, but it doesn't stay constant if the dimensionality of the array is changed but typically increases linearily with the number of dimensions. If a contiguous area of the MDD is iterated over, this can be optimized to be constant for the inner cells, thereby improving performance considerably; for random access, this is not possible.

$$x_1 = \left\lfloor \frac{z}{\prod_{2}^{n} d_i} \right\rfloor; \quad x_i = \left\lfloor \frac{z - \sum_{k=1}^{i-1} (x_k - l_k) \prod_{j=k+1}^{n} d_j}{\prod_{j=i+1}^{n} d_j} \right\rfloor, \quad 2 \leq i \leq n. \qquad (2.2)$$

Other linearization functions are possible and in some cases advantageous because they might better preserve proximity among cells; any space-filling curve is a candidate for a linearization function, e.g. the Hilbert curve. The above example is attractive because of its simplicity, however.

Another point in favour of array modelling is that it allows determining neighbouring cells much easier than in vector representation due to constant random access time in array modelling. Efficiently finding neighbouring cells is important for many mathematical operators like gradients as well as for extracting local correlations, which is an important issue in compression. Furthermore, there are ways to overcome these array shortcomings:

Sparsity: use compression to eliminate the empty space. In effect, a vector representation is equivalent to an offset-compression technique storing only those values that differ from a default value. Like a vector representation, this approach has the disadvantage of losing constant random access time, but unlike the vector approach, the storage requirements per cell are still constant in the number of dimensions[2].

Resolution: rather than using the same resolution throughout the array, one can superimpose arrays of higher resolution in areas of interest only and apply the same technique recursively to achieve a multiresolution representation which has logarithmic[3] random access time. A multiresolution representation like this is an integral part of some advanced compression techniques like wavelets, as a matter of fact.

None of these techniques preserve the aspect of constant random access time, therefore care must be taken when they're applied. Neither affects the storage requirements of a cell relative to the dimensionality as in the vector case, however. In many application areas, array modelling – with the possible addition of compression or multiresolution representations where feasible – has become considerably more popular than vector modelling, for example raster images, tomograms, many kinds of simulation and observation data.

The terminology used in the remainder of this thesis is founded on the concept of *Multidimensional Discrete Data*, or MDD for short [1]. MDD are based on array modelling of multidimensional data and provide a generic template type for this kind of data. The MDD template type must be instantiated with cell type information and a spatial extent descriptor to obtain a concrete data type. I will use the following definitions:

Cell: an MDD consists of cells situated at the nodes of a regular grid bounded by the MDD's spatial domain; each cell has a base type.

[2]The reason for this is that in array modelling a linearization function always exists and therefore instead of the actual coordinates the linearization offsets can be used instead, which are independent of the dimensionality.

[3]Logarithmic in the number of hierarchical resolution levels.

Base Type: a base type describes the structure of each cell within a given MDD object. A base type can be an atomic type like `int` or `float` or a structured type which may contain atomic types or other structured types. With the conventional atomic types defined by ODMG [9], this leads to the following definition of a base type:

```
   base_type  =  struct_type | atomic_type
 atomic_type  =  boolean | char | octet | short | ushort | long |
                 ulong | float | double  [ name ]
 struct_type  =  struct [name] { base_type [ , base_type ] * }
       name   =  any variable name
```

The *name* is optional when describing the structure of the base type, but can be used to bind arbitrary (sub)types to identifiers; in the absence of names, numbers have to be used for identification. An example for a structured base type is the standard `RGB` type `struct { char red, char green, char blue }` used for colour images.

Because the important array property of constant random access time of arbitrary cells should be preserved, base types (and therefore cells) must have constant length, within an MDD[4], as this is required in order to calculate a cell's address as $o_\omega(x)$ times the size of a cell; if the cell size was variable, this would no longer be possible and an iteration over the preceding cells would be required, thereby losing the excellent random access properties of arrays. Because according to the above definition a base type consists of a set of elements in *atomic_type*, all of which have constant length by definition, any base type generated from this grammar also has constant length (constant meaning independent from the value).

Channel: array data belonging to the same atomic type forms a channel, for example an MDD over the `RGB` base type consists of three channels for the three atomic types contained in the base type.

Spatial Domain: (*sdom*) an nD interval describing the spatial extent of an array. I will use the notation $sdom_n = [l_1 : u_1, \ldots, l_n : u_n]$ to describe an nD interval with lower bounds l_i and upper bounds u_i, $l_i, u_i \in \mathbb{Z}, l_i \leq u_i$. There is also a special form without fixed boundaries for MDD type definitions only, using the asterisk $*$ for l_i or u_i (or both) which stands for any value; an MDD can't be instantiated with a spatial domain containing $*$, but an MDD type can be defined that way (see the examples below). A spatial domain differs from the carthesian product $[l_1, u_1] \times \cdots \times [l_n, u_n]$ in that it supports only integer coordinates, which allows applying a linearization function like equation (2.1), whereas the carthesian product describes a continuous region in $\mathbb{R}^n$ where this is not generally possible[5].

Using these definitions, any multidimensional array satisfying the constraint of constant base type size can be constructed from the MDD template by instantiating it with a

[4] i.e. the length must be independent from the value, which is not true for e.g. strings.

[5] the value of $o(x)$ would no longer be an integer, which means it can't be used to calculate a cell address in memory.

(concrete) spatial domain and a base type. MDD types describe classes of MDD and may be parameterized with arbitrary spatial domains, like in some of the following examples:

$M_{[*:*],\text{char}}$	any 1D array of characters
$M_{[0:255,0:255],\text{struct \{char r, char g, char b\}}}$	2D RGB images with 256×256 pixels
$M_{[*:*,*:*,*:*],\text{short}}$	any 3D array over the base type short
$M_{[*:*,0:15,0:31,0:63],\text{float}}$	any 4D array over a floating point base type with arbitrary extent in the first dimension and fixed extent in the other 3

2.3 Operations on Multidimensional Arrays

Having established the MDD concept, we now look at typical operations we want to execute on this data type. These can be classified into the following major categories:

1. **spatial transformations**: changing the spatial extent of an MDD, typically reducing it;
2. **base type projections**: restricting an MDD over a structured base type to the values belonging to a subset of the base type, typically one atomic type;
3. **cell operations**: performing calculations on the data.

These operations represent orthogonal concepts which can be combined in arbitrary fashion, such as executing a data operation on an MDD to which a spatial transformation and a base type projection were applied. We will now describe these operations in more depth (for a full description of the algebra see [3]):

2.3.1 Spatial Transformations

Spatial transformations change the size of an MDD or its dimensionality (or both). For the remainder of the section we will assume that the MDD has the spatial domain $[l_1 : u_1, \ldots, l_n : u_n]$. A spatial transformation is described by one restriction r_i per dimension; this restriction can have the following forms:

section: $a_i : b_i$, denoting a lower boundary a_i and an upper boundary b_i for dimension i, which means that in dimension i all data with coordinates between a_i and b_i (both inclusive) should be used and all values outside of that interval are discarded. Obviously, the necessary restrictions on the values are $l_i \leq a_i \leq b_i \leq u_i$ (where $a_i = b_i$ is a borderline case which should normally be replaced by a projection). For convenience we will also use the asterisk $*$ as an alias for l_i when $a_i = *$ and as an alias for u_i when $b_i = *$, i.e. $* : *$ would be the entire range.

projection: p_i, which is the value to project to in dimension i. A projection collapses the interval to a point and thereby reduces the dimensionality of the MDD by one. The values at the projection hyperplane $x_i = p_i$ are used, all the other ones are discarded. Naturally, the restriction $l_i \leq p_i \leq u_i$ must hold.

A projection to p_i and a section with $a_i = b_i = p_i$ result in identical data, but the projection also reduces the dimensionality by one in contrast to the section which leads to an interval containing only one legal coordinate. Since this representation is redundant, such a section should normally not be used.

Spatial transformations can be used to perform range queries on MDD via sections, such as selecting an area of interest within a large MDD, such as the hypothalamus area within a 3D volume tomogram, or to retrieve orthogonal cuts through the data using projections, for instance a still frame from a movie (3D spatio-temporal MDD). Sections and projections can be mixed arbitrarily for different dimensions.

2.3.2 Base Type Projections

Base type projections can be used on MDD over a structured base type to select a subset of the MDD's channels and discarding the rest. A typical example of this would be selecting one spectral channel of a multispectral image (e.g. the green channel of an RGB image).

2.3.3 Cell Operations

Cell operations perform calculations on the actual data (in contrast to the transformations mentioned in the previous two sections which operate on the data's schema and map the data to a subset). Typical examples of such operations range from adding constants to all cells, over masking two MDD or summing up the cell values, to complex operations like a Fourier transformation. The basic calculation classes are

induced: these operations simultaneously apply a base operation to all cells of an MDD [42]. Depending on the cardinality of the operator, there are the following induction types:

- **unary induced:** MDD → MDD. Examples are "-" which multiplies all cells by -1, or `not` which performs the boolean *not*-operation on all cells;
- **binary induced:** MDD × scalar → MDD or MDD × MDD → MDD. Examples for the first case would be adding or multiplying by a constant to brighten/darken an image. The second case could be e.g. subtracting or multiplying two MDD: subtracting an MDD from a shifted version of itself can be used to obtain a discrete approximation of the gradient field in a specific direction, whereas multiplying two MDD can be used for masking, for instance when a complex region of interest is stored as a binary mask in a separate MDD;

condense: MDD → scalar, e.g. summing all cells.

2.4 Implementation of MDD in RasDaMan

In this section I will introduce the basic $\mathcal{RASDAMAN}$ architecture, show how MDD are handled and what kinds of operations are supported on them. I will then explain where a multidimensional compression engine fits into this architecture and what modules it affects.

$\mathcal{RASDAMAN}$ is an array DBMS which uses a conventional base DBMS (BDBMS) as storage and transaction manager; an overview of the architecture is depicted in figure 2.1. $\mathcal{RASDAMAN}$ interacts with the BDBMS via a specific interface layer at the lowest level of its internal module hierarchy which encapsulates all communications with the BDBMS in question; this means that $\mathcal{RASDAMAN}$ can be ported to a different BDBMS by simply exchanging this interface layer without having to change higher level modules. At the time of writing this thesis, there existed drivers for three different BDBMSs, the object-oriented O_2 system as well as the relational DBMSs *Oracle* and *DB2*.

MDD M are stored in the BDBMS as BLOBs (*Binary Large Objects*), but usually not in their entirety but decomposed into non-overlapping smaller units called *tiles*, where each tile t_i is a small MDD itself with the same base type and number of dimensions as the entire MDD and $\bigcup_i t_i = M$. A tile contains that portion of the MDD's data, which intersects the tile's spatial domain $sdom(t_i)$, linearized into row major format as used in C/C++ which is also the implementation language, i.e. those cells whose last coordinate differs by 1 lie closest together in the linearized representation. For structured base types, the data for each cell is stored interleaved, so for the RGB type the red, green and blue components of cell c_i are stored as three consecutive bytes, the next three bytes are the colours for cell c_{i+1} and so forth.

Tiles are normally several database pages large, typically between 32kB and 512kB, depending on the size of the entire MDD and its regions of interest. They represent the finest access granularity, i.e. whenever an arbitrary cell within a tile is accessed, the entire tile has to be loaded, which is an important factor when designing the compression engine. The tiling model was chosen in order to allow efficiently handling arbitrary query boxes independently of the size of the entire MDD: if the MDD was stored in one BLOB, large and mostly discontinuous parts of the BLOB would have to be accessed in order to retrieve cells within a query box [24]; by using tiles, data is better localized, plus only those tiles that lie fully or in part within the query box have to be accessed by the server in the first place, which makes the system scale with the size of the query box rather than the size of the MDD.

Very small tiles are stored inside the tile index itself (*inlined tiles*) rather than in external BLOBs. The main motivation for this is to avoid wasting storage space in case the tile size is noticeably below the size of a database page, because then the space wasted could actually exceed the tile size. While the size of uncompressed tiles can easily be configured to be several database pages large, this is no longer possible with compression, where for instance a tile with identical cells everywhere can usually be compressed to just a few bytes. Therefore it is important to be able to store very small tiles efficiently, in particular if the DBMS is capable of compression, and inlining them is one possible

solution.

The next higher level of $\mathcal{RasDaMan}$ contains modules for the tile index, tile-based operation execution and the catalog for base type management. The tile index encompasses several multidimensional indexing techniques including a simple directory index as well as an nD R+ tree; its purpose is efficiently finding tiles contained in given spatial domains when doing range queries on MDD. In depth information about the tile index is provided in [24]. The catalog manager implements the dynamic base type system, including its persistent storage and materialization, as well as operations on various combinations of base types, which is used by the query execution engine. Query operations are executed on tile level in the tile manager module.

The top level module in the $\mathcal{RasDaMan}$ server is the query engine consisting of parser, optimizer and executor. The query engine implements the query language *RasQL* [3, 42], an extension of SQL92 [29], which adds the MDD paradigm to SQL. The query parser module builds a query tree from the textual query. The optimizer then uses this tree structure to find an efficient execution plan using various algebraic transformations; this plan is then fed into the executor which processes the query using the lower level modules of the architecture, on tile level if possible, to keep down memory requirements. A detailed

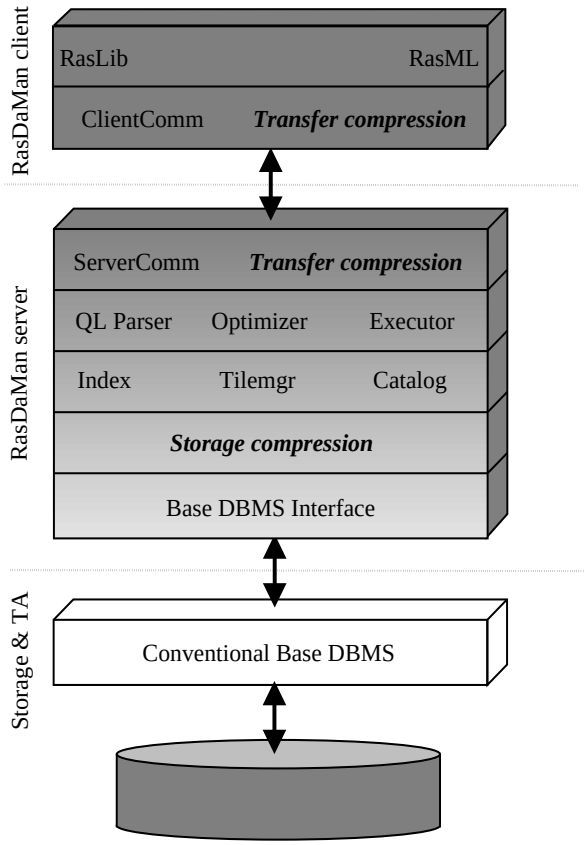


Figure 2.1: The basic $\mathcal{RasDaMan}$ architecture

overview of the query engine can be found in [42].

$\mathcal{R}asDa\mathcal{M}an$ server and client applications interact through network-transparent communication layers which translate primitive operations like sending or receiving scalars, MDD or MDD collections into RPC calls, thereby allowing client and server to reside on arbitrary computers as long as they have a network connection. In addition to the client communication layer, the C++ client library *RasLib* contains classes encapsulating entities like base types, spatial domains, MDD or queries in order to facilitate the development of client applications.

Chapter 3

Compression Engine Architecture

In this section, I will present the architecture of the MDD compression engine as implemented in the $\mathcal{R}asDa\mathcal{M}an$ array DBMS [20]. One application of the compression engine already mentioned in section 1.1 is transfer compression, which requires some compression functionality in client programs as well. In order to improve source code maintainability and grant maximum flexibility in transfer compression, the engine was implemented as a module shared by client and server. The only difference is how data exchange formats are handled on client and server, which will be discussed in more detail below.

A major design criterion was modularity and extensibility, because as mentioned in the introduction, there exists an abundance of specialized compression methods to choose from whose efficiency depends on the type of data they're applied to. Most importantly, there is no universally ideal compression method, therefore several alternatives should be provided, covering a wide range of data types. In addition, the engine should be easy to extend as requirements change over time or better compression methods become available. These requirements can be modelled very well in an object oriented fashion with an abstract compression base class defining the interface and an extensive class hierarchy derived from it implementing the concrete compression techniques.

The compression granularity is another important design decision. Because compression usually makes random data access impossible, the granularity must be chosen fine enough to avoid (de)compressing large amounts of data when only a small fraction of it is actually accessed. There are some simple compression techniques where this problem is reduced, e.g. changing a cell value in offset-compressed data doesn't require recompressing all data following the changed cell, but these techniques usually don't achieve very good compression, so the engine must cater for the limitations of more complex methods like dictionary or variable length compression techniques as well. An extreme example would be the access of a single cell in a volume tomogram which was compressed in one block: for all access types, all data from the start up to the cell in question[1] has to be decompressed, whereas write-operations additionally require compressing all data from that cell to the last one in the encoding order used. Clearly, this kind of access granularity is unacceptable.

[1] Relative to the chosen encoding order.

Since $\mathcal{R}as\mathcal{D}a\mathcal{M}an$ is already tile-based – for the same reason of reducing access granularity – it is natural to make tiles the compression granularity as well. That means that

- any access to data within a tile requires (de)compressing at most the entire tile. If a compression method allows decompressing subsets of the tile it may be less;
- the compression type is defined on tile level rather than MDD level (or even higher). This means that the compression type can differ between tiles belonging to the same MDD, like uncompressed tiles, tiles with lightweight compression and others with high performance compression. This flexibility is attractive with respect to known access patterns and regions of interest, where a strategy could be to compress data that is rarely used and keep frequently accessed regions uncompressed to achieve a good ratio of average access time vs. total storage requirements.

Most modern raster data compression technology, like JPEG [60] or the upcoming JPEG2000 standard [51], have a two-layer architecture where the functionality is divided in the following way over the two layers:

Model layer: the top layer transforms the data according to a data model into a format more suitable for compression, such as the discrete cosine transformation (DCT) in JPEG and/or channel decorrelation such as the RGB → YUV conversion often found in image compression[2]. In case of lossy compression, quantization is usually attributed to the model layer as well. The top layer does not perform any actual compression, in many cases the data is actually expanded[3] (but typically sparsely populated);

Compression layer: the bottom layer implements traditional symbol-stream-oriented compression methods such as variable length techniques like Huffmann coding and the newer arithmetic coding (AC) [63], or pattern-oriented techniques like run-length encoding (*RLE*) and dictionary techniques like the popular LZ77 [64] used in `gzip` and `pkzip`, or even combinations of these like *RLE* followed by AC. Actual data compression is done in this layer only. Quantization could also be done on this level, but usually is performed at the top layer because additional information such as "acceptable" loss levels etc. is no longer available in the bottom layer.

There are considerable advantages to this approach: formally there is the clean separation of different tasks (transformation vs. compression) into two distinct modules which allows exchanging components of either layer without affecting the other, such as using a different compression layer without having to change any part of the transformation layer

[2]Essentially this transformation converts image colours into luminance (Y: grey level) and chrominance (U,V: difference from grey level) components. This system is often used in compression because the human eye is better at distinguishing differences in brightness than differences in colour, so the chrominance components can be encoded in fewer bits than luminance without affecting perceived image quality.

[3]In JPEG, for example, the DCT coefficients need more precision than the input data, therefore the transformed data is larger than the original data.

or being able to reuse all components in the compression layer for different transformation techniques. Due to the similarities of image data and many common types of MDD, it is therefore natural to also use this tried and tested two-layer architecture in an MDD compression engine. This leads to two separate class hierarchies linstream described in section 3.1 for the compression layer and tilecompression described in section 3.2 for the transformation layer.

An important issue in compression is offering the user the possibility to configure the compression methods with various parameters. Quoting two popular examples, *ZLib* allows specifying a compression level as a time vs. size reduction tradeoff parameter, whereas *JPEG* offers parameters for the image quality vs. size reduction and a smoothing factor. While the parameters in these examples are integers, this is not necessarily true for other compression techniques where floating point or string parameters can prove useful, neither is there a fixed set of parameters nor is it sensible to burden any compression type with the parameters of all compression types taken together. This problem will be addressed in more detail with the dynamic parameter system in section 3.6.

3.1 The Compression Layer

The compression layer represents traditional compression methods based on the concept of a (1D) symbol stream. Most importantly, no MDD properties are known at this level, therefore preprocessing done in the transformation layer must exploit MDD properties and dispatch data to the compression layer in a format that allows efficient compression, such as transforming coefficients channel-by-channel rather than interleaving channel coefficients. Because all data is interpreted as a 1D (or *linear*) symbol stream in this layer, it is represented by an abstract base class linstream which acts as a filter with linear streams of symbols as input and output. Figure 3.1 shows an outline of this class hierarchy. Due to this generality, the various compression classes in this layer may also be used for other tasks than MDD compression, such as compressing server log files etc.

All classes derive from a common ancestor class linstream, but immediately split into separate branches for compression and decompression streams where the actual interfaces are defined. The reason for this is that normally compression and decompression work very differently internally. For instance when compressing, the size of the compressed data is usually not known in advance, so the compression stream must be able to efficiently handle compressed data of almost arbitrary size, whereas when decompressing the sizes of both the compressed and the uncompressed data are typically known. This leads to considerable differences in the interface, not to mention the internal implementation. Nevertheless, both compression and decompression streams provide mostly the same interface methods, although their signature differs. Both branches provide a static `create()` method for conveniently constructing an object of class lincompstream or lindecompstream from a format enumerator. The lincodecstream class is separated from this hierarchy and serves as a convenient way to get matching compression/decompression streams. The basic interface methods for the compression and decompression classes are the following:

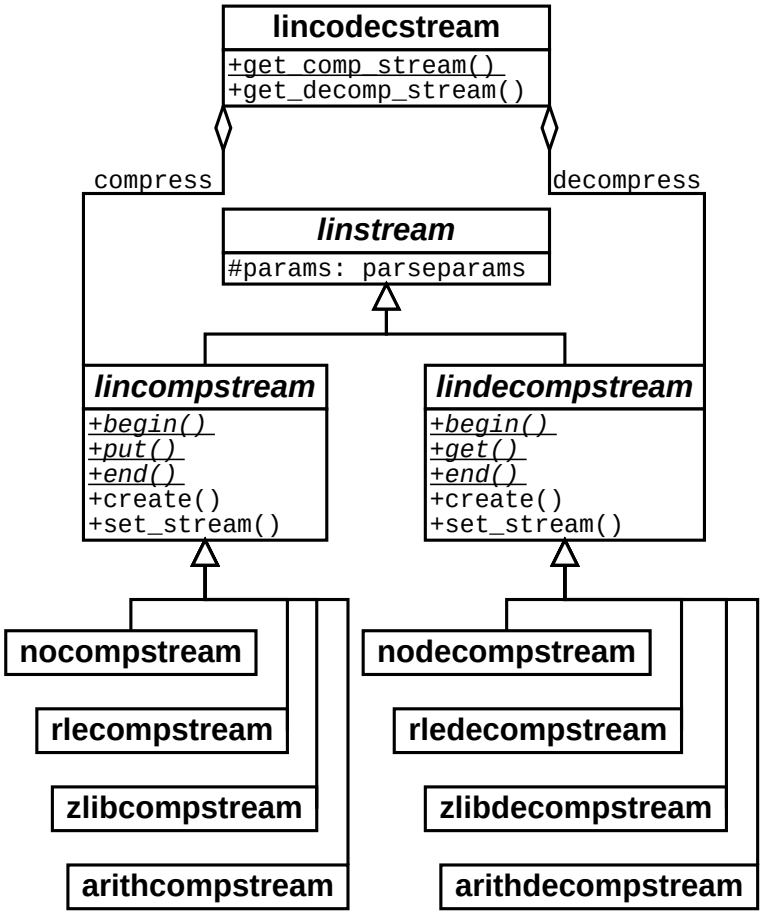


Figure 3.1: The linstream class hierarchy
This diagram shows a skeleton of the bottom (= compression) layer of the MDD compression engine in UML *notation. Attributes are ignored and only a subset of the available methods is displayed to keep the diagram readable.*

lincompstream:

`begin(`*`typeSize`*`, `*`dataSize`*`):` prepare the compression stream for symbols of length *`typeSize`* and a total size of approximately *`dataSize`*. Typically, this method allocates internal buffers and prepares the compression environment. Neither parameter is strictly necessary but can be used by compression streams to improve their performance. For example *`dataSize`* can be used to determine the size of internal buffers and *`typeSize`* is used by *RLE* compression (see below).

`put(`*`data`*`, `*`size`*`):` compress *`size`* bytes starting at *`data`* and store the result internally.

`end(`*`data`*`, `*`size`*`):` stop compression, flush all data and free resources; return the compressed data and its size.

`set_stream(stream)`: activates the streaming interface (see below) and redirects the compressed output data into another lincompstream object.

lindecompstream:

`begin(typeSize, data, dataSize)`: prepare the decompression stream for symbols of length *typeSize* from compressed data starting at *data* and size *dataSize*. The *typeSize* parameter is not strictly necessary, but if anything but the default value is used it must be consistent over compression and decompression.

`get(data, size)`: read *size* bytes of symbols from the compressed data and store them at *data*.

`end()`: stop decompression and free resources.

`set_stream(stream)`: activates the streaming interface and redirects the stream's input data from another lindecompstream object.

All linear streams support two modes of operation, a *static* one where the compressed data is directly stored to and read from memory and a *streaming* one where data is stored to and read from another linear stream. The streaming mode allows efficiently concatenating linear streams to arbitrary depth and thereby forming banks of compression streams, whereas static mode is intended for the last stream in a bank or a standalone stream for actual storage. In this context, efficiency means that no stream in a bank has to finish before the next in line can start operation, i.e. there is no need to explicitly store intermediate data in its entirety: small internal buffers suffice, saving considerable amounts of memory for larger stream banks. While it is usually not advantageous to concatenate two streams of the same type[4], certain combinations of different streams can cooperate quite well.

3.1.1 The Compression Streams

The compression layer consists of four stream types, a trivial one that just stores its input data plus three compression streams representing the major compression techniques in use today with different weights on compression quality vs. computational overhead. I will now give an overview of these streams including typical application areas; a class diagram of all compression streams can be seen in figure 3.1.

None:

No compression, the data is not (de)compressed but merely copied. This stream type is present for symmetry and to facilitate debugging higher level compression techniques which use compression streams for storage.

[4] A given compression method usually can't reduce the entropy of its own output data any further.

RLE:

Run-Length Encoding compression is one of the oldest and simplest compression techniques there are. Nonetheless, it is still in common use today because it achieves good results on sparse data and requires very little computational power. *RLE* algorithms compress sequences of symbols $s_i \in S$, $1 \leq i \leq n$ over an alphabet $\mathcal{A} = \{a_1, \ldots, a_m\}$ with the same value ($\rightarrow$ *runs*) by decomposing the symbols into a sequence of k tuples $r_j = (l_j, s_{i_j}) \in L \times \mathcal{A}$, $1 \leq j \leq k \leq n$ where s_{i_j} are those symbol values $s_i : s_{i+1} \neq s_i$ and $l_j \in L$ are the number of repeats (L is a set of length counts). Obviously this algorithm is fast because it only has to pass over the data once, i.e. $O(n)$; it also works well for sparse data where there are many consecutive symbols with identical values. A big problem of primitive *RLE* algorithms is worst case data *expansion*, however, because if there are no runs, $k = n$ and a primitive encoding of the tuples would make the size of the compressed data $n \cdot |l|$ *larger* than the original data where $|l|$ is the data size of a length count. In the typical case, the sizes of a symbol and a length count are both bytes, leading to a worst case data expansion to twice the original size.

There are more efficient ways to encode the tuples, however. One is to define an escape symbol s_e, encoding tuples with $l_j > 2$ as triples (s_e, l_j, s_{i_j}) where s_e signifies that the following symbols represent a run tuple. The escape symbol itself is encoded by the tuple $(s_e, 1)$, i.e. the value of the run symbol is redundant because no real run has length 1; all real runs require tuples with all three entries. The main disadvantage of this technique apart from less efficient encoding of run-tuples is that usually there is no distinct escape symbol which doesn't appear as normal symbol too, i.e. the (normal) symbol must be encoded as $(s_e, 1)$. Depending on the probability $p(s_e)$ of the escape symbol in the input stream, this leads to a data expansion by $p(s_e) \cdot n \cdot |l|$. For a fixed escape symbol this leads again to a worst case data expansion of $n \cdot |l|$ which is unacceptable, although much less likely than the worst case when using primitive encoding. An adaptive algorithm, on the other hand, can use the least frequent symbol as escape symbol, i.e. $p(s_e) \leq \frac{1}{m}$, and thereby limit the data expansion to $\frac{n}{m}|l|$. For the typical case of bytes with 256 symbols as alphabet, this means a worst case data expansion by only $\frac{1}{256}$th. Unfortunately an adaptive algorithm needs two passes over the data which adds complexity and doesn't fit very well with a streaming interface, therefore this kind of encoding is not suitable for a compression layer stream.

A very efficient way to encode run length data is the *PackBits* algorithm described in [53]. This introduces the concept of a *literal run* as a sequence of tuples $(1, s_{i_j})$ (corresponding to sequences of different symbols in the input stream) which are encoded as a sequence $\bar{l_j}, s_{i_j}, s_{i_j+1}, \ldots$ where $\bar{l_j}$ is the number of symbols in the literal run. This approach requires dividing the set of length counts L into two disjunct sets L_l and L_r for literal and run counts, typically with $|L_l| = |L_r|$. This can be implemented by interpreting length counts $l_j \in L$ as signed numbers, where a positive sign identifies a literal run followed by the l_j literal symbols and a negative sign

identifies a normal run (i.e. repeating symbols) followed by the symbol that is to be repeated $|l_j|$ times. This approach has the disadvantage that due to the distinction into positive and negative values, the maximum length of a run is limited to half the value range of L; on the other hand, the worst case data expansion is $2n\frac{|l|}{|L|}$ without needing more than one pass over the data, i.e. $\frac{1}{128}$th when using bytes for lengths and symbols. Furthermore, there is no additional overhead when encoding normal runs like when using an escape symbol, i.e. a run can be encoded in two symbols rather than 3. Therefore *PackBits* was chosen as the *RLE* format for the linear stream.

As a further optimization, the algorithm was extended to operate on input symbols larger than one byte to allow compressing data of larger atomic types more efficiently. Currently supported sizes are 1,2,4 and 8, covering the sizes of all popular atomic types. Without this optimization, the input stream `<1,2,2,2,2>` over a 16 bit base type would be interpreted as a sequence of bytes `<0,1,0,2,0,2,0,2,0,2>`[5] which obviously doesn't compress at all in *RLE*. Length counts are always encoded in 8 bits, however, because larger base types usually don't mean longer runs.

ZLib:

This stream corresponds to the well-known standard compression library *ZLib* [67]. *ZLib* compression is based on the LZ77 adaptive dictionary compression technique [64] combined with Huffman coding of symbols and dictionary references, which is far superior to *RLE* in terms of achievable compression rates, but also of considerably higher complexity. *ZLib* compression and equivalent techniques have been in common use for many years in the form of the popular *gzip* and *PKZip* compression programs and its continued relevance is stressed by the relatively new PNG image standard [41] where *ZLib* is used for data compression.

Dictionary techniques replace sequences of symbols with references into a dictionary, thereby achieving compression as long as the reference takes up less storage than the actual symbol sequence (this is a criterion for the minimum length of sequences in the dictionary). The performance of this class of compression obviously depends heavily on how well the dictionary suits the data to compress; if for instance a dictionary of english words was used to compress a german text, much fewer sequences could be replaced by references into the dictionary than if a german dictionary had been used. One could set up a large number of different static dictionaries and choose the optimum one for compression, but there are several problems with this approach:

- finding the optimum dictionary for a stream of symbols requires compressing the data with all dictionaries and comparing the compressed data sizes. Considering the large amount of dictionaries needed for the vast number of different data types, this is not feasible;

[5] On a little endian computer; for a big endian computer, exchange the values at even and odd positions. The consequences for *RLE* compression are the same for either endianness.

- finding a "good" dictionary requires heuristics, such as "use an english dictionary to compress an english text". While this will work reasonably well in special cases like text, this is not true for arbitrary binary data;
- in most cases, self-contained data is preferred, which means the dictionary should be included in the compressed data, but explicitly storing the dictionary along with the compressed data can increase the data size considerably.

Therefore most dictionary techniques in common use today are adaptive, i.e. they build the dictionary from the actual data rather than using a static dictionary. The most popular approaches are the ones developed by Lempel and Ziv, LZ77 [64] and LZ78 [65]. LZ77 uses a window W of width w over the most recently processed symbols as a dictionary, which allows making a dictionary reference by using an offset from the current position and a length (see figure 3.2). The disadvantage of this approach is that sequences are removed from the dictionary based on their position, not their frequency. This was addressed in LZ78 [65] where an explicit dictionary is built from new sequences found in the data. Despite the removal of the theoretically undesirable properties of LZ77, LZ78 implementations often perform worse than LZ77 ones[6], as can be seen by comparing the compression tools `compress` (LZ78) and `gzip` (LZ77), therefore LZ77 was chosen implicitly via *ZLib* as the dictionary technique used in the lincodec class hierarchy.

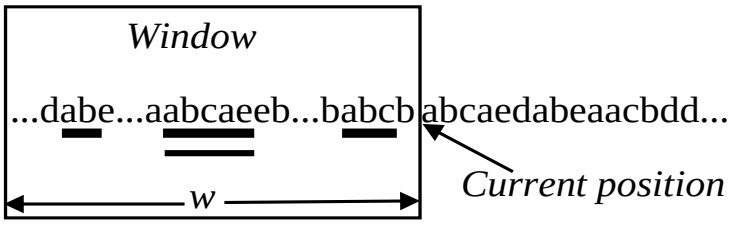


Figure 3.2: LZ77 dictionary compression

This figure shows the dictionary window in LZ77 compression and its role when compressing data at the current position. The algorithm searches in the dictionary window ending at the current position for the longest match with the sequence starting at the current position. Matches in the dictionary are underlined, the best (longest) match is underlined twice.

Regarding complexity, dictionary techniques are very expensive during compression and cheap during decompression. For LZ77, compressing data at a position p within the stream with at least w preceding and w following symbols requires comparing the sequence starting at position p with the sequences at positions $p-j$, $1 \leq j \leq w$; note that the sequence can be longer than j symbols, i.e. the referenced sequence may end after the current position. In order to lower the complexity of the compression algorithm, the maximum length of a reference is usually limited to m symbols (258

[6]The reason for this is that due to the higher complexity of building an explicit dictionary, its size is usually smaller (12 bit) than the LZ77 window (15 bit).

in *ZLib*). This leads to a worst case complexity of $w \cdot m$ comparisons for finding the optimum dictionary entry at each position p, although it is usually considerably lower because the comparison fails early on for most j. Considering the typical window size of 32kB in *ZLib*, the complexity is obviously much higher than for *RLE*, although still linear in the number of input symbols. The complexity can often be reduced dramatically by using hashtables[7], but as usual with hashing there is no guaranteed performance improvement and the worst case is still theoretically possible.

Arithmetic:

The principles of arithmetic coding date back to the 1960ies, but it wasn't before the end of the 1980ies that implementations of arithmetic coding started appearing, one of the first of which was presented in [63]. The basic idea is to represent a sequence of input symbols $S = \{s_1, \ldots, s_n\}$ over an alphabet $\mathcal{A} = \{a_1, \ldots, a_m\}$ with probabilities $p_1, \ldots, p_m$ by a real number $\in [0, 1[$. Let $v_i = \sum_{j=1}^{i} p_j$, then obviously $v_0 = 0 < v_1 < \cdots < v_m = 1$ and the unit interval can be represented as the union of the non-overlapping intervals $I_i = [v_{i-1}, v_i[$, $1 \leq i \leq m$. Now a symbol a_i can be represented by any number within I_i. If we now rescale interval I_i to the unit interval, we can apply the same partitioning scheme recursively to encode the next symbol, right unto the end of the stream of input symbols. With each new symbol encoded, the interval representing this symbol becomes smaller and the number of bits required to encode it with sufficient precision to allow exactly reconstructing the input sequence increases, without a fixed limit; this was one of the main reasons why arithmetic coding was not implemented for a long time and its complexity limited its practical application until recently.

The reason why probabilities are used as basis of the unit interval decomposition when any sequence with $0 < p_i \leq 1$ and $\sum p_i = 1$ could be used is to minimize the number of bits required for frequent symbols: if p_i is high, the interval I_i is big and relatively few bits are required to represent any value within it, whereas for an infrequent symbol the opposite applies. This allows modelling the probability of a symbol much more precisely than in e.g. Huffman coding which assigns a fixed integral number of bits to each symbol and therefore can only represent probabilities 2^{-i} precisely.

The main problem when implementing arithmetic coding is modelling the unlimited precision required for the interval boundaries. Unlimited precision would be very expensive because it's not supported by computer hardware and therefore would have to be emulated in software (and the complexity for encoding a symbol would increase with the length of the input sequence). Fortunately, this isn't necessary because limited precision is sufficient as long as encoder and decoder stay in sync;

[7] For each position within the dictionary window a hashcode is generated from the n_h symbols starting at this position, where n_h is a small number, typically three. These hashcodes are then used as keys to look up the positions of the sequences with the same hashcode as the one starting at the current position.

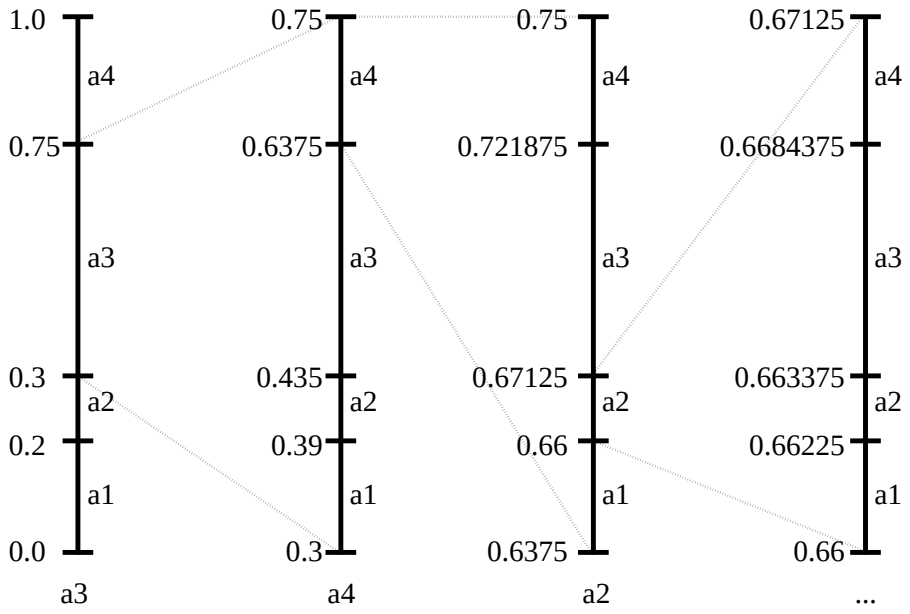


Figure 3.3: Arithmetic coding

This figure shows how arithmetic coding of a symbol sequence $a_3a_4a_2$ works, given an alphabet $\mathcal{A}$ containing four symbols $a_1, \ldots, a_4$ with probabilites $p_1 = 0.2, p_2 = 0.1, p_3 = 0.45$ and $p_4 = 0.25$. The first symbol is a_3, corresponding to the interval $[0.3, 0.75[$, which is then partitioned according to the probabilities like the unit interval. With each new symbol, the interval representing the sequence encoded so far is getting smaller. After three steps, any number in the interval $[0.66, 0.67125[$ can be used to represent the input sequence.

this allows using fixpoint arithmetic and flushing out the most significant bits at suitable points during encoding, rescaling the interval bounds and operating with a limited number of less significant bits. With a fixpoint representation for the interval bounds l, h in $[0, 1]$ there are three situations where the current interval can be rescaled to compensate for its contraction, which are explained here for the encoder:

1. $l \geq 0.5$: in this case the most significant bit of both l and h is 1. Because the interval after encoding a new symbol is always fully contained in the interval before the encoding, the most significant bit can't change in this situation: it may be shifted out, resulting in a rescaling of the current interval by 2 ($l := 2(l - 0.5)$, $h := 2(h - 0.5) + 1$);
2. $h < 0.5$: in this case the most significant bit of both l and h is 0 and can be shifted out for the same reasons as in the case of $l \geq 0.5$ ($l := 2l$, $h := 2h + 1$);
3. $0.25 \leq l < h < 0.75$: in this case the interval is rescaled to $l := 2(l - 0.25)$, $h := 2(h - 0.25) + 1$ and the encoder memorizes how many consecutive times k transformation (3) has been performed in succession. As soon as (1) or (2) are executed, k bits opposite to the one emitted by one of these are emitted first and k is reset to 0.

Note that these rescaling operations ensure that at all times $h - l \geq 0.5$, thereby making sure that there is always a sufficient number of bits of precision to compensate for interval contraction, let alone contraction to a point.

The main advantages of arithmetic coding compared to Huffman coding [27] are better modelling of probabilities into number of bits used and more efficient techniques for adaptive coding and alphabets with a very large number of symbols. Concerning the last point, in Huffman coding the full tree representation for all symbols must be materialized during encoding/decoding, making it unfeasible for very large alphabets, as for instance a Huffman code tree for 32 bit symbols would have to contain codes for 2^{32} symbols, which would take up several gigabytes of memory; in arithmetic coding, even such a vast number of input symbols could still be modelled efficiently provided the probability function has an analytic representation[8]. These advantages have made arithmetic coding the de facto standard in modern entropy coding techniques and has also been adapted in JPEG2000 [51]. For more details on arithmetic coding see [63, 47].

3.2 The Transformation Layer

Whereas the compression streams in section 3.1 perform actual compression on already linearized data, the transformation layer's task is to preprocess the data in a way to make the subsequent compression more efficient. The transformation layer itself doesn't perform data compression, in some cases it may actually expand the data; e.g. for general wavelets, data is transformed into floating point coefficients, which are later quantized into integers which may be larger than the original data type. Usually this transformed data is sparse, however, and will therefore compress better even though it may be larger than the original data in its uncompressed form. The transformation layer is MDD-aware and can therefore exploit MDD properties such as the semantics of (structured) base types and correlations of neighbouring cells.

The transformation layer is modelled as an extensive class hierarchy based in the abstract root class tilecompression; the name was chosen because the compression engine is tile-based and the transformation layer operates directly on tiles. Each tile references an object of the tilecompression hierarchy it uses for transparent compression and decompression of the tile data. The interface provided by the tilecompression class mainly consists of `compress()` and `decompress()` methods, apart from means of identification. The immediate children of the tilecompression class add predictors, which will be treated in more detail in section 3.5. Tiles normally use the interface methods transparently, i.e. when tile data is accessed by a higher level module (see section 2.4) and the tile was compressed, it is decompressed; conversely, when a tile is stored or transferred, it is compressed with the compression class it references. This allowed easy integration of tile compression into the DBMS architecture, but leaves out some optimization potential, as some compression

[8] Otherwise probability tables are needed, whose size would also increase with the alphabet size.

methods like *RLE* allow executing simple induced and aggregation operations (see section 2.3.3) directly on the the compressed data. Since this is only true for very simple compression algorithms and decompression can't be avoided when the tile needs to be trimmed, the advantages of extending the system in this respect are doubtful; OLAP data cubes would probably profit, however, being sparse and therefore highly compressible even with simple compression like *RLE*.

An outline of the entire tilecompression hierarchy can be seen in figure 3.4. The immediate children of the tilecompression base class are tilecompinter which includes interchannel prediction (correlates cell values across channels) and tilecompred which furthermore adds intrachannel prediction (correlating neighbouring cell values within one channel). The simpler classes will be explained in section 3.2.1, whereas wavelets are covered separately in section 3.3; predictors will be presented in section 3.5.

3.2.1 The Transformation Classes

There are four major transformation categories in the tilecompression hierarchy, differing in the MDD properties used and the weights of complexity vs. achievable compression. I will give a short overview of these classes in the remainder of this section; the wavelet class is considerably more complex than the other ones, therefore wavelets will be introduced in more depth in section 3.3.

tilecompnone: No compression.

This class is optimized for fast handling of uncompressed data to avoid copy operations which are unneccessary when no (de)compression takes place; as far as mere functionality goes it could also have been implemented in e.g. tilecompstream described below. As a special case there is also the derived class tilecompother which handles all tile data formats which don't fall into any of the other categories of the transformation layer; the only difference to its parent class is that it stores the data format descriptor (which is otherwise given implicitly by the kind of the tilecompression object). This system is normally used for data exchange formats (DEFs) like PNG [41] or HDF [26] which have functionality similar to MDD compression but are usually not as generic, e.g. PNG can only be applied to 2D MDD with a small subset of base types. tilecompother is the only class of the compression engine that differs on client and server: whereas on the server, the `compress()` and `decompress()` methods are mapped to the DEF's `encode()` and `decode()` methods to unify MDD compression and DEFs in a common interface, they're dummies on the client to avoid burdening client applications with the various external libraries required for the DEF convertors[9].

tilecompstream: No transformation.

This class represents the simplest form of tile compression which does not perform

[9] In $\mathcal{RasDaMan}$ there are DEF convertors for BMP, JPEG, PNG, TIFF, HDF and VFF, some of which require external conversion libraries.

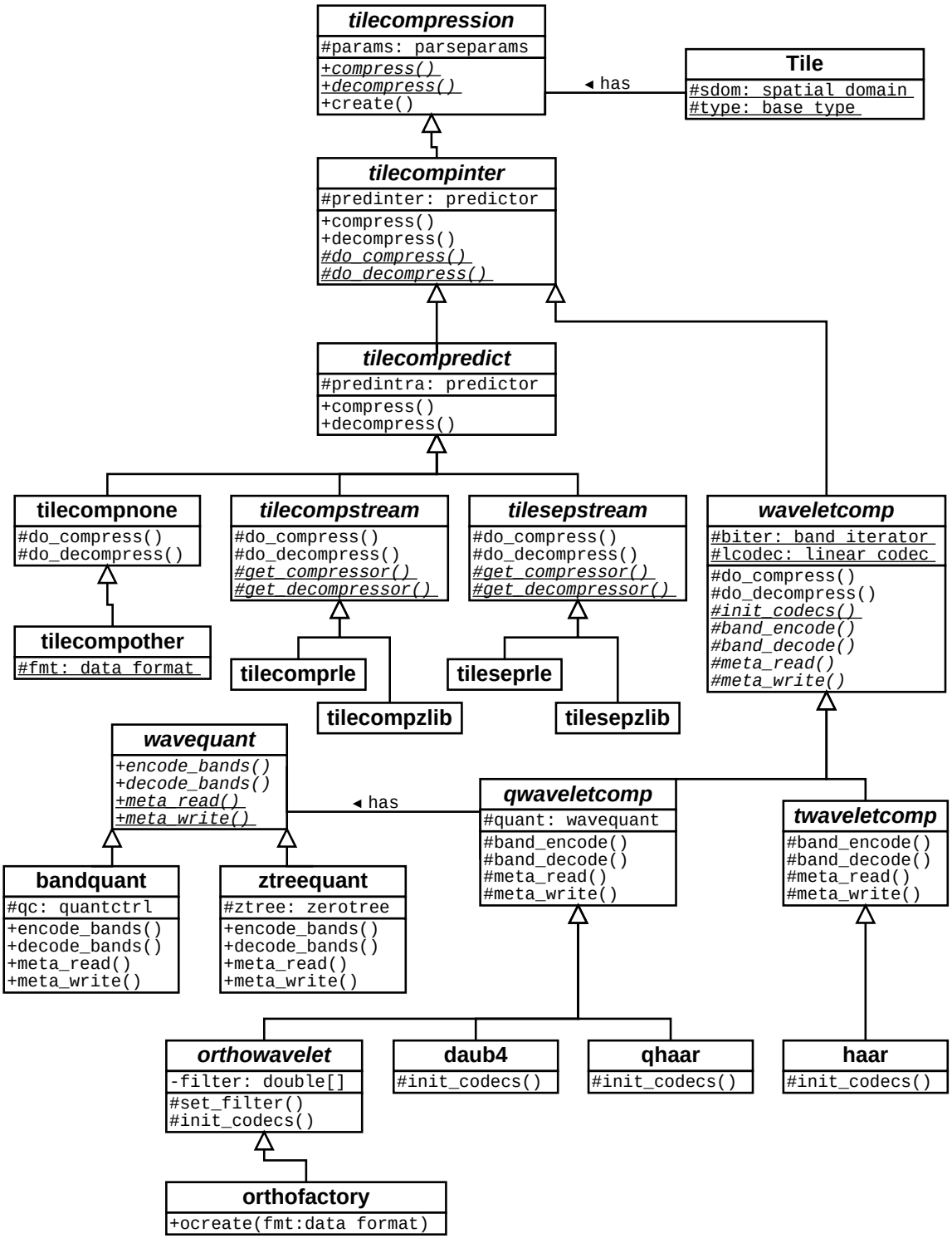


Figure 3.4: The tilecompression class hierarchy

This figure shows an outline of the class hierarchy of the top (= transformation) layer in UML *notation with* tilecompression *as the root class. Only a small subset of the available methods and member variables is shown for readability.*

any transformation nor uses any MDD properties but merely feeds the linearized tile data into an object of the compression layer; because the data belonging to structured base types is stored interleaved in $\mathcal{R}as\mathcal{D}a\mathcal{M}an$ (see section 2.4), data belonging to different channels is also mixed. Since all methods in the compression layer are accessible through a common interface, all functionality but the creation of suitable (de)compression streams can be provided in tilecompstream scope. The streams are obtained by derived classes through a `get_compressor()` / `get_decompressor()` interface which is implemented in the child classes tilecomprle and tilecompzlib. Arithmetic coding is not used in this context, because it usually only performs better than *ZLib* on special, decorrelated alphabets like the zerotree one (see section 3.4.3).

tilesepstream: Base type transformation.

This class represents a more sophisticated type of tile compression which exploits the semantics of structured base types and compresses the values of each channel separately. For instance an RGB image would be processed by first compressing the values belonging to the red channel, next the green and finally the blue channel. The output of the individual passes is then concatenated to form the final compressed tile data. Apart from the separation of base type values, this class works like tilecompstream and also uses the same interface to obtain (de)compression streams. In case the tile has an atomic base type, tilecompstream and tilesepstream are obviously equivalent. For structured base types, tilesepstream is usually more efficient because data belonging to the same channel is typically correlated stronger than data at the same position across channels, but there are exceptions; for instance the trivial case of a greyscale image stored as an RGB image where at all positions the values of all channels are identical. This situation is also addressed by predictors which will be discussed in more detail in section 3.5.

waveletcomp: Wavelet compression.

This is the most advanced compression method which exploits both the semantics of structured base types as well as spatial correlations. Wavelets are base functions with certain properties like compact or near compact support, which are used to analyse the input data at various resolutions. Wavelet transformations are related to the well known *Fourier* transformation[10], but because wavelets have compact support, they allow localized analysis without needing an additional window function like the 8×8 blocks in JPEG. The wavelet coefficients resulting from this base transformation are usually very sparse or of negligible amplitude and can be decimated considerably with little impact on the quality of the reconstructed signal. This property has made wavelet transformations the current state-of-the-art in image compression and because MDD share many properties with images which are attractive for compression, wavelets were also chosen in this MDD compression engine, in particular for lossy compression.

[10]The discrete cosine transformation used in JPEG is also strongly related to the Fourier transformation.

3.3 Wavelets and Multiresolution Analysis

In recent years, wavelets have received a huge amount of attention, especially in the fields of data compression [52, 55, 45, 28, 14, 56], computer graphics [23] and numerical mathematics [32, 25, 66], but have also found application in data mining problems [59]. Wavelets are base functions with special properties which we will describe in more depth in section 3.3.2. What's commonly referred to as *wavelet transformation* is actually a *multiresolution analysis* with wavelets, i.e. representing a signal as a superposition of weighted base functions at different scales and translations. The coefficients resulting from such a base transformation can be used to reconstruct the original signal during the *multiresolution synthesis* by calculating the sum of these coefficients multiplied with the base function they correspond to. The idea of the multiresolution analysis is to start analysing the signal at a certain coarse scale, resulting in a rough approximation of the signal, then to apply the analysis at the next finer resolution to the difference between the original signal and its coarse approximation and recursively continuing up to a specific finest resolution. This corresponds to applying a low-pass filter to obtain the coarse representation and a high-pass filter to obtain the difference, then recursively doing the same for the difference signal. A major advantage of using wavelets rather than arbitrary base functions for this procedure is the existence of fast analysis/synthesis algorithms which operate on a neighbouring resolution level rather than the signal resolution. The fast analysis algorithm, for instance, starts with the signal c_J at the finest resolution with n samples, calculates average and detail signals c_{J-1} and d_{J-1} from this, where the number of samples in c_{J-1} plus the number of samples in d_{J-1} is n, and recursively operates on c_{J-1} with fewer (typically $\frac{n}{2}$) samples, to calculate c_{J-2} and d_{J-2} etc.

Provided the signal is smooth, the difference between the approximations at consecutive resolution levels will be small, allowing for more efficient compression than the original signal. In many cases, the majority of the wavelet coefficients will be either zero or close enough to it to allow setting them to zero without a noticeable degradation of the signal reconstructed by the multiresolution synthesis. The goal of using wavelet transformations in the compression context is therefore to find an equivalent (lossless) or similar (lossy) representation of the data which compresses better than the original data.

3.3.1 Wavelet Examples

Before addressing some mathematical background of wavelets in section 3.3.2, we will have a look at two examples which illustrate how a multiresolution analysis with wavelets works in principle. In both examples, Haar wavelet transformations will be applied recursively over several hierarchical levels. As will be shown in section 3.3.2, applying a discrete wavelet transformation corresponds to folding the data with matching low-pass and high-pass filters, which for Haar wavelets have the filter coefficients $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$ (low-pass) and $(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$ (high-pass). Haar wavelets (see also figure 3.9) were chosen due to their simplicity: the filter is short (which is important for the numerical example) and easy to understand because, apart from the normalization, the low-pass filter calculates the arithmetic average

and the high-pass filter the difference from this average. Other wavelet filters are more complex and can be highly irregular, such as the orthogonal Daubechies series, some of which are depicted in figure 3.12 on page 54.

3.3.1.1 Numerical Example, 1D

The first example is a short 1D sequence of numbers. The original data was chosen to be the following sequence of 16 numbers which is relatively smooth[11] in accordance with the data model wavelets are targetted at:

10	16	13	9	0	-5	-2	4	0	3	8	12	10	15	20	30

In the first pass, pairs of consecutive values starting at even positions are folded with the filter coefficients; the response of the low-pass filter ($c_i = \frac{1}{\sqrt{2}}(v_{2i} + v_{2i+1})$) is then stored in the first half and the response of the high-pass filter ($d_i = \frac{1}{\sqrt{2}}(v_{2i} - v_{2i+1})$) in the second half of the data interval.

c3 (averages level 3)								d3 (detail level 3)							
$\frac{26}{\sqrt{2}}$	$\frac{22}{\sqrt{2}}$	$\frac{-5}{\sqrt{2}}$	$\frac{2}{\sqrt{2}}$	$\frac{3}{\sqrt{2}}$	$\frac{20}{\sqrt{2}}$	$\frac{25}{\sqrt{2}}$	$\frac{50}{\sqrt{2}}$	$\frac{-6}{\sqrt{2}}$	$\frac{4}{\sqrt{2}}$	$\frac{5}{\sqrt{2}}$	$\frac{-6}{\sqrt{2}}$	$\frac{-3}{\sqrt{2}}$	$\frac{-4}{\sqrt{2}}$	$\frac{-5}{\sqrt{2}}$	$\frac{-10}{\sqrt{2}}$

Now the multiresolution aspect comes into play as the same transformation is applied recursively to the averages that were just calculated (`c3`); the detail coefficients on this level remain unchanged for the remainder of the multiresolution analysis:

c2				d2				d3							
$\frac{48}{2}$	$\frac{-3}{2}$	$\frac{23}{2}$	$\frac{75}{2}$	$\frac{4}{2}$	$\frac{-7}{2}$	$\frac{-17}{2}$	$\frac{-25}{2}$	$\frac{-6}{\sqrt{2}}$	$\frac{4}{\sqrt{2}}$	$\frac{5}{\sqrt{2}}$	$\frac{-6}{\sqrt{2}}$	$\frac{-3}{\sqrt{2}}$	$\frac{-4}{\sqrt{2}}$	$\frac{-5}{\sqrt{2}}$	$\frac{-10}{\sqrt{2}}$

Again, the algorithm is applied recursively to the coarsest averages; this can be continued twice more before we arrive at one global average value:

c1		d1		d2				d3							
$\frac{45}{2\sqrt{2}}$	$\frac{98}{2\sqrt{2}}$	$\frac{51}{2\sqrt{2}}$	$\frac{-52}{2\sqrt{2}}$	$\frac{4}{2}$	$\frac{-7}{2}$	$\frac{-17}{2}$	$\frac{-25}{2}$	$\frac{-6}{\sqrt{2}}$	$\frac{4}{\sqrt{2}}$	$\frac{5}{\sqrt{2}}$	$\frac{-6}{\sqrt{2}}$	$\frac{-3}{\sqrt{2}}$	$\frac{-4}{\sqrt{2}}$	$\frac{-5}{\sqrt{2}}$	$\frac{-10}{\sqrt{2}}$

c0	d0	d1		d2				d3							
$\frac{143}{4}$	$\frac{-53}{4}$	$\frac{51}{2\sqrt{2}}$	$\frac{-52}{2\sqrt{2}}$	$\frac{4}{2}$	$\frac{-7}{2}$	$\frac{-17}{2}$	$\frac{-25}{2}$	$\frac{-6}{\sqrt{2}}$	$\frac{4}{\sqrt{2}}$	$\frac{5}{\sqrt{2}}$	$\frac{-6}{\sqrt{2}}$	$\frac{-3}{\sqrt{2}}$	$\frac{-4}{\sqrt{2}}$	$\frac{-5}{\sqrt{2}}$	$\frac{-10}{\sqrt{2}}$

At this point, the multiresolution analysis of the original data is complete and no coarser scale levels are possible. Note that the general trend is that the average coefficients are larger than the detail coefficients and that the coefficients' magnitude increases with the depth of the hierarchical recursion, so the maximum absolute value is ≈ 7.07107 in `d3`, 12.5 in `d2`, ≈ 18.38478 in `d1`, 13.25 in `d0` (the only exception of the trend) and 35.75 in `c0`. This is typical for smooth data and the reason why wavelet transformations can improve the compression rate of this kind of data considerably.

[11] i.e. the maximum difference between consecutive numbers (10) is small compared to the data's numeric range (35).

3.3.1.2 Image Coding Example, 2D

In the second example, shown in figures 3.5 – 3.7, Haar wavelet transformations were applied to the well-known *Lena* image [33] in successive horizontal and vertical passes over two resolution levels. This is the standard multidimensional extension of wavelet transformations taken in image compression and will be motivated (and generalized) in the implementation-biased section 3.3.3.

Detail coefficients are visualized such that zero corresponds to white and the larger the absolute values the darker the pixels in the detail bands; therefore a mostly white detail band means small detail coefficients. Note how sharp edges in the original image result in dark detail coefficients (large absolute values), whereas areas with uniform colour or smooth colour gradients result in detail coefficients close to zero. Because the detail bands in this example are dominated by small values, their contrast is low, therefore this example closes with an image where the detail bands were rescaled to use the full greyscale range in figure 3.8, which illustrates how wavelet compression achieves its substantial compression rates by filtering out this noise.

3.3.2 Mathematical Background

In this section, some mathematical background on wavelets will be given, mostly based on [15]. Wavelet mathematics is a highly complex field, however, so only an overview of the most important aspects will be given here; the interested reader is referred to [15, 18] for a profound introduction to wavelets.

Wavelet transformation is related to the well known Fourier transformation which has been a standard signal analysis procedure for many years. The Fourier transformation of a function f

$$(\mathcal{F}f)(\xi) := \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-ix\xi} f(x)\,\mathrm{d}x \tag{3.1}$$

performs a base transformation of the function using the complex exponential functions $b_\xi(x) = e^{ix\xi} = \cos(x\xi) + i\sin(x\xi)$ as base functions, thereby transforming the signal into its frequency spectrum where $(\mathcal{F}f)(\xi)$ is the Fourier coefficient for frequency ξ. Thus a function is expressed as a superposition of complex sine and cosine functions at various frequencies. This is the reason why the Fourier transformation is often used in signal processing to analyse the frequency spectrum of a signal. The inverse Fourier transformation

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{ix\xi} (\mathcal{F}f)(\xi)\,\mathrm{d}\xi \tag{3.2}$$

reconstructs the original signal from its Fourier coefficients. An important observation is that sharp edges in the input signal result in Fourier coefficients for high frequencies whereas a smooth input signal results in a concentration at low frequencies and zero or near zero

Figure 3.5: Wavelet transformation example: original image
This is the original Lena *image with a spatial extent* 512×512 *in 256 grey levels (i.e. 8 bits per pixel).*

coefficients for high frequencies. Because the base functions $b_\xi(x)$ have infinite support[12], the Fourier transformation is not localized, i.e. changing any value of the input signal will change *all* Fourier coefficients. This property is often not desirable and one way to deal with it is folding $f(x)$ with a *window function* $g(x)$ before the Fourier transformation, i.e. $\int_{-\infty}^{\infty} e^{-ix\xi} f(x)g(x-\xi)\,\mathrm{d}x$ to achieve localization to the support of $g(x-\xi)$. For example in JPEG the window function isolates 8×8 pixels before applying the DCT (another relative of the Fourier transformation). However, since g is usually a constant function, this approach doesn't allow analysing the input signal both in areas of rapid change and in areas of little change efficiently.

Wavelet transformations are similar to the Fourier transformation in that they perform a base transformation of the signal, but there is a large number of wavelet functions which ideally have compact support or are almost zero outside of a finite interval. Wavelet trans-

[12]The support of a function $f(x)$ is the interval $[x_0, x_1] : f(x)$ defined and $\neq 0$ for $x_0 \leq x \leq x_1$

Figure 3.6: Wavelet transformation example: finest level
To the left, the image from figure 3.5 is shown after applying a wavelet transformation in horizontal direction; this partitioned the image into two bands*, with the average coefficients in the left half and the detail coefficients in the right half. The image to the right resulted from applying vertical wavelet transformations to the left image; this operation divides each of the two bands in the left image into two further bands, resulting in a total of four bands: the total average to the top left, horizontal detail and vertical average to the top right, horizontal average and vertical detail to the bottom left and total detail to the bottom right. Now let the letters* `c` *and* `d` *represent average and detail coefficients along a given direction; using the convention that the horizontal coefficient type comes before the vertical one and appending the number of the resolution level the band belongs to (starting from 3 and decreasing with coarseness), we can label these bands as* `cc3`*,* `cd3`*,* `dc3` *and* `dd3` *respectively. See also figure 3.13 on page 55.*

formations are based on a *mother wavelet* $\psi(t)$ (the corresponding "mother" function of the Fourier transformation is the complex exponential function e^{-ix}), scaled and translated versions of which are used in the actual transformation. Three elementary properties the mother wavelet must have are

$$\int_{-\infty}^{\infty} \psi(t)\,\mathrm{d}t = 0 \quad \text{(zero-mean)} \tag{3.3}$$

$$\int_{-\infty}^{\infty} \psi(t)\overline{\psi(t)}\,\mathrm{d}t = 1 \quad \text{(normalization)} \tag{3.4}$$

$$\int_{-\infty}^{\infty} \frac{|(\mathcal{F}\psi)(t)|^2}{|t|}\,\mathrm{d}t = C_\psi < \infty \quad \text{(reversability)}, \tag{3.5}$$

Figure 3.7: Wavelet transformation example: next coarser level
The image to the left resulted from applying a horizontal wavelet transformation to the total averages of the next finer level, i.e. the upper left band (`cc3`) of the right image in figure 3.6; the other bands of this image (`cd3`, `dc3`, `dd3`) are not transformed. As in the previous case, the image to the right displays the result of the corresponding vertical transformation, leading to seven bands `cc2`, `cd2`, `dc2`, `dd2` which replace `cc3`, and the bands of the next finer level that were untransformed at this level: `cd3`, `dc3` and `dd3`.

where equation (3.5) places restrictions on the Fourier transformation of ψ, which is important regarding the existence of an inverse wavelet transformation [15, 18].

Because of the (near) compact support of the mother wavelet, it is obvious that the mother wavelet alone can't be used to analyse any given input signal, because those parts of the signal that fall outside the support would be ignored. Therefore, translated versions of the mother wavelet must be used: $\psi_b(t) = \psi(t-b)$, $b \in \mathbb{R}$. Adaption to areas of rapid change is done by using the mother wavelet at various resolution levels (*scales*), which leads to the set of base functions

$$\psi_b^a(t) = |a|^{-\frac{1}{2}} \psi\left(\frac{t-b}{a}\right),\ a, b \in \mathbb{R}, \tag{3.6}$$

which is a times as wide as the mother wavelet and shifted by $\frac{b}{a}$ relative to it. Using these scaled and translated versions of the mother wavelet, the continuous wavelet transformation has the form

$$(\mathcal{W}f)(a,b) = \int_{-\infty}^{\infty} f(t)\psi_b^a(t)\,\mathrm{d}t, \tag{3.7}$$

which is used to calculate the wavelet coefficients $(\mathcal{W}f)(a,b)$ for all resolutions a and translations b; note that equation (3.7) is of a form very similar to the Fourier transformation

Figure 3.8: Wavelet transformation example: enhanced detail bands
This image shows the right image of figure 3.7 where the values in the detail bands were rescaled to cover the full greyscale range. Note that this has a particularily large effect on the `dd3` *band at the bottom right which was almost entirely white without the rescaling (i.e. small detail coefficients) and now shows a substantial amount of random noise. Because these coefficients are small, they have little influence on the image quality and can be ignored, and because (truly) random data doesn't compress at all, this modification seriously improves the compression rate; this is how wavelet compression techniques achieve their superior performance.*

in equation (3.1). An inverse transformation for a given mother wavelet ψ exists if the mother wavelet satisfies the conditions in equations (3.3)-(3.5) and has the form

$$f(x) = \frac{1}{C_\psi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (\mathcal{W}f)(a,b) \frac{\psi_b^a(x)}{a^2} \,\mathrm{d}a \,\mathrm{d}b. \tag{3.8}$$

In order to process discrete signals stored in a computer, the discrete wavelet transformation is needed which essentially requires replacing the integrals with sums, resulting in a countable number of wavelet coefficients $(\mathcal{W}f)(a,b)$, $a,b \in \mathbb{Z}$. However, a problem with this representation of the wavelet transform is that it turns a 1D signal $f(x)$ into a 2D signal $(\mathcal{W}f)(a,b)$ and thereby expands the data rather than deflate it. But as we will see, by exploiting dependencies between the wavelet coefficients of neighbouring resolution levels in a multiresolution analysis, the number of wavelet coefficients can be restricted to the number of samples in the input signal.

So far, arbitrary values were possible for the scale and translation parameters. Typically only integer powers of 2 are used, i.e. $(a,b) = (\frac{1}{2^j}, \frac{k}{2^j})$, resulting in the base functions $\psi_{j,k}(t) = 2^{j/2}\psi(2^j t - k)$, $j,k \in \mathbb{Z}$, an example of which can be seen in figure 3.9.

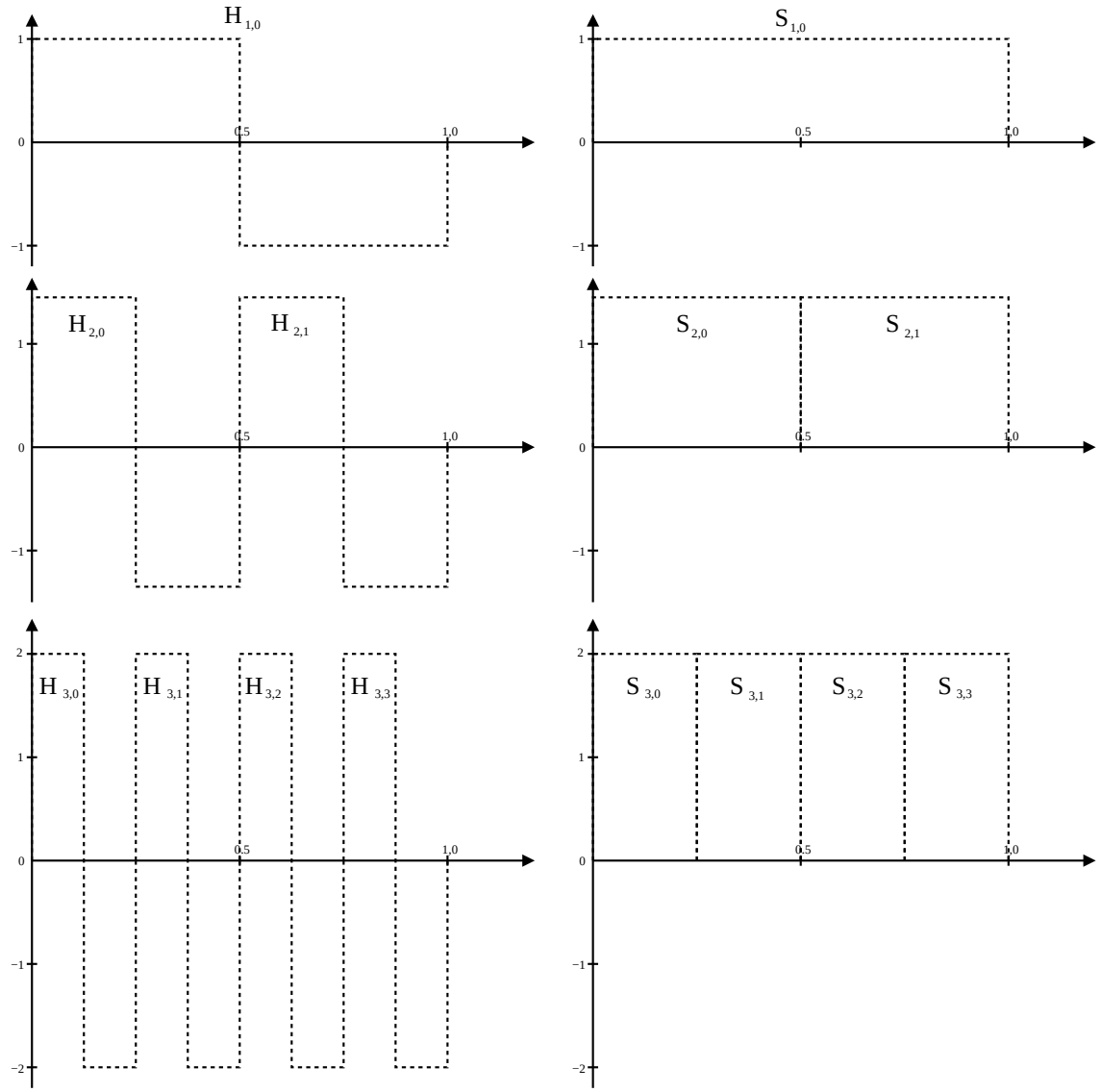


Figure 3.9: Multiresolution Haar wavelets

This figure shows some Haar wavelets $H_{i,j}$ at different scales (i) and translations (j) and the corresponding scaling functions $S_{i,j}$, as used in a multiresolution analysis. The scale factor between successive levels is $\frac{1}{2}$, i.e. the wavelets at the next higher scale level $i+1$ are half as wide and have $\sqrt{2}$ as large an amplitude (to satisfy the normalization equation (3.4)) as the ones at the previous level i.

We will now examine the dependencies of wavelet coefficients at different scale levels in the L^2 space of square-integrable functions[13], leading to the multiresolution wavelet analysis. The $\psi_{j,k}(t)$ with constant j (i.e. constant scale) and $k \in \mathbb{Z}$ span a subspace W_j of

[13]This is a necessary restriction for base transformations like Fourier- or wavelet transformations.

L^2, i.e. there are functions $f_j \in L^2$ which can be represented exactly as linear combinations of the base functions in W_j: $f_j \in W_j \Leftrightarrow \exists\,(s_k) : f_j(x) = \sum_{k=-\infty}^{\infty} s_k \psi_{j,k}(x)$. The subspaces W_j converge towards L^2 and can be used to approximate any function $f \in L^2$ with arbitrary precision, i.e. for each $\varepsilon > 0$ there exists a j such that $\|f - f_j\| < \varepsilon$; furthermore, the finer the scale level j, the smaller the approximation error gets, so naturally if $\|f - f_j\| < \varepsilon \Rightarrow \|f - f_{j'}\| < \varepsilon,\ j' \geq j$.

For the remainder of this section we will assume *orthogonal* mother wavelets (i.e. $\int_{-\infty}^{\infty} \psi_{j,k}(t)\overline{\psi_{l,m}(t)}\,\mathrm{d}t = \delta_{j,l}\delta_{k,m}$; this is only 1 if both $j = l$ and $k = m$, i.e. $\int_{-\infty}^{\infty} \psi_{j,k}(t)\overline{\psi_{j,k}(t)}\,\mathrm{d}t$: all wavelets generated out of an orthogonal mother wavelet are orthogonal to each other and normalized to 1), which means that the subspaces W_j are mutually orthogonal: $W_j \perp W_k \quad \forall j \neq k$. Now consider another sequence of subspaces $V_j = \bigcup_{k=-\infty}^{j-1} W_k$ where the resolution also increases with j. The important consequences of this definition are

$$V_{j+1} = V_j \cup W_j \tag{3.9}$$

$$V_j \subset V_{j+1} \tag{3.10}$$

$$\lim_{j\to\infty} V_j = L^2. \tag{3.11}$$

In analogy to the W_j which are spanned by the $\psi_{j,k}, k \in \mathbb{Z}$ generated out of a mother wavelet ψ, let's now assume the V_j are spanned by $\phi_{j,k}, k \in \mathbb{Z}$ generated out of a *scaling function* ϕ in exactly the same way as the $\psi_{j,k}$, i.e. $\phi_{j,k}(t) = 2^{j/2}\phi(2^j t - k)$. Details on the existence of such a scaling function are beyond the scope of this work, but can be found in the standard literature on wavelet mathematics [15, 18]. Provided matching ψ and ϕ exist, they can be used to perform a multiresolution analysis.

In a multiresolution analysis, a signal is decomposed into average and detail coefficients for various scale levels, using correlations between the coefficients of successive scale levels to calculate the wavelet coefficients of the next coarser level $j - 1$ during analysis and those of the next finer level $j + 1$ during synthesis from the coefficients at level j; finest and coarsests resolution in this operation are application-dependent and will be addressed at a later point in this section. Average coefficients are those belonging to the $\phi_{j,k}$ generated out of the scaling function and detail coefficients are those belonging to the $\psi_{j,k}$ generated out of the mother wavelet; the term "wavelet coefficients" is used as a shorter alias for both average and detail coefficients. The remainder of this section covers the decomposition of a signal into wavelet coefficients during analysis and the reconstruction of the signal from its wavelet coefficients. We will first look at how the $\phi_{j,k}$ and $\psi_{j,k}$ at different scale levels are correlated and use the result to correlate the wavelet coefficients themselves across different scale levels.

Because of equation (3.9), all $\phi_{j-1,k}$ and $\psi_{j-1,k}$ are contained in V_j and can therefore be expressed as linear combinations of the $\phi_{j,k}$ which span V_j, i.e. there exist unique sequences (p_i) and (q_i) such that

$$\begin{aligned} \phi_{j-1,0}(x) &= \sum_{i=-\infty}^{\infty} p_i \phi_{j,i}(x) = 2^{j/2} \sum_{i=-\infty}^{\infty} p_i \phi(2^j x - i) \\ \psi_{j-1,0}(x) &= \sum_{i=-\infty}^{\infty} q_i \phi_{j,i}(x) = 2^{j/2} \sum_{i=-\infty}^{\infty} q_i \phi(2^j x - i) \end{aligned} \tag{3.12}$$

The sequences (p_i) and (q_i) correspond to the analysis filter coefficients and their actual relation with the wavelet filter coefficients listed in the standard literature [18] will be explained towards the end of this section. We now extend equation (3.12) to also include translations $\neq 0$ by expressing $\phi_{j-1,k}(x)$ as $\phi_{j-1,0}(x')$ as follows:

$$\phi_{j-1,k}(x) = 2^{(j-1)/2}\phi(2^{j-1}x - k) =: 2^{(j-1)/2}\phi(2^{j-1}x') = \phi_{j-1,0}(x'),$$

and analogously for $\psi_{j-1,k}(x)$. That means $2^{j-1}x - k = 2^{j-1}x'$ must hold, or $x' = x - 2^{1-j}k$. We can then calculate $\phi_{j-1,k}(x)$ and $\psi_{j-1,k}(x)$ using equations (3.12) and replacing x with x', which gives us (using the index transformation $i' = i + 2k$ in the fourth step)

$$\begin{aligned} \phi_{j-1,k}(x) &= \phi_{j-1,0}(x') \\ &= 2^{j/2} \sum_{i=-\infty}^{\infty} p_i \phi(2^j(x - 2^{1-j}k) - i) \\ &= 2^{j/2} \sum_{i=-\infty}^{\infty} p_i \phi(2^j x - 2k - i) \\ &= 2^{j/2} \sum_{i'=-\infty}^{\infty} p_{i'-2k} \phi(2^j x - i') \\ &= \sum_{i=-\infty}^{\infty} p_{i-2k} \phi_{j,i}(x) \end{aligned} \tag{3.13}$$

$$\psi_{j-1,k}(x) = \psi_{j-1,0}(x') = \ldots = \sum_{i=-\infty}^{\infty} q_{i-2k} \phi_{j,i}(x). \tag{3.14}$$

Thus equations (3.13) and (3.14) allow expressing scaling functions and wavelets of the next coarser level by using the scaling functions of the nearest finer level. The sequences (p_i) and (q_i) are constant across all scale levels and unique for each pair (ϕ, ψ) of scaling function and mother wavelet. Due to the (near) compact support of wavelets and scaling functions, the sequences (p_i) and (q_i) are zero everywhere outside of a small window around $i = 0$.

For the other direction (synthesis) we use the fact that $V_j = V_{j-1} \cup W_{j-1}$, i.e. any $\phi_{j,k}$ can be expressed as a linear combination of $\phi_{j-1,k}$ and $\psi_{j-1,k}$ using unique sequences (a_i) and (b_i):

$$\phi_{j,0}(x) = \sum_{i=-\infty}^{\infty} (a_{2i}\phi_{j-1,i}(x) + b_{2i}\psi_{j-1,i}(x))$$

$$= 2^{(j-1)/2} \sum_{i=-\infty}^{\infty} \left(a_{2i}\phi(2^{j-1}x - i) + b_{2i}\psi(2^{j-1}x - i) \right). \tag{3.15}$$

In analogy to the sequences (p_i) and (q_i), the sequences (a_i) and (b_i) correspond to the synthesis filter coefficients and their relation with the wavelet coefficients will be explained at the same point later in this section. The reason for using the index $2i$ rather than i is to avoid non-integer indices in the following calculations. Non-integer indices of sequences are not a problem as such, but they're unusual and therefore are rescaled to integers. We can extend equation (3.15) to include translations again by using $\phi_{j,k}(x) =: \phi_{j,0}(x')$, which leads to $x' = x - 2^{-j}k$. Replacing x in equation (3.15) with this x' and using the index mapping $i' = \frac{k}{2} + i$ in the fourth step we obtain

$$\begin{aligned}
\phi_{j,k}(x) &= \phi_{j,0}(x') \\
&= 2^{(j-1)/2} \sum_{i=-\infty}^{\infty} \left(a_{2i}\phi(2^{j-1}(x - 2^{-j}k) - i) + b_{2i}\psi(2^{j-1}(x - 2^{-j}k) - i) \right) \\
&= 2^{(j-1)/2} \sum_{i=-\infty}^{\infty} \left(a_{2i}\phi(2^{j-1}x - \frac{k}{2} - i) + b_{2i}\psi(2^{j-1}x - \frac{k}{2} - i) \right) \\
&= 2^{(j-1)/2} \sum_{i'=-\infty}^{\infty} \left(a_{2i'-k}\phi(2^{j-1}x - i') + b_{2i'-k}\psi(2^{j-1}x - i') \right) \\
&= \sum_{i=-\infty}^{\infty} \left(a_{2i-k}\phi_{j-1,i}(x) + b_{2i-k}\psi_{j-1,i}(x) \right).
\end{aligned} \tag{3.16}$$

Therefore, equation (3.16) allows expressing the scaling function at the next finer level by using the scaling functions and wavelets of the nearest coarser level. Just like the sequences (p_i) and (q_i), the sequences (a_i) and (b_i) are constant across all scale levels, unique for each pair (ϕ, ψ) and zero everywhere outside a small window around $i = 0$.

We will now apply the correlations between the $\psi_{j,k}$ and $\phi_{j,k}$ we just found to correlate wavelet coefficients of a function $f(x) = \lim_{j\to\infty} f_j(x)$, $f_j \in V_j$ across different scale levels; any square-integrable function can be expressed this way provided j is large enough. Because of equation (3.9), any function $f_{j+1} \in V_{j+1}$ can be written as the sum of two functions $f_j \in V_j$ and $g_j \in W_j$ for all j. Because $f_j \in V_j$ and $g_j \in W_j$, there exist unique sequences (c_i^j), (d_i^j) such that

$$f_j(x) = \sum_{i=-\infty}^{\infty} c_i^j \phi_{j,i}(x) \tag{3.17}$$

$$g_j(x) = \sum_{i=-\infty}^{\infty} d_i^j \psi_{j,i}(x), \tag{3.18}$$

where c_i^j are the average coefficients and d_i^j are the detail coefficients, both at scale level j. Equation (3.17) is also true for f_{j+1}, however; we therefore get the following equations by substituting $\phi_{j+1,k}$ according to equation (3.16):

$$\begin{aligned} f_j(x) + g_j(x) &= f_{j+1}(x) \\ \sum_{i=-\infty}^{\infty} c_i^j \phi_{j,i}(x) + \sum_{i=-\infty}^{\infty} d_i^j \psi_{j,i}(x) &= \sum_{l=-\infty}^{\infty} c_l^{j+1} \phi_{j+1,l}(x) \\ &= \sum_{l=-\infty}^{\infty} c_l^{j+1} \sum_{i=-\infty}^{\infty} \left(a_{2i-l}\phi_{j,i}(x) + b_{2i-l}\psi_{j,i}(x)\right) \\ &= \sum_{i=-\infty}^{\infty} \left(\phi_{j,i}(x) \sum_{l=-\infty}^{\infty} c_l^{j+1} a_{2i-l} + \psi_{j,i}(x) \sum_{l=-\infty}^{\infty} c_l^{j+1} b_{2i-l}\right). \end{aligned} \tag{3.19}$$

Because $\phi_{j,i}(x)$ and $\psi_{j,i}(x)$ are orthogonal base functions, this equation can only be true if all their coefficients are equal, therefore we get the following equations for calculating the average and detail coefficients during analysis:

$$c_i^j = \sum_{l=-\infty}^{\infty} c_l^{j+1} a_{2i-l} \tag{3.20}$$

$$d_i^j = \sum_{l=-\infty}^{\infty} c_l^{j+1} b_{2i-l}. \tag{3.21}$$

We can obtain the inverse operations by starting from the same decomposition of $f_{j+1}(x)$ but substituting $\phi_{j,k}(x)$ and $\psi_{j,k}(x)$ according to equations (3.13) and (3.14):

$$\begin{aligned} f_{j+1}(x) &= f_j(x) + g_j(x) \\ \sum_{i=-\infty}^{\infty} c_i^{j+1} \phi_{j+1,i}(x) &= \sum_{l=-\infty}^{\infty} c_l^j \phi_{j,l}(x) + \sum_{l=-\infty}^{\infty} d_l^j \psi_{j,l}(x) \\ &= \sum_{l=-\infty}^{\infty} \left(c_l^j \sum_{i=-\infty}^{\infty} p_{i-2l} \phi_{j+1,i}(x) + d_l^j \sum_{i=-\infty}^{\infty} q_{i-2l} \phi_{j+1,i}(x)\right) \\ &= \sum_{i=-\infty}^{\infty} \phi_{j+1,i}(x) \sum_{l=-\infty}^{\infty} \left(c_l^j p_{i-2l} + d_l^j q_{i-2l}\right). \end{aligned} \tag{3.22}$$

Because the $\phi_{j,i}(x)$ are base functions, this equation can again only hold if all their coefficients are equal, leading to the synthesis equation

$$c_i^{j+1} = \sum_{l=-\infty}^{\infty} \left(c_l^j p_{i-2l} + d_l^j q_{i-2l}\right). \tag{3.23}$$

It is usually more convenient to write this equation as two equations for even and odd i, since different $p_\cdot$ and $q_\cdot$ are used in these cases:

$$c_{2i}^{j+1} = \sum_{l=-\infty}^{\infty} \left(c_l^j p_{2i-2l} + d_l^j q_{2i-2l} \right) \tag{3.24}$$

$$c_{2i+1}^{j+1} = \sum_{l=-\infty}^{\infty} \left(c_l^j p_{2i-2l+1} + d_l^j q_{2i-2l+1} \right). \tag{3.25}$$

Summarizing the results so far, equations (3.20) and (3.21) allow us to decompose a sequence (c_i) (= a discrete signal) into sequences of average and detail coefficients. Provided the sequence is non-trivial for a compact interval of indices only, and ignoring border effects, the number of average or detail coefficients is half the number of elements in the original sequence; this is a direct consequence of the (near) compact support of wavelet functions and the fact that the support of scaling functions and wavelets doubles with each coarser level. Finally, equation (3.23) provides the inverse operation to reconstruct a signal from its average and detail coefficients. Therefore these three equations allow performing the decomposition of a signal into multiresolution wavelet coefficients and back.

Provided we have the average coefficients at a finest level J, we can calculate average and detail coefficients at the next coarser level $J-1$, then recursively decompose the average coefficients at level $J-1$ into average and detail coefficients at level $J-2$ and so on until we reach a coarsest level J_0; J_0 is typically the level that contains only one average coefficient (since the number of average coefficients halves for each coarser level) or a finer level if a full decomposition is undesirable, e.g. for reasons of numerical stability (an issue that will play an important role in section 3.4). We store the average coefficients at the coarsest level only, but the detail coefficients at *all* scale levels. All other average coefficients are redundant since they can be calculated from the next coarser average and detail coefficients using equation (3.23). Because the number of coefficients halves for each coarser level, the number of coefficients that have to be processed in a multiresolution analysis is $\sum_{i=0}^{J-J_0} \frac{n}{2^i} \leq \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$, where n is the number of average coefficients at the finest level J. The multiresolution analysis is shown schematically in figure 3.10, where in each column equations (3.20) and (3.21) are applied.

$$\begin{array}{ccccccccc} & & d_k^{J-1} & & d_k^{J-2} & \cdots & d_k^{J_0+1} & & d_k^{J_0} \\ & \nearrow & & \nearrow & & \nearrow & & \nearrow & \\ c_k^J & \rightarrow & c_k^{J-1} & \rightarrow & \cdots & \rightarrow & c_k^{J_0+1} & \rightarrow & c_k^{J_0} \end{array}$$

Figure 3.10: Decomposing a signal into multiresolution wavelet coefficients

When the signal has to be reconstructed from its multiresolution wavelet coefficients during synthesis, this is done by starting with average and detail coefficients at the coarsest level J_0 and calculating the average coefficients for the next finer level J_0+1 using equation (3.23), then using the detail coefficients at level J_0+1 to calculate the average coefficients

at level $J_0 + 2$ and so on until we've reached the finest level J again where the original signal is reconstructed. Multiresolution synthesis is shown schematically in figure 3.11, where in each column equation (3.23) is applied.

$$\begin{array}{ccccccccc} d_k^{J_0} & & d_k^{J_0+1} & & \cdots & & d_k^{J-1} & & \\ & \searrow & & \searrow & & \searrow & & \searrow & \\ c_k^{J_0} & \rightarrow & c_k^{J_0+1} & \rightarrow & \cdots & \rightarrow & c_k^{J-1} & \rightarrow & c_k^J \end{array}$$

Figure 3.11: Reassembling a signal from its multiresolution wavelet coefficients

In case of a discrete input signal[14], the finest level is given naturally by the resolution of the input signal and the average coefficients at this level are identical to the signal values at the sample points.

The actual values of the four sequences (a_i), (b_i), (p_i) and (q_i) used in the multiresolution analysis and synthesis are still unspecified so far. For orthogonal wavelets, ϕ and ψ form a *quadrature mirror filter* (QMF) with an even number of filter coefficients h_i, $0 \leq i < 2N$. A QMF is a pair of low-pass and high-pass filters where the high-pass filter consists of the filter coefficients of the low-pass filter in inverse order and with alternating signs. All filter coefficients for orthogonal wavelets are normalized in the following way:

$$\sum_{i=0}^{2N-1} h_i = \sqrt{2}, \qquad \sum_{i=0}^{2N-1} h_i^2 = 1. \tag{3.26}$$

Many filter coefficients have been published in wavelet literature [18, 47] and are usually given for the scaling function rather than the mother wavelet. The mapping of these filter coefficients to the sequences (a_i), (b_i), (p_i) and (q_i) is closely linked to the algorithms used to calculate the filter coefficients, which are not covered in this work. For orthogonal wavelets, the analysis sequences (a_i) and (b_i) are typically mapped to the wavelet filter coefficients like this:

$$\left.\begin{array}{rcl} a_{-i} & = & h_i \\ b_{-i} & = & (-1)^i \cdot h_{2N-i-1} \end{array}\right\} \qquad 0 \leq i \leq 2N-1; \tag{3.27}$$

for all other i, a_{-i} and b_{-i} are 0. The mapping of the synthesis sequences (p_i) and (q_i) is less standardized due to the filter offset, but a typical mapping is the following for a filter offset of 1:

$$\left.\begin{array}{rcl} p_{-i} & = & h_{(2N-i-4) \bmod 2N} \\ q_{-i} & = & (-1)^i h_{(i+3) \bmod 2N} \end{array}\right\} \qquad -3 \leq i \leq 2N-4. \tag{3.28}$$

[14] Typically equidistant, but this is not strictly necessary for a multiresolution analysis; equidistant input signals allow the most efficient implementation, however.

Let's look at a 4-tap filter ($N = 2$) as an example. According to equations (3.20), (3.21) and (3.27) we get the following analysis equations (in matrix-vector notation for more clarity):

$$\begin{pmatrix} c_i^{j-1} \\ d_i^{j-1} \end{pmatrix} = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_3 & -h_2 & h_1 & -h_0 \end{pmatrix} \begin{pmatrix} c_{2i}^j \\ c_{2i+1}^j \\ c_{2i+2}^j \\ c_{2i+3}^j \end{pmatrix}.$$

The synthesis equations are then obtained by equations (3.24), (3.25) and (3.28) as

$$\begin{pmatrix} c_{2i}^j \\ c_{2i+1}^j \end{pmatrix} = \begin{pmatrix} h_2 & h_1 & h_0 & h_3 \\ h_3 & -h_0 & h_1 & -h_2 \end{pmatrix} \begin{pmatrix} c_{i-1}^{j-1} \\ d_{i-1}^{j-1} \\ c_i^{j-1} \\ d_i^{j-1} \end{pmatrix}.$$

The filter offset in this case is 1, because the average and detail coefficients used in synthesis start at $i - 1$. Other offsets are possible, depending on the type of wavelet used.

3.3.3 Wavelet Implementation Aspects

In the previous section, we covered some theoretical aspects of wavelet transformations, leading to the discrete wavelet transform as a QMF filter. Now we will concentrate on implementational issues of wavelet transformations in a multidimensional compression engine.

The discrete wavelet transformation introduced in the previous section is a filter applied to a 1D signal which we have to extend for the multidimensional case. The most commonly used approach in image compression is to apply 1D wavelet transformations to the rows of the image, storing the average coefficients in the first half and the detail coefficients in the second half of the row, thereby transforming a $2^n \times 2^n$ image into a $2^{n-1} \times 2^n$ image containing average coefficients and a $2^{n-1} \times 2^n$ image containing detail coefficients (see the left image in figure 3.6 on page 43); then 1D wavelet transformations are applied to the columns of these two images, resulting in four $2^{n-1} \times 2^{n-1}$ images (*bands*) containing all combinations of average (`c`) and detail (`d`) coefficients in the two directions (see the right image in figure 3.6). On the next coarser level, the transformations are applied to the band containing only average coefficients in both directions. This process is continued recursively until a coarsest level is reached, where the `cc` band is left. This has already been shown in the wavelet example figures 3.5–3.7 and can also be seen schematically in figure 3.13 on page 55 over three hierarchical levels.

For synthesis, this process is performed in the inverse order, using the synthesis filters: starting at the coarsest level, the wavelet synthesis is applied to the columns of the four subimages at the coarsest level (`cc1`, `cd1`, `dc1` and `dd1` in figure 3.13), thereby creating two $2^{n-1} \times 2^n$ images containing average and detail coefficients for the horizontal direction,

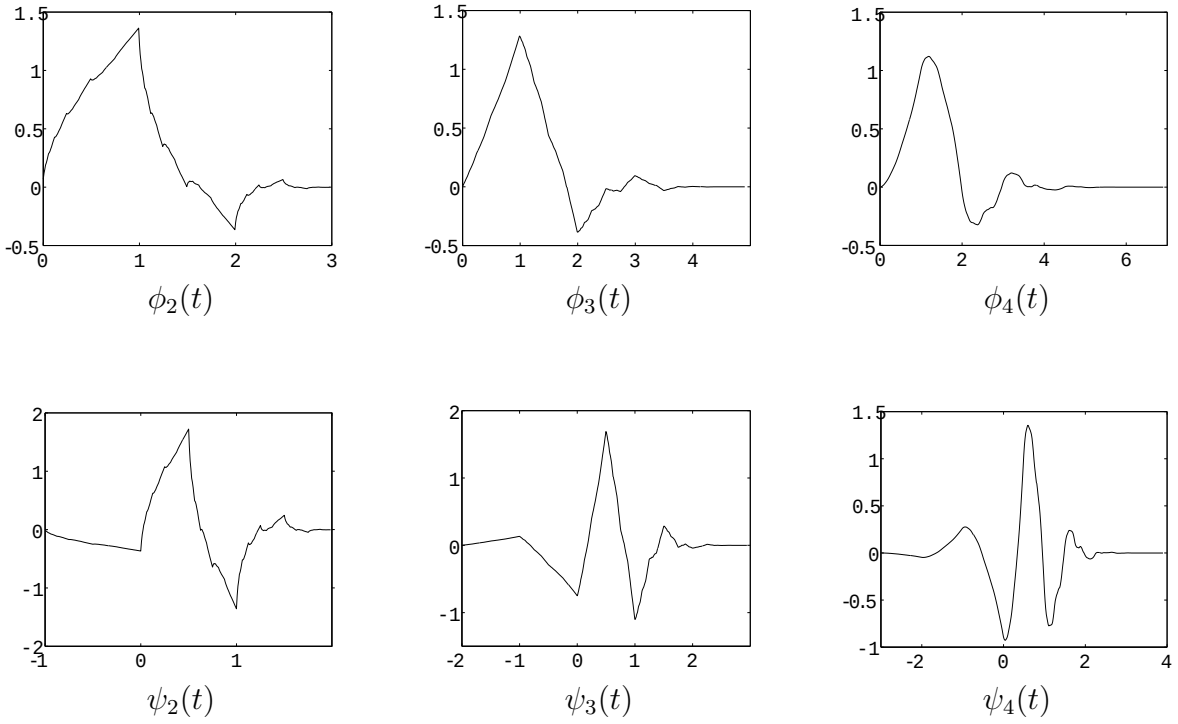


Figure 3.12: Daubechies wavelets and scaling functions
This figure shows the scaling functions (top row) and mother wavelets (bottom row) of the orthogonal Daubechies wavelet family with 2–4 vanishing moments (corresponding to 4–8 filter taps; the filter coefficients can be found in appendix B.1 on page 153). Figures taken from [35].

which are then synthesized to one $2^n \times 2^n$ image with a horizontal synthesis pass. This image is then used as the `cc` region for the next finer level, respectively the final image if there are no more finer levels.

An alternative approach is using multidimensional wavelet bases rather than separable ones, but that system is hard to adapt to the generic case; some theoretical considerations about multidimensional wavelet bases can be found in [18] and a concrete 2D example is in [56] (*Red-Black Wavelet*). Separable wavelet bases on the other hand are easy to extend for additional dimensions, i.e. applying 1D analysis filters along all dimensions during analysis and 1D synthesis filters along the dimensions in inverse order during synthesis. For D dimensions, this process partitions the data into 2^D bands on each scale level, which are tagged with identifiers in $\{\texttt{c,d}\}^D$, for example `c···cc` for the total average, `c···cd` for the averages in the first $D-1$ dimensions and detail in the last dimension etc. The *degree of detail* of a band is the number of dimensions for which the band holds detail information, which is the number of `d` tags appearing in the band's label; e.g. in the 3D case, band `ccc` has degree of detail 0, `dcc` and `cdc` have 1 and `ddd` has 3.

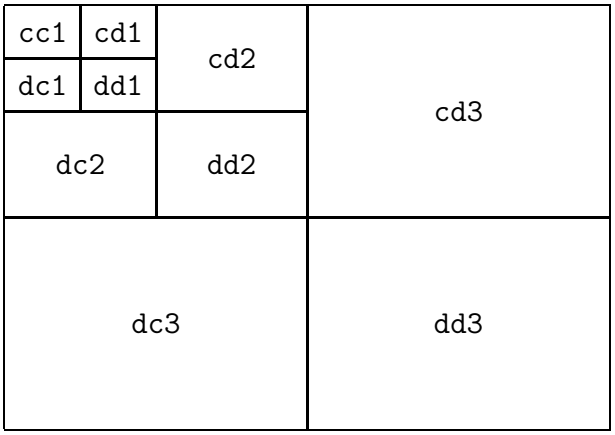


Figure 3.13: Wavelet transformations in 2D
This figure shows the commonly used approach of wavelet transformations in 2D over three hierarchical levels, as for instance in image compression. Applying horizontal and vertical transformations partitions the data on each hierarchical level into four bands representing all combinations of average and detail coefficients for the two dimensions. At the next coarser level, the transformations are applied to the `cc` *region thus obtained.*

The complexity C_0 of this D-dimensional wavelet transformation on a multidimensional array M with the spatial extent $w_1, \ldots, w_D$ at the finest hierarchical level is given as follows: applying the transformation along dimension i involves folding w_i values with the wavelet filter of length $|f|$, i.e. $w_i|f|$ multiplications/additions for each "line" parallel to dimension i through M (e.g. for each row or column in an image). The number of these lines is determined by the width of M in the other dimensions, i.e. $\prod_{j=1}^{i-1} w_j \prod_{j=i+1}^{D} w_j$, leading to a total of $|f|w_i \prod_{j=1}^{i-1} w_j \prod_{j=i+1}^{D} w_j = |f| \prod_{j=1}^{D} w_j$ multiplications/additions per dimension (which is independent of i and therefore constant for all dimensions). This results in a total complexity of

$$C_0(M) = \sum_{i=1}^{D} |f| \prod_{j=1}^{D} w_j = |f| D \prod_{j=1}^{D} w_j \tag{3.29}$$

multiplications/additions on the finest scale level, which is linear in the filter length, the number of dimensions and the number of cells ($= \prod_{j=1}^{D} w_j$). At each coarser level $l < 0$, the spatial extent is halved in all dimensions, leading to complexities $C_l(M) = |f| D \prod_{j=1}^{D} 2^l w_j$ and the upper limit of the complexity on all scale levels

$$C(M) = \sum_{l=-\infty}^{0} C_l(M) = |f|D \sum_{l=-\infty}^{0} \prod_{j=1}^{D} 2^l w_j = |f|D \prod_{j=1}^{D} w_j \sum_{l=0}^{\infty} \frac{1}{2^{lD}} = |f|D \frac{2^D}{2^D - 1} \prod_{j=1}^{D} w_j \tag{3.30}$$

using the correspondence $\sum_{i=0}^{\infty} \frac{1}{2^{iD}} = \frac{2^D}{2^D-1}$. An important result of this total complexity is that it still scales linearily rather than exponentially with the number of dimensions, since $\lim_{D\to\infty} \frac{2^D}{2^D-1} = 1$, thus making it an *efficient* transformation technique even in high-dimensional spaces.

Another important implementational aspect is the treatment of the boundaries during analysis and synthesis. Because the wavelet filters we have discussed so far have length $2N$, there is need for special boundary treatment in signals with odd length (on each scale level) or for filters with $N > 1$. The easiest solution for signals with odd length is to ignore the last sample in the transformation and store it explicitly at the end of the average values instead, thereby transforming a signal with $2n + 1$ samples into $n + 1$ $(= \lceil \frac{2n+1}{2} \rceil)$ average- and n $(= \lfloor \frac{2n+1}{2} \rfloor)$ detail coefficients. Appending the untransformed samples to the averages means that they will still be transformed by subsequent passes in other directions and on coarser scale levels without introducing singularities by boundary extensions. We will now examine how the width of the average band changes for this approach as we progress to coarser levels by looking at the 1D case. Assuming our original 1D data has length $|a^0| = \sum_{i=0}^{k} 2^i j_i,\ j_i \in \{0, 1\}$, we calculate the number $|a^l|$ of average coefficients after l coarsening steps

$$|a^{l+1}| = \left\lceil \frac{|a^l|}{2} \right\rceil = \begin{cases} \frac{|a^l|}{2} & |a^l| \text{ even} \\ \frac{|a^l|+1}{2} & |a^l| \text{ odd} \end{cases} \tag{3.31}$$

in the following way:

$$|a^l| = \sum_{i=l}^{k} 2^{i-l} j_i + r_l, \qquad r_0 = 0; \quad r_i = \left\lceil \frac{j_{i-1} + r_{i-1}}{2} \right\rceil, 1 \le i < k. \tag{3.32}$$

This can be shown by complete induction: the induction start $|a^0|$ is obviously correct; for the induction step $|a^l| \to |a^{l+1}|$ we get

$$|a^{l+1}| = \left\lceil \frac{|a^l|}{2} \right\rceil = \left\lceil \frac{1}{2} \left(\sum_{i=l}^{k} 2^{i-l} j_i + r_l \right) \right\rceil = \sum_{i=l+1}^{k} 2^{i-l-1} j_i + \left\lceil \frac{j_l + r_l}{2} \right\rceil = \sum_{i=l+1}^{k} 2^{i-(l+1)} j_i + r_{l+1}.$$

We can also formulate r_l in a non-recursive definition. First, we can prove by induction that all r_l are either 0 or 1: for r_0 it's true by definition; with $j_{l-1} \in \{0, 1\}$ and $r_{l-1} \in \{0, 1\}$ then naturally $\left\lceil \frac{j_{l-1}+r_{l-1}}{2} \right\rceil \in \{0, 1\}$ too. With the same mechanism we can show that $r_l = 0 \Leftrightarrow j_0 = \cdots = j_{l-1} = 0$: for $l = 0$ this is again true by definition and $r_{l+1} = \left\lceil \frac{j_l+r_l}{2} \right\rceil$ is either 0 if both j_l and r_l are $0 \Leftrightarrow j_0 = \cdots = j_{l-1} = j_l = 0$, or 1 if j_l or r_l differ from 0,

i.e. $\exists j_i \neq 0,\ 0 \leq i \leq l$, which proves the assumption. Therefore r_l can also be written in closed form as

$$r_l = \left\lceil \frac{1}{l} \sum_{i=0}^{l-1} j_i \right\rceil, \tag{3.33}$$

which is 0 if $j_0 = \cdots = j_{l-1} = 0$ and 1 otherwise for $0 \leq l \leq k$, just like the recursive definition. That means we can also express $|a^l|$ without recursion as

$$|a^l| = \sum_{i=l}^{k} 2^{i-l} j_i + \left\lceil \frac{1}{l} \sum_{i=0}^{l-1} j_i \right\rceil. \tag{3.34}$$

This sequence converges monotonously towards 1, as can best be seen from the original definition in equation (3.31), which states that $|a^{l+1}| \leq \frac{|a^l|+1}{2}$, or in other words

$$|a^{l+1}| \leq |a^l| \Leftrightarrow \frac{|a^l|+1}{2} \leq |a^l| \Leftrightarrow |a^l| \geq 1.$$

Since the multiresolution analysis has to stop for $|a^l| = 1$ at the latest, this is always true which proves the monotonous convergence of $|a^l|$ towards 1.

As for the boundary treatment of longer filters, there are several possible approaches to simulate the $2(N-1)$ missing samples (which would also be alternatives to the handling of odd-length samples):

1. set the missing samples to zero;
2. repeat the last sample;
3. mirror the final samples (*symmetric extension*);
4. periodic, i.e. concatenate the first $2(N-1)$ samples.
5. special boundary wavelets with shorter filter length;

The ideal border extension avoids singularities which would result in large detail coefficients. Only items 3 and 5 guarantee a certain degree of smoothness in the extended signal, independent of the original signal. However, alternatives 1–3 have the major disadvantage that they require storing the extra boundary coefficients for synthesis. Alternative 5 is quite complicated to do, especially for long wavelet filters, and is rarely implemented. Periodic extension (4), on the other hand, maintains one wavelet filter throughout the transformation and doesn't require storing more coefficients than samples in the original signal because the boundary coefficients can also be obtained by periodic extension (of both average and detail coefficients respectively). It is therefore a very frequently used boundary extension, despite potential singularities at the boundary, and was chosen for this compression engine as well; alternative boundary treatment is an issue for future work.

Another important question when implementing a generic wavelet engine is what data types to use for the arithmetic operations (folding with the filter coefficients). Most literature on wavelet compression concentrates on image or video compression where the arrays being transformed have the cell type `byte`, which allows using a conventional 32 bit type for the arithmetic of some special wavelet types[15], thereby doing the wavelet transformations in pure integer arithmetic. With the exception of Haar wavelets, it would be much more complicated (and often impossible) to use integer arithmetic in a generic compression engine, therefore the `double` type is used for most of the wavelet types, which has the highest precision of the available standard types, but represents a bottleneck on systems with low floating point performance. However, the consistent usage of `double` arrays for wavelet coefficients greatly facilitates not only the actual wavelet transformations but also further processing stages like quantization, which can all be written for one data type, no matter what the type of the original data was.

3.3.4 The Wavelet Class Hierarchy

We will now have a closer look at the structure of the wavelet classes in the tilecompression class hierarchy shown in figure 3.4 on page 37. In contrast to the other compression classes, the waveletcomp hierarchy only uses interchannel prediction, because wavelet transformations are equivalent to intrachannel prediction themselves, thereby rendering additional intrachannel prediction superfluous. Predictors will be covered in depth in section 3.5 on page 89.

The waveletcomp class provides functionality shared by all wavelet classes, i.e.

- separating the channels of MDD over structured base types and presenting these individually to child classes which perform the actual encoding/decoding algorithms;
- managing the compression streams (including stream concatenations of arbitrary length) used for the actual compression of the transformed data. The linstream class hierarchy representing compression streams was described in section 3.1;
- managing the banditerator object which divides the bands into equivalence classes and determines the order in which the bands are processed. The banditerator class hierarchy is described in section 3.4.2.1;
- packing the compressed data for all channels and the meta data of all components of the waveletcomp hierarchy required to correctly interpret the compressed channel data into one block of binary data during encoding and extracting meta data and compressed channel data from such a block during decoding.

The child classes of waveletcomp are twaveletcomp for lossless (transform only) wavelet compression and qwaveletcomp for lossy (quantizing) wavelet compression; note that given

[15] Haar wavelets and some wavelet filters which can be approximated in integer arithmetic, as e.g. used in JPEG2000 lossless mode.

sufficiently accurate quantization, qwaveletcomp can also perform lossless compression for most base types[16].

3.3.4.1 Lossless Wavelets

twaveletcomp currently only supports Haar wavelets. The reason for this is that there is a simple way to do this transformation in integer arithmetic by normalizing the filter coefficients differently for analysis and synthesis. The exact Haar coefficients are $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, which are replaced by the coefficients $(\frac{1}{2}, \frac{1}{2})$ during analysis and $(1, 1)$ during synthesis. In other words, the wavelet coefficients are scaled by $\frac{1}{\sqrt{2}}$ compared to those resulting from a transformation using the real Haar filter coefficients, but they're also scaled by $\sqrt{2}$ during synthesis and both effects cancel each other out. This representation also illustrates why Haar wavelets have already been used at times when the wavelet concept was still unknown, because in this form an average coefficient is calculated as the *arithmetic average* (an archetypical low-pass filter) of two consecutive values ($c_i^j = \frac{1}{2}(c_{2i}^{j+1} + c_{2i+1}^{j+1})$) and a detail coefficient is the difference of the first value from this average coefficient ($d_i^j = \frac{1}{2}(c_{2i}^{j+1} - c_{2i+1}^{j+1})$), which is also the negative difference of the second value from the average coefficient. Note that d_j^i is always a signed value, whereas c_i^j has the same sign as the original data. With this encoding, the original values can obviously be restored during synthesis by adding and subtracting the detail coefficient from the average value.

However, there is the problem that the division by 2 can't be done in integer arithmetic on one hand, and that without the division the average and detail coefficients can fall outside the range of the original type. It is still possible to encode average and detail coefficients to allow lossless reconstruction without having to change the base type. The proof for that involves some low-level bit manipulation and is given in appendix A for completeness. The special encoding proposed there for the wavelet coefficients of integer input values allows using the same number of bits for the wavelet coefficients as for the input values, which is both more efficient in terms of runtime memory consumption as well as compression rate; this is not possible for generic wavelets. Finally, the order in which the bands are quantized in twaveletcomp is determined by a banditerator object, which is described in section 3.4.2.1.

3.3.4.2 Quantizing Wavelets

The majority of the wavelet classes in the compression engine are quantizing, which usually implies loss. For most base types, loss is not inevitable provided the quantization is sufficiently accurate, but in this case data normally doesn't compress well, if at all (compare with the results in section 4.3). All quantizing wavelet classes are derived from a common parent class qwaveletcomp, which contains an object of the wavequant class for

[16]Mostly integer types. If the wavelet transformation itself introduced an irreversible error, no lossless compression is possible, no matter how good a quantization is used. More on this will follow in section 3.4.

(de)quantization of wavelet coefficients; more on quantization itself will follow in section 3.4.

In contrast to the twaveletcomp hierarchy, wavelet coefficients in the qwaveletcomp hierarchy are real numbers (IEEE double precision) irrespective of the base type of the original data, i.e. the wavelet coefficients form a D-dimensional array of floating point data. This is necessary because a) generic wavelet filter coefficients are real numbers and therefore the filter response is too, and b) because optimizations like the ones for Haar wavelets in section 3.3.4.1 are unfeasible for generic wavelets. The last stage of decoding wavelet coefficients consists of converting such an array of floating point values back into the base type of the original data. Due to loss, the floating point values are no longer guaranteed to lie within the range of the original data's base type (this mostly applies to integer base types) and must therefore be restricted in range to avoid aliasing errors[17]. This effect has a major impact on the use of predictors and will be addressed in more detail in the corresponding section 3.5.4. The wavequant class is the abstract root class of a (de)quantization class hierarchy for D-dimensional wavelet coefficients. It currently contains two child classes, one for homogeneous band quantization (section 3.4.2) and one for the more efficient (and more complex) Generalized Zerotree coding (section 3.4.3). Other quantization approaches such as a generalized SPIHT (*Set Partitioning in Hierarchical Trees* [46]) or a combination with vector quantization [39] can be added as further child classes to this hierarchy in the future.

There are currently three child classes to qwaveletcomp which implement the following wavelet types (see figure 3.4):

qhaar: lossy Haar wavelets (in contrast to the lossless Haar wavelets in section 3.3.4.1);

daub4: Daubechies 4-tap wavelets [18, 47];

orthowavelet: abstract base class for orthogonal wavelet filters such as Daubechies wavelets [18, 47] or Coiflet wavelets [18]. The class operates on generic orthogonal wavelet filters with even length; concrete filters are initialized by its child class orthofactory which currently contains the filter coefficients for the Daubechies wavelets with 6–20 taps, the Least Asymmetric wavelets with 8–20 taps and the Coiflet wavelets with 6–30 taps.

The reason for separate classes for Haar (2-tap) and Daubechies 4-tap wavelets is that the overhead of the generic approach has more impact on the performance of short filters than that of long ones; both could also be integrated into orthowavelet/orthofactory, but only at a speed penalty. All filters currently used in this engine and their coefficients can be found in appendix B.

There is no support for biorthogonal wavelets often used in image compression (i.e. different filter coefficients for transformation and inverse transformation) at the time of

[17]For instance values that are too large to fit into the base type appear as very large *negative* values and vice versa, e.g. the value 130 appearing as -126 when cast to a signed 8 bit type.

writing this thesis, but these can easily be added as sibling classes to orthowavelet and orthofactory in the future. Thus, the evaluation of the performance of biorthogonal wavelet filters for MDD is part of the future work section.

3.4 Quantization

Quantization is a mapping of a space C (possibly non-countable, e.g. real values) into a subspace $\hat{C}$ (normally countable); another view on quantization is partitioning C into equivalence classes. A simple example for such a quantization would be the function $f_{int} : \mathbb{R} \rightarrow \mathbb{Z}, f_{int}(v) = \lfloor v \rfloor$ which maps real numbers to integer numbers using floor rounding. Another, more complex example is the mapping of real valued numbers to floating point numbers on computers, which consist of a constant number of bits for mantissa and exponent. Quantization is an irreversible process, i.e. it is not generally possible to reconstruct all exact values $c \in C$ from the quantized values $\hat{c} \in \hat{C}$. But quantization allows approximating values using less storage and is therefore an important part of lossy compression algorithms. In the current state of the compression engine, only wavelets require quantization, therefore only the quantization of wavelet coefficients will be discussed in this section in depth. Note that in this context, quantization does not necessarily imply loss, at least not for integer data: the data is converted to its floating point representation, transformed and then quantized during analysis, whereas during synthesis the data is first dequantized, then the inverse transformation is applied to the resulting floating point data, followed by a conversion of the floating point array back into the required integer representation (another quantization step for integer types). Errors in the quantized wavelet coefficients can be insignificant enough to be reduced to zero after the final quantization during synthesis, resulting in lossless reconstruction despite loss in the intermediate representation.

Efficient quantization of wavelet coefficients is a large field of research in itself [49, 50, 12, 10, 46]. In this context, "efficient" usually means minimal rate (average number of bits per cell) with minimal *distortion* (deviation from the original signal). The most commonly used objective distortion measure in lossy compression is the so-called *signal-to-noise* ratio (SNR) defined as

$$\mathrm{SNR} = \frac{\sum_{i}^{N} c_i^2}{\sum_{i}^{N} (\hat{c}_i - c_i)^2} \tag{3.35}$$

which sets the total energy of the original signal (c_i) consisting of N samples in relation to the total energy of the error ($c_i - \hat{c}_i$). The larger the SNR, the better a signal is approximated, leading to $\mathrm{SNR} = \infty$ for the ideal case of lossless reconstruction. Another frequently used distortion measure is the *peak-signal-to-noise* ratio (PSNR) defined as

$$\mathrm{PSNR} = \frac{N \max_i c_i^2}{\sum_i^N (\hat{c}_i - c_i)^2} \tag{3.36}$$

which uses the peak energy of the original signal rather than its average. The study of compression rate vs. distortion is known as *rate-distortion theory*. Possible approaches to the quantization of wavelet coefficients can be classified into two major branches:

1. Quantizing the coefficients of an entire band (section 3.3.3 and figure 3.13) with a uniform number of bits: this typically involves a two-pass approach where statistical information about the bands is gathered in the first pass, which is then used for the actual quantization in the second pass. This approach has the advantage of relatively low complexity, but the compression performance depends critically on the statistic model(s) which are hard to find for the generic case required in this compression engine. An approach like this was used in [12, 13], for example.

2. Successive approximation of all coefficients depending on their magnitude, starting with large coefficients; this corresponds to bitplane coding where the most significant bits are coded first. One effect of this approach is that the reconstructed data is refined successively as more of the compressed data is read, i.e. no blocks of certain length have to be processed in one go, and therefore the decoding process can usually be terminated at arbitrary points without "wasting" symbols in incomplete blocks. Another frequently found effect is *embeddedness*, where any prefix of an encoded symbol stream is itself a valid encoded stream (at lower quality), which also allows encoding data losslessly and merely aborting the decoding process early in case loss is acceptable, thereby eliminating the need for different compressed data for different distortion levels. The most popular representatives of this quantization approach are the *Embedded Zerotree* [49, 50, 10] and *SPIHT* [46].

Both approaches are implemented in the compression engine, however the current statistical model for alternative 1 described in section 3.4.2 is very simple and doesn't perform too well in the generic case, therefore the Generalized Zerotree described in section 3.4.3 is the default.

3.4.1 Wavelet Error Propagation

Wavelet transformations – continuous or discrete – are lossless from a mathematical point of view, i.e. assuming infinite precision of the arithmetic unit performing the transformation, it is possible to reconstruct any signal without loss from the wavelet coefficients. On a real computer, there are naturally always limitations on the precision of floating point numbers, the highest standardized precision being IEEE double precision where a real number is represented as a 64 bit sequence. In order to minimize transformation errors, double precision is used for all internal calculations and the wavelet coefficients themselves.

It must be noted, however, that every arithmetic operation can cause an error with a relative magnitude in the area of the machine precision[18]. While this error can be small enough to still allow lossless reconstruction for some base types, it is usually impossible to reconstruct an MDD whose channels are already in double precision without loss, because the wavelet transformation itself introduced an irreversible error.

Because arithmetic errors can't be avoided in any implementation, it is important to analyse how these errors propagate in the multiresolution wavelet algorithms. This concerns errors introduced by limited machine precision, and in particular quantization errors and their effect on synthesis, where the quantized coefficients are used to reconstruct an approximation of the original signal. Synthesis starts with average and detail coefficients at the coarsest scale level J_0, which are used to calculate the average coefficients at level J_0+1, then these synthesized average coefficients are combined with the (explicitly stored) detail coefficients at level J_0+1 to calculate the average coefficients at level J_0+2 and so forth until at the finest level J an approximation of the original data is achieved (see figure 3.11 on page 52). It is imporant to note that the average coefficients at level j accumulate all errors at levels $J_0, \ldots, j-1$, whereas the detail coefficients – being stored explicitly on each level rather than being calculated like the averages – only contain the quantization error they were stored with (figure 3.14 illustrates this for the 2D case). We will now calculate an upper threshold for the error propagation in multiresolution wavelet synthesis based on these observations.

Let's have a look at the synthesis equations (3.24) and (3.25) as well as the filter coefficient mapping in equation (3.28). Important observations regarding equation (3.28) are that

- the mappings are bijective, i.e. every p_i is mapped to a different h_k and every q_i is mapped to a different h_l;
- there can be no $p_i = q_k$ if both i and k are either even or odd. In combination with the fact that the mappings are bijective, this means that the sets $\{p_i, q_k : i, k \text{ even}\}$ and $\{p_i, q_k : i, k \text{ odd}\}$ are both identical to the set of all filter coefficients $\{h_i\}$.

Both conditions are true for all possible filter offsets. The important consequences regarding the synthesis equations (3.24), (3.25) are that

- every filter coefficient h_i appears exactly once in each of the sums;
- the set of filter coefficients folded with the c_l^j in equation (3.24) is identical to the set of filter coefficients folded with the d_l^j in equation (3.25) and the same goes for the opposite sets.

Now we can examine error propagation, assuming approximate coefficients $\hat{c}_i^j = c_i^j + \delta_{c,i}^j$ and $\hat{d}_i^j = d_i^j + \delta_{d,i}^j$ where $\delta_{c,i}^j$ is the error of the average coefficient c_i^j and $\delta_{d,i}^j$ is the error

[18]The relative magnitude of an error is the ratio of the error to the number the error is modulated on.

of the detail coefficient d_i^j. The total error on the next finer level $\delta_{c,i}^{j+1}$ resulting from a wavelet transformation with these approximate coefficients is the sum of the errors of the two average coefficients calculated, i.e. $\delta_{c,i}^{j+1} = (\hat{c}_{2i}^{j+1} - c_{2i}^{j+1}) + (\hat{c}_{2i+1}^{j+1} - c_{2i+1}^{j+1})$. Using the synthesis equations (3.24), (3.25) we get

$$\begin{aligned}
\delta_{c,i}^{j+1} &= \sum_{l=-\infty}^{\infty} \left(\hat{c}_l^j p_{2i-2l} + \hat{d}_l^j q_{2i-2l}\right) - \sum_{l=-\infty}^{\infty} \left(c_l^j p_{2i-2l} + d_l^j q_{2i-2l}\right) \\
&+ \sum_{l=-\infty}^{\infty} \left(\hat{c}_l^j p_{2i-2l+1} + \hat{d}_l^j q_{2i-2l+1}\right) - \sum_{l=-\infty}^{\infty} \left(c_l^j p_{2i-2l+1} + d_l^j q_{2i-2l+1}\right) \\
&= \sum_{l=-\infty}^{\infty} \left(\left(\delta_{c,l}^j p_{2i-2l} + \delta_{d,l}^j q_{2i-2l}\right) + \left(\delta_{c,l}^j p_{2i-2l+1} + \delta_{d,l}^j q_{2i-2l+1}\right)\right) \\
&= \sum_{l=-\infty}^{\infty} \left(\delta_{c,l}^j (p_{2i-2l} + p_{2i-2l+1}) + \delta_{d,l}^j (q_{2i-2l} + q_{2i-2l+1})\right). \qquad (3.37)
\end{aligned}$$

Due to the previous observations regarding the filter coefficient mappings, we can see that every filter coefficient h_i is folded exactly once with a $\delta_{c,l}^j$ and a $\delta_{d,l}^j$. Now we will calculate an upper limit for $|\delta_{c,i}^{j+1}|$, assuming maximum errors $\delta_{c,\max}^j = \max_l(|\delta_{c,l}^j|)$ and $\delta_{d,\max}^j = \max_l(|\delta_{d,l}^j|)$. Then we get the following upper threshold for the error of each value calculated during a 1D wavelet transformation:

$$\begin{aligned}
|\delta_{c,i}^{j+1}| &= \left| \sum_{l=-\infty}^{\infty} \left(\delta_{c,l}^j (p_{2i-2l} + p_{2i-2l+1}) + \delta_{d,l}^j (q_{2i-2l} + q_{2i-2l+1})\right) \right| \\
&\leq \sum_{l=-\infty}^{\infty} \left| \delta_{c,l}^j (p_{2i-2l} + p_{2i-2l+1}) + \delta_{d,l}^j (q_{2i-2l} + q_{2i-2l+1}) \right| \\
&\leq \sum_{l=-\infty}^{\infty} \left(|\delta_{c,l}^j| (|p_{2i-2l}| + |p_{2i-2l+1}|) + |\delta_{d,l}^j| (|q_{2i-2l}| + |q_{2i-2l+1}|) \right) \\
&\leq \delta_{c,\max}^j \sum_{l=-\infty}^{\infty} (|p_{2i-2l}| + |p_{2i-2l+1}|) + \delta_{d,\max}^j \sum_{l=-\infty}^{\infty} (|q_{2i-2l}| + |q_{2i-2l+1}|) \\
&= (\delta_{c,\max}^j + \delta_{d,\max}^j) \sum_{l=0}^{2N-1} |h_l| \\
&= (\delta_{c,\max}^j + \delta_{d,\max}^j) H, \qquad (3.38)
\end{aligned}$$

where $H = \sum_{l=0}^{2N-1} |h_l|$ is the sum of the absolute values of the filter coefficients, i.e. a filter-dependent constant; since wavelet filter coefficients are normalized to $\sum h_i = \sqrt{2}$ (see equation (3.26)), we can always approximate $H \geq \sqrt{2}$, i.e. the error can grow for all possible wavelet filters. Now let's assume an upper error threshold $\delta_{\max}$ for the quantization error of

all coefficients (coarsest average and all detail bands), i.e. $\delta_{\max} = \max(\delta^{J_0}_{c,\max}, \max_j(\delta^j_{d,\max}))$, using which equation (3.38) can be generalized further to the form

$$|\delta^{j+1}_{c,l}| \leq (\delta^j_{c,\max} + \delta_{\max})H. \tag{3.39}$$

In a D-dimensional wavelet transformation, average and detail bands at the same scale level are transformed along each dimension, i.e. D times (see section 3.3.3). Each pass over the data halves the number of bands by merging all average bands along the current direction with their corresponding detail bands[19], thereby also merging their errors according to equation (3.38). A 2D example of how the error propagates in this case is shown in figure 3.14.

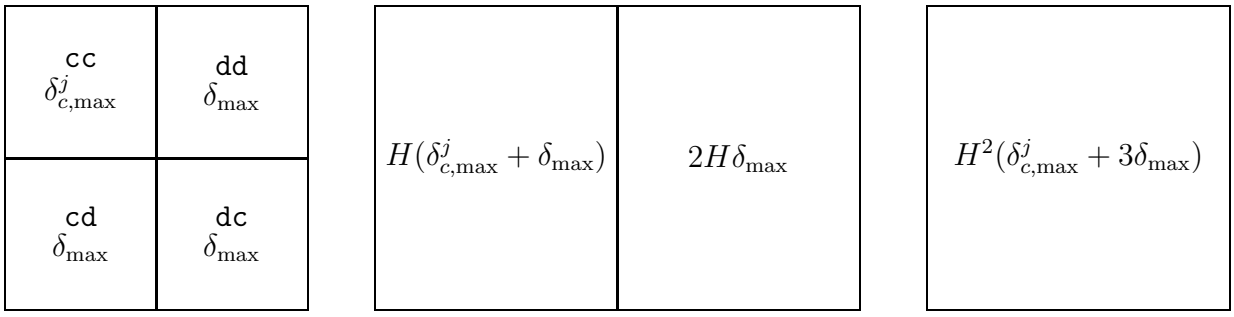


Figure 3.14: Propagation of maximum error in 2D wavelet synthesis
This figure shows how the maximum error propagates in 2D wavelet synthesis. The image to the left shows the bands on the scale level currently processed with their respective labels and maximum errors. The maximum error in the `cc` *band was accumulated from coarser scale levels, whereas it's assumed to be constant for the detail bands which were stored explicitly. The center image shows these bands after a vertical wavelet synthesis pass which merged bands (*`cc`*,*`cd`*) and (*`dc`*,*`dd`*) and their errors according to equation (3.38). The rightmost image shows the result of applying a horizontal wavelet synthesis pass to the bands in the center image, which again merges two bands and their errors in the same manner. Note that the order in which the passes are performed does not affect the final error.*

So starting with one average band with a maximum error $\delta^{j,D}_{c,\max}$ and $2^D - 1$ detail bands with maximum error $\delta_{\max}$, we get one band with a maximum error $H(\delta^{j,D}_{c,\max} + \delta_{\max})$ and $2^{D-1} - 1$ bands with a maximum error $2H\delta_{\max}$ after the first pass by applying equation (3.39) to matching average and detail bands. In the next pass, these errors have to be used instead of the $\delta^j_{c,\max}$ and $\delta^j_{d,\max}$ in equation (3.38), yielding one band with a maximum error $H^2(\delta^{j,D}_{c,\max} + 3\delta_{max})$ and $2^{D-2} - 1$ bands with a maximum error $4H^2\delta_{\max}$. Continuing in this fashion, we get one band with a maximum error $H^i(\delta^{j,D}_{c,\max} + (2^i - 1)\delta_{\max})$ and $2^{D-i} - 1$ bands with a maximum error $2^i H^i \delta_{\max}$ after the ith pass and after the full D passes just one band (the total average band at the next finer level) with a maximum error

[19] e.g. in the 3D case: a pass along the first dimension merges bands `cxy` with bands `dxy`, `x,y` $\in$ { c,d }.

$$\delta_{c,\max}^{j+1,D} \leq H^D \left(\delta_{c,\max}^{j,D} + (2^D - 1)\delta_{\max} \right). \tag{3.40}$$

Starting from the coarsest level and applying this formula over j scale levels, we can see that the maximum error modulated on a coefficient in the finest total average band is bounded by

$$\delta_{c,\max}^{j,D} \leq H^D \delta_{\max} \left(2^D H^{(j-1)D} + (2^D - 1) \sum_{i=0}^{j-2} H^{iD} \right). \tag{3.41}$$

This can be proven by complete induction and equation (3.40); on the coarsest level, the total average band has a maximum error $\delta_{c,\max}^{j,D} \leq \delta_{\max}$ because it was stored rather than calculated. So after the coarsest level is processed ($j = 1$) , the error is bounded by $H^D(\delta_{\max} + (2^D - 1)\delta_{\max}) = \delta_{\max} 2^D H^D$ according to equation (3.40), which is identical to the value of equation (3.41) for $j = 1$. Now for the induction step: by entering equation (3.41) as $\delta_{c,\max}^{j,D}$ into equation (3.40) we get

$$\begin{aligned}
\delta_{c,\max}^{j+1,D} &\leq H^D \left(H^D \delta_{\max} \left(2^D H^{(j-1)D} + (2^D - 1) \sum_{i=0}^{j-2} H^{iD} \right) + (2^D - 1)\delta_{\max} \right) \\
&= H^D \delta_{\max} \left(2^D H^{jD} + (2^D - 1) \sum_{i=1}^{j-1} H^{iD} + (2^D - 1) \right) \\
&= H^D \delta_{\max} \left(2^D H^{jD} + (2^D - 1) \sum_{i=0}^{j-1} H^{iD} \right),
\end{aligned}$$

which is identical to equation (3.41) with $j \to j + 1$, thereby proving equation (3.41). An equivalent form of equation (3.41) without the sum is

$$\delta_{c,\max}^{j,D} \leq H^D \delta_{\max} \left(2^D H^{(j-1)D} + (2^D - 1) \frac{H^{(j-1)D} - 1}{H^D - 1} \right), \tag{3.42}$$

which is better suited to examine the behaviour for high dimensionality or large number of scale levels. As can be clearly seen in this form, the maximum error can grow exponentially in both the number of dimensions as well as the number of scale levels in the worst case, which seriously affects the numerical stability of multiresolution wavelet algorithms in these cases, at least for low rates. Although the actual error propagation is usually far less dramatic, the worst case behaviour must be kept in mind; in particular it is advisable to reduce the number of scale levels for high-dimensional data. Because wavelets work best on smooth data – which is mostly data with a spatial and/or temporal interpretation – the dimensionality of the data being transformed is typically limited to at most 4, so the exponential behaviour of equation (3.42) is not a very severe restriction in real life applications.

3.4.2 Homogeneous Band Quantization

The simplest way to quantize wavelet coefficients is defining equivalence groups on all bands and using a fixed number of bits for all coefficients within bands in the same equivalence group. The simplicity of this approach derives from the fact that every coefficient only has to be visited twice – once while gathering statistical information about the data, such as the value range, and once when actually encoding – and there is no need for complex data structures like in the case of the Generalized Zerotree, which will be introduced in section 3.4.3. This simplicity makes the algorithm fast, but often the quality is considerably inferior to that which can be obtained using zerotree coding at the same rate; furthermore the rate depends critically on the bit allocation strategy across band equivalence groups, which is a non-trivial problem and very dependent on the kind of data being transformed, i.e. strategies for 2D images may differ wildly from strategies for 4D vector fields, but an MDD compression engine can't afford to specialize. These things make homogeneous band quantization inherently problematic in this generic context, despite the lower complexity; it is mainly supported by the compression engine both for completeness and historic reasons, being the first attempt at the quantization of wavelet coefficients. Homogeneous band quantization is based on three modules performing the following tasks:

- defining equivalence groups on the wavelet bands. This is done by the *Band Iterators* covered in section 3.4.2.1;
- gathering statistical information on the coefficient distribution in all band equivalence groups and deciding how many bits to use per coefficient in each equivalence group. This is done by a quantctrl object (see section 3.4.2.3);
- quantizing sequences of coefficients using a given number of bits and a quantization algorithm (e.g. linear, exponential, ...), which is done by a quantizer object (see section 3.4.2.2).

The bandquant class combines these three modules for the actual homogeneous band quantization; it derives from the wavequant class, which defines the interface for the quantization of wavelet coefficients (see figure 3.4 on page 37). An approach like this was also suggested in [12], for instance, albeit with the usual bias on image compression and more emphasis on the statistical model than in this work.

3.4.2.1 Band Iterators

The first step in quantizing the coefficient array is to decide on an order in which the bands should be processed and which bands are grouped together to share a statistical model. There are two extreme approches:

- all bands share the same statistics, i.e. the value distribution and the extreme values are determined by processing all bands in one go. This approach has the advantage that there is a minimum amount of meta data (the data distribution) which has

to be stored in addition to the actual band data for the decoder. The obvious disadvantage is loss of resolution in local deviations from the statistical averages; this is very common in wavelet coefficients, because most detail coefficients are very small compared to the average coefficients and will therefore often be quantized to zero if they share the same statistics as the average band;

- every band has individual statistics. In this case, each band can be encoded with its ideal statistical model, but the amount of meta data that has to be stored is considerably larger than in the first case. Because the meta data needed per band is usually very small compared to the actual band data (typically minimum and maximum value, i.e. two floating point numbers), this is not a critical problem, however.

In order to use a good statistical model for each band while keeping the amount of additional meta data low, a compromise between these two extremes has to be found. Note that because the size of the meta data is relatively small, the disadvantages of the first extreme easily outweigh those of the second, i.e. using more statistical units than strictly necessary has less negative influence on the quantizer's efficiency than using too few.

Band iterators perform the task of grouping bands into logical units (band equivalence groups) sharing a statistical model. Bands are iterated over in such a way that those bands belonging to the same equivalence group are visited consecutively, as required to efficiently encode these groups. This functionality is provided by a separate class hierarchy with an abstract base class banditerator defining the common interface. There are currently three different band iterator classes available:

isolevel: all bands on the same scale level are grouped. The number of band equivalence groups scales with the number of resolution levels;

leveldet: all bands on the same scale level and with the same degree of detail are grouped. The number of band equivalence groups scales with the product of the number of dimensions and the number of scale levels;

isodetail: all bands (across scale levels) with the same degree of detail are grouped. The number of band equivalence groups scales with the number of dimensions.

Band equivalence groups are numbered, starting from 0 at the total average band, and iterated over in increasing order. For **isolevel** and **leveldet**, the numbers increase towards finer levels, for **leveldet** and **isodetail** the numbers (also) increase with the degree of detail, so for instance `cd` comes before `dd`; this order is relevant when manually fine-tuning the bit allocation in the band statistics module (see the `relqbits` parameter in appendix C). In most cases **leveldet** performs best and will be used for the tests in section 4.3.2 and is therefore the default band iterator (it also has the finest equivalence group granularity).

3.4.2.2 Quantizers

The actual quantization of the real-valued coefficients is performed by a separate quantizer class hierarchy. Given minimum and maximum values, all values are translated and scaled such that the midrange value is mapped to 0 and the minimum and maximum values are also the extreme values of the quantized representation, assuming a linear quantizer[20]; the offset and scaling factor are stored as meta data for the dequantizer. Quantizers don't gather any statistical information about the data themselves, but receive the relevant information from a separate quantctrl object (see section 3.4.2.3). The quantized data is not aligned to bits but to the smallest standard data type with at least the same size, e.g. bytes for 1–8 bits or 32 bit integers for 17–32 bits. It is eventually written to a compression stream which (implicitly) removes unused bits in case there is no standard base type with exactly the size required, e.g. when quantizing to 4 bits and the remaining 4 bits in the byte (= standard base type used in this case) are unused. Because most compression algorithms examine data on byte- rather than bit-level, they would actually perform worse if the quantized data was not at least aligned to bytes.

The quantizer hierarchy roots in an abstract base class defining the interface. When quantization is started, midrange offset o_{mr} and scale factor[21] s_q that should be applied to each value v in the form $v' = (v - o_{\mathrm{mr}})s_q$ before the actual quantization, as well as a compression stream which should receive the quantized data are passed to the quantizer; following that the data can be streamed into the quantizer. Note that use of the quantizer classes is not limited to wavelet coefficients, but extends to any kind of real valued data and can therefore be used even outside the tile compression context. Currently available quantizers are:

linear: linear quantizers are the most common ones and achieve the best results when the data is distributed uniformly across the value range. A linear quantizer using b bits per value merely transforms each value (using the midrange offset and scale factor) so it fits into the range of a signed integer type with b bits, i.e. from -2^{b-1} to $2^{b-1}-1$, and then rounds the result to the nearest integer to obtain the quantized value, i.e. $v_q = \lfloor v' + \frac{1}{2} \rfloor$. For instance when quantizing values ranging from -1000 to 2000 to 5 bits, then $o_{\mathrm{mr}} = 500$ and $s_q = 0.01$. Applying these to the value 300 would result in the quantized value $v_q = \lfloor (300 - 500)0.01 + 0.5 \rfloor = \lfloor -1.5 \rfloor = -2$. The quantizer step size [22] $\frac{1}{s_q}$ is constant across the entire value range. In the example, the quantizer step size is 100, which leads to a maximum error per coefficient of 50.

exponential: this exponential quantizer has increased resolution around 0, which decreases exponentially for large values. This is achieved by dividing the value range of the *quantized* data (-2^{b-1} to $2^{b-1}-1$ for b bits) into buckets containing 2^r uniformly

[20] Other types of quantizers can use midrange offset and scale factor to reconstruct the minimum and maximum values of the data if necessary.

[21] Note that this scale factor has nothing to do with the multiresolution scale levels of the wavelet transformation.

[22] The maximum difference between two values that are quantized to the same value.

distributed values each. This results in 2^{b-r} buckets, half of which cover the positive and half the negative values of the quantized data range. The range of values in the original data covered by a bucket is twice the range of the bucket's nearest neighbour towards the zero point of the quantized data, e.g. if the first positive bucket covers the range $[0, 3[$ then the next positive bucket covers the range $[3, 9[$, the next one $[9, 21[$ etc. With the smallest value range v_0 we get the largest positive value that can be represented that way as $v_{\max} = v_0 \sum_{i=0}^{b-r-1} 2^i = v_0(2^{b-r} - 1) = -v_{\min}$, which can be used to calculate v_0 such that $[v_{\min}, v_{\max}]$ covers the entire range of the original data centered around 0. As a result, every bucket can be assigned a unique value range in the original data and a value is quantized by first determining the number of the bucket it falls into ($b - r$ bits of the quantized value) and then the closest of the 2^r values within the bucket (the remaining r bits of the quantized value). Using the same example as for the linear quantizer and 8 buckets with 4 entries each ($r = 2$), we get $v_0 = \frac{1500}{2^{5-2}-1} = \frac{1500}{7} \approx 214.286$. The value 300 is offset from the midrange point at 500 by -200 and therefore falls in the first negative bucket and within that bucket it's closest to the last of the four values (which covers the interval $]-\frac{6000}{28}, -\frac{4500}{28}] \approx]-214.286, -160.71]$ in the original data). How exactly bucket numbers and nearest bucket value are encoded in b bits is of no further concern in this example. The minimum quantizer step size is $\frac{1500}{28} \approx 53.57$ in the bucket closest to zero, whereas the maximum quantizer step size is 8 times that (≈ 428.571), so the maximum error per coefficient ranges from ≈ 26.79 to ≈ 214.286 in this example.

Which quantizer should be chosen depends entirely on the data distribution. If there is a concentration in the middle of the value range and relatively few larger values, the exponential quantizer will result in lower average quantization error, because the quantization error is smaller in the buckets close to 0. In case the data is distributed uniformly, the opposite applies and the exponential quantizer's large errors for big values outweigh the small errors in the middle and result in a bigger average quantization error. Note that ideally there should be quantizers for every major distribution class, so the optimum one can be chosen by a statistical data analyser. More on this issue will follow in the next section.

3.4.2.3 Quantization Statistics

Efficient quantization requires knowledge about properties of the data being quantized, foremost of all its value range. For instance if the value range assumed by the quantizer was twice as large as the actual range, one bit of precision in the quantized data would be wasted; on the other hand, if the assumed value range was smaller than the actual one, huge quantization errors would result for those values outside the assumed range. Generally speaking, the more information about the data is known and used in quantization, the smaller the distortion gets without having to increase the number of bits. For instance if the data is concentrated in certain areas, a quantizer with higher resolution in these

areas usually achieves lower average distortion than a standard linear quantizer. It follows from these considerations that the data has to be analysed first before it can be quantized, making the quantization a two-pass operation. The dequantizer doesn't have a statistical pass, because initially it only sees the quantized data and not the original data, therefore data properties used by the quantizer (*quantizer meta data*) must be stored along with the quantized data to allow the dequantizer to correctly interpret the data.

The task of gathering statistics and then encoding the data based on these statistics is performed by an object of the quantctrl class. Each band equivalence group is encoded in two passes:

1. **statistical pass**: a new statistical unit is started by issuing a call to the `band_start_statistics()` method. Following that, all values in the same band equivalence group are fed into the quantctrl object using the `band_get_statistics()` method, whereby properties meaningful for the quantization are extracted. After all data in the equivalence group has been processed that way, the statistical pass is ended with a call to the `band_end_statistics()` method, after which another statistical pass or an encoding pass can follow.

2. **quantization pass**: the actual quantization of the band equivalence group starts with a call to the `band_start_encode()` method, which initializes a quantizer based on the statistical data available. Data in the currently processed band equivalence group is then fed into the quantctrl object using the `band_put_values()` method which passes it on to the quantizer used. After all data has been processed that way, the quantization pass is terminated with a call to the `band_end_encode()` method.

Statistical data used for all band equivalence groups is stored internally, so after the coefficients of all channels have been processed, the quantctrl object contains the quantization meta data for the coefficient arrays of all channels; this meta data is stored within the compressed tile along with the meta data of all other compression modules involved in the current compression operation. For decoding, first the meta data is read from the compressed tile, then all band equivalence groups are dequantized using the `band_start_decode()`, `band_get_values()` and `band_end_decode()` methods. Note that the quantctrl class is not restricted to the wavelet subband context but can be used on arbitrary real valued data, just like the quantizer hierarchy.

Ideally the quantctrl class will first gather substantial statistical data from all band equivalence groups, use a quality measure like a desired signal-to-noise ratio to determine how many bits to use for the quantization of each group and which quantizer is best suited for the data distribution within the group. However, the currently implemented system is less powerful in that it neither automatically distributes the bits nor chooses a quantizer, both of which are left for the user to decide. Both extensions should be implemented as part of the future work because they'd simplify using homogeneous band quantization and make it more efficient; since there is also Generalized Zerotree coding available (see section 3.4.3), which is both easy to use and typically achieves better compression rates

than more sophisticated variants of homogeneous band quantization [12], work in this area was postponed for the time being. Note that in order to exploit special data distributions, several new types of quantizers would have to be added as well.

The currently implemented system allows specifying the base number of bits to use per quantized value (1–31), a distribution function $d : \mathbb{N} \rightarrow \mathbb{R}^+$ which determines for each band equivalence group how many bits relative to that base number should be used within that group, and a cutoff group number g_{co}, which is the number of the first band equivalence group which should be ignored (quantized with 0 bits; this allows discarding detail information finer than a given resolution level, i.e. this detail is ignored entirely by both encoder and decoder). The distribution functions currently available (for band groups g with $0 \leq g < g_{\text{co}}$) are

const: $d(g) = 1.0$: use the same number of bits for all band equivalence groups;

linear: $d(g) = 1.0 - \frac{g}{g_{\text{co}}}$: linearily decrease the number of bits with the band equivalence group number;

exponential: $d(g) = e^{-\ln(8)\frac{g}{g_{\text{co}}}}$: exponentially decrease the number of bits with the band equivalence group number. The function is normalized to return $\frac{1}{8}$ for $g = g_{co}$;

gauss: $d(g) = e^{-\ln(8)\left(\frac{g}{g_{\text{co}}}\right)^2}$: exponentially decrease the number of bits with the square of the band equivalence group number;

custom: $d(g) =$ user defined: the user specifies the bit allocation for each band equivalence group manually as a comma-separated string (see the parameter system in section 3.6 and the `relqbits` parameter in appendix C).

The kind of distribution function used is stored as an enumerator in the quantization meta data. For the **custom** distribution function the bit allocation for each band equivalence group has to be stored in the meta data as well.

3.4.3 The Generalized Zerotree

In order to improve the efficiency of encoding wavelet coefficients, they are typically encoded depending on their magnitude rather than their band, because large coefficients contain most of the data's characteristic information, irrespective of their band. This approach is related to bitplane coding often used in computer graphics, where arrays with a depth of b bits are represented as b arrays with a depth of 1 bit; if these arrays are encoded in decreasing order of bit significance, the array values are encoded in decreasing order of magnitude and the data approximation is successively refined with each bitplane during decoding. This approach also treats all coefficients identically, no matter what band they belong to, so it doesn't require any heuristics about the number of bits to allocate to each band. Due to this, it can adapt to data with different characteristics far easier than techniques using a fixed number of bits per band can. On the other hand, it must be

noted that successive refinement is more expensive, because the values have to be touched several times as they're encoded bit-by-bit.

More sophisticated bitplane coding techniques additionally exploit the multiresolution hierarchy of wavelet bands by correlating coefficients corresponding to the same position in the original data, but in neighbouring scale levels. The argument in favour of this approach is that sharp edges in the original data usually result in large wavelet coefficients on all scale levels, and conversely smooth areas in the original data result in small wavelet coefficients on all levels. One of the most popular representatives of this variety of bitplane coding algorithms is Shapiro's *Embedded Zerotree* (EZT) [49], another one is SPIHT [46]; we will concentrate on EZT here, which was generalized and implemented in the $\mathcal{RasDaMan}$ compression engine.

3.4.3.1 The 2D Zerotree Structure

We will first introduce the tree structure of the EZT for the original 2D case and then extend this structure for an arbitrary number of dimensions in section 3.4.3.4. Regarding the correlations of coefficients, which are modelled as parent-child nodes in the tree structure, we have to recall that the wavelet coefficients within *each* band represent (properties of) the *entire* original data at specific scale levels, e.g. the coefficients in the corner points of each band correspond to data properties in the corner points of the original data. Combined with the fact that the size of the bands is halved in each dimension every time the scale level is coarsened, this means that one wavelet coefficient at level j covers the same area in the original data as two coefficients in each dimension of the next finer level $j+1$ (see figure 3.15). Consequently, there is a mapping function $p_2 : (\mathbb{Z}^2, \mathbb{Z}) \rightarrow (\mathbb{Z}, \mathbb{Z})^2$ which maps a 2D coordinate (x, y) relative to the band origin and a scale level j to a 2D interval in the spatial domain of the original data:

$$p_2((x, y), j) = [2^{J-j}x, 2^{J-j}(x+1) - 1] \times [2^{J-j}y, 2^{J-j}(y+1) - 1], \tag{3.43}$$

using the same variable binding as in section 3.3.2, where J was the finest scale level and $j < J$ were the coarser scale levels, up to the coarsest level J_0. The EZT uses this fact to correlate nodes covering the same area in the original data in the following way (see figure 3.15 for a graphical representation and figure 3.13 on page 55 for the band labels used):

- a "zerotree" structure consists of as many subtrees as there are average coefficients on the coarsest level (`cc1` in the example in figure 3.15); each subtree root contains one average coefficient.

- the average coefficient on the coarsest level at position (x, y) relative to its band's origin covers the same area in the original data as the detail coefficients at the coarsest level (`cd1`, `dc1`, `dd1` in the example) at position (x, y) relative to their bands' origins. Therefore each root node has up to three child nodes in the detail bands at the same level; boundary effects could reduce the number.

- each (non-root) node at relative position (x, y) on level j covers the same area in the original data as the nodes described by equation (3.43) in all bands on level $j + 1$. However, coefficients are usually correlated stronger if they are for the same direction, in other words the coefficients of band `cd1` are correlated stronger with those of band `cd2` than those of band `dc2` or `dd2`. Furthermore, modelling the correlations for all directions would result in a DAG (*Directed Acyclic Graph*) rather than a tree, thereby increasing the number of correlations dramatically. Therefore, only bands of the same direction are correlated in the EZT structure, which results in up to four child nodes at the next finer scale level for each parent node; once again, boundary effects can reduce that number.

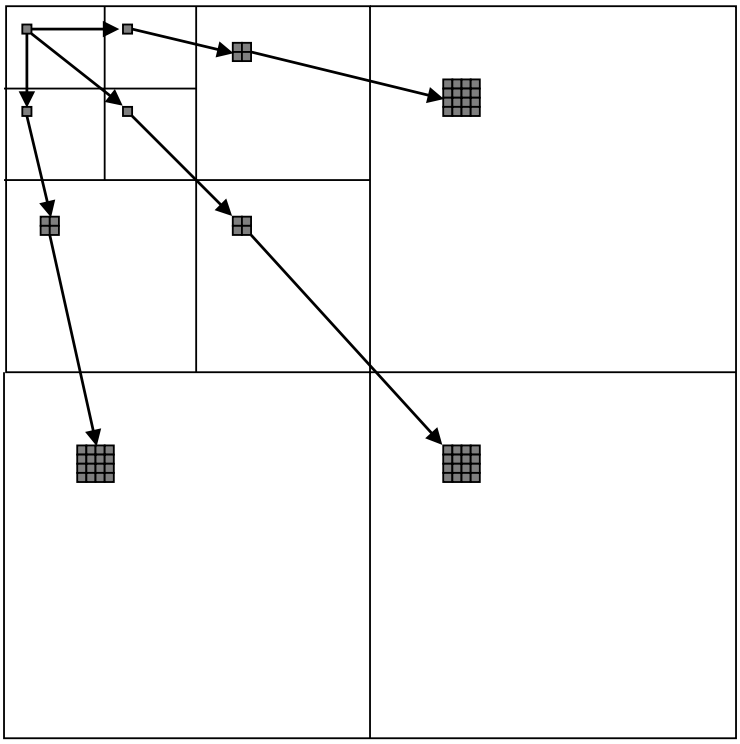


Figure 3.15: Parent-child node relationships in a 2D Zerotree
This figure shows the zerotree structure for the 2D case; the matching band decomposition can be found in figure 3.13 on page 55. Using the same labels for the band decomposition, all root nodes are located in band `cc1` *and can have child nodes in the detail bands at the same scale level (*`cd1`*,* `dc1` *and* `dd1`*). Each of these can in turn have up to four child nodes at the next finer scale level in the band with the same direction.*

Note that in this tree structure, there are exactly as many nodes N as there are coefficients in the array the tree is based on, because every node, including inner nodes, is associated with a value in the original array data (compare with figure 3.15). This is in contrast to e.g. a Quadtree, where only leaf nodes have this property, but not the inner

nodes, and the ratio of nodes in the tree to the number of coefficients in the array is $\frac{4}{3}$ (for the 2D case).

Also note that the tree structure depends on the band partition only, not the coefficients themselves. The band partition in turn depends on the spatial extent of the original data and the number of scale levels in the band partition. That means that while the spatial extent and the number of hierarchical levels remain constant, the tree structure remains constant as well and can be reused. For instance in a colour image, the tree structure only has to be built once and can then be used to encode the coefficients of all three colours. If many tiles in an MDD have the same spatial extents, there are also performance gains possible by caching the tree structure over different tiles – at a memory premium, of course.

3.4.3.2 Encoding and Tree Alphabet

Once the EZT structure has been built, it can be used to iteratively encode the wavelet coefficients within the bands it is based on. The original EZT [49] used two passes per iteration and a six-symbol alphabet for this purpose. The data is processed in alternating *dominant* and *subordinate* passes until the coefficients have been encoded with sufficient precision; this can be the case when an SNR threshold was met (the most common policy), or the absolute values of all coefficients are below a residual threshold, or similar criteria. The coding starts with an initial threshold value T with $\frac{M}{2} \leq T < M$, where $M = \max_i(|v_i|)$ is the maximum absolute value of all coefficients in the tree (any threshold value T within this interval can be used as initial coding threshold, typically $\frac{M}{2}$ is used); T is halved after each iteration, as the data is successively refined. The two passes work as follows:

dominant pass: in this pass, the nodes on the coarsest level are iterated over for *significance* coding (a node is significant if its absolute value is larger or equal to T). If a node hasn't been encoded as significant in a previous dominant pass yet, its significance is checked against the current threshold value T and one of the following four symbols is emitted depending on the result:

zpositive: the node is significant and has a positive value;

znegative: the node is significant and has a negative value;

zisolated: the node is insignificant, but at least one of its children is significant;

ztreeroot: the node is insignificant and so are all of its children.

In case the symbol is **zpositive** or **znegative**, the node will be ignored in all subsequent dominant passes; its child nodes will always be iterated over, however. In case the symbol is **ztreeroot**, no children of this node are processed any more in this dominant pass; if the symbol differs or the node has already been encoded as significant in a previous dominant pass, all its child nodes are processed recursively in the same manner. This allows exploiting correlations between wavelet coefficients over neighbouring scale levels and typically greatly reduces the number of symbols

needed to represent the tree. In case the node is significant (and because of the initial threshold value and the encoding algorithm's design also smaller than $2T$ at all times, i.e. $|v_i| \in [T, 2T[$), its reconstruction value will be the midpoint value $\pm\frac{3}{2}T$ (depending on its sign), which is the value the subordinate pass (and of course the decoder) will operate on in subsequent passes.

subordinate pass: in this pass, all nodes that have already been encoded as significant in a previous dominant pass are refined with one additional bit of precision, which determines whether the currently used reconstruction value is larger or smaller than the actual value by subtracting or adding $\frac{T}{2}$ respectively. Thus, the subordinate pass uses a two-symbol alphabet **zplus**, **zminus** and ideally a different compression stream.

Newer publications on the EZT [10] extended the alphabet used in the dominant pass with additional symbols for the cases where the node is significant (positive or negative) but none of its children are; this extension can easily be added to the existing zerotree implementation, but currently the original alphabet is used (more on encoding variants follows in section 3.4.3.7).

Note that the subordinate pass doesn't work with exact matches, so if a coefficient was already encoded exactly before the subordinate pass, its value will change to a less precise value in the subordinate pass (whether the value will be incremented or decremented in this case is a heuristics on the part of the encoder and only affects the sign, not the amplitude of the error). Nor is it possible to stop further processing of these exact coefficients with the tree alphabet listed above, because this would require an extra symbol to inform the decoder that an exact value has been reached, as it has no other means to differentiate between a fully and a partially encoded value. This shortcoming of the subordinate pass has hardly any impact on the encoding of real-life wavelet coefficients for two reasons, however:

- the coefficients are real numbers which almost never allow an exact match in the first place;
- in the rare event that an exactly encoded coefficient was modified in a subordinate pass, the coefficient will converge towards its exact value again in subsequent subordinate passes with arbitrary precision because $2^i x = x \cdot \lim_{k \to -\infty} \sum_{j=k}^{i-1} 2^j$ where $2^i x$ is the value the exact match was changed by (this value is halved in each subsequent subordinate pass, i.e. $2^{i-1}x, 2^{i-2}x, \ldots$, hence the sum).

The compression stream most frequently used to actually compress the symbols emitted during encoding is an adaptive arithmetic coder as described in section 3.1.1. The small size of the alphabet greatly improves the performance of the adaptive layer and thereby results in a substantial speedup compared to *ZLib* which performs very badly on the EZT alphabet (see results section 4.3.3).

The successive refinement of the encoded data also makes the zerotree *embedded*, i.e. the symbol stream for data with a given quality also contains (= embeds) the symbol streams for data at all lower quality levels and is a prefix for all versions of the data at higher quality. Thus, higher quality approximations are done by simply appending symbols to the stream. If the data is encoded without loss, it is therefore possible to serve arbitrary quality requests with *one* stream of zerotree symbols by truncating the stream earlier for lower or later for higher quality. It also allows meeting exact SNR rates much easier than with the homogeneous band quantization covered in section 3.4.2. Finally, the amount of meta data that has to be stored along with the encoded data to allow correctly decoding the zerotree symbol stream is very small, consisting solely of the initial threshold value.

3.4.3.3 Encoding Example

This section illustrates how EZT coding works by encoding a small 2D example. The coefficient array contains 4×4 coefficients over two scale levels in seven bands, where each band is represented graphically by a frame. Furthermore, each node in this example has an entire band as children, which is not normally the case, but keeps the example small and easy to handle. The coefficient values were chosen at random. The coarsest average band is to the top left and contains one coefficient, whose three children are in the detail bands of the same level and each contain one coefficient as well. Each of these detail coefficients has four children in the next finer band of the same direction. The order in which child nodes within a band are processed was not specified by the original EZT publication; I will use Z order, i.e. top-left → top-right → bottom-left → bottom-right as shown in figure 3.16.

For each iteration, threshold and reconstruction value will be shown, followed by a graphical representation of the band structure to the left and the symbols output in dominant and subordinate pass to the right. For more clarity, values encoded in the dominant pass will be appended to the output symbol in parantheses; encoded symbols will be added to the *subordinate list* (SL), which has the format *actual value*:*current reconstruction value*, which is initially empty and will be shown after both passes. Values encoded as significant in a dominant pass will be replaced by an X in the band diagram of the next iteration and ignored in subsequent dominant passes. In order to save space, the symbol names will be abbreviated throughout this example. In leaf nodes, the symbols **zroot** and **zisol** are equivalent and **zisol** is used in preference. In case of an exact match at the beginning of the subordinate pass, the value is incremented via the **zplus** symbol; in this case **zplus** is shown (underlined to stress this special case; the symbol actually output is the regular **zplus**).

The symbols from the zerotree alphabet listed for dominant and subordinate pass are for the current iteration only. After each iteration, these symbols are written to the compression stream and the next iteration starts with empty symbol lists. Only the subordinate list accumulates information from all previous iterations, but is discarded once encoding stops.

Before encoding an entire coefficient tree, I will first show which symbols are output for

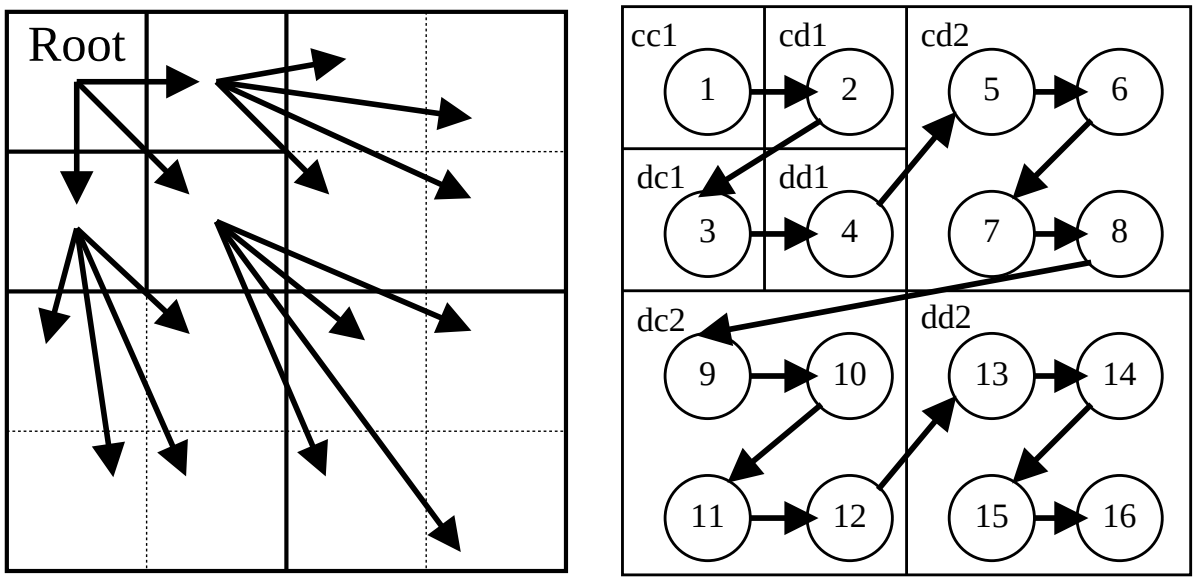


Figure 3.16: Zerotree example relationships and coding order
These figures show the band decomposition of the 4×4 *wavelet coefficients used in the zerotree coding example. To the left, the parent-child relationships are explained using arrows from each parent node to all its children. The encoding order can be seen to the right, where for each tree node its coding position is represented by an encircled number and the arrows describe the coding path (Z-order). Not all nodes are visited during a dominant pass (e.g. no nodes in* `cd2` *will be visited if the node in* `cd1` *was a zerotree root in this and all previous dominant passes), but the numbers of the nodes processed in a dominant pass are monotonously increasing.*

a leaf node with the coefficient value 7 during each iteration. The table has to be read left-to-right and top-to-bottom; the subordinate list is shown twice, once after the dominant pass and once after the subordinate pass. This example also shows how the reconstruction value converges towards the actual value after an exact match was destroyed with **zplus**.

T	dominant	SL	subordinate	SL
16	**zisol**			
8	**zisol**			
4	**zpos**	7:6	**zplus**	7:8
2		7:8	**zminus**	7:7
1		7:7	**zplus**	7:7.5
0.5		7:7.5	**zminus**	7:7.25
0.25		7:7.25	**zminus**	7:7.125

Now for the encoding of a real zerotree, where additionally the order in which nodes are encoded and relationships between the values of parent and child nodes have to be

taken into account.

Iteration 1: Threshold 16, reconstruction value 24

17	8	3	1
7	11	2	9
6	2	0	4
1	5	3	2

Dominant:
zpos(17), **zroot**(8), **zroot**(7), **zroot**(11);
SL: 17:24

Subordinate: ±8
zminus;
SL: 17:16

Iteration 2: Threshold 8, reconstruction value 12

X	8	3	1
7	11	2	9
6	2	0	4
1	5	3	2

Dominant:
zpos(8), **zroot**(7), **zpos**(11), **zisol**(3), **zisol**(1), **zisol**(2), **zpos**(9), **zisol**(0), **zisol**(4), **zisol**(3), **zisol**(2);
SL: 17:16, 8:12, 11:12, 9:12

Subordinate: ±4
zplus, **zminus**, **zminus**, **zminus**;
SL: 17:20, 8:8, 11:8, 9:8

Iteration 3: Threshold 4, reconstruction value 6

X	X	3	1
7	X	2	X
6	2	0	4
1	5	3	2

Dominant:
zpos(7), **zisol**(3), **zisol**(1), **zisol**(2), **zpos**(6), **zisol**(2), **zisol**(1), **zpos**(5), **zisol**(0), **zpos**(4), **zisol**(3), **zisol**(2);
SL: 17:20, 8:8, 11:8, 9:8, 7:6, 6:6, 5:6, 4:6

Subordinate: ±2
zminus, **zplus**, **zplus**, **zplus**, **zplus**, **zplus**, **zminus**, **zminus**;
SL: 17:18, 8:10, 11:10, 9:10, 7:8, 6:8, 5:4, 4:4

Iteration 4: Threshold 2, reconstruction value 3

X	X	3	1
X	X	2	X
X	2	0	X
1	X	3	2

...

Encoding continues in a similar manner until sufficient precision has been reached, but the example ends here.

3.4.3.4 The Generalized Zerotree Structure

The original EZT paper [49] was for the 2D case only; several years later there was also a publication on a 3D version used in wavelet-based video compression [10], but no generic solutions are known so far. Because the EZT structure depends on the band partition performed by the multiresolution wavelet analysis, which in turn depends on the dimensionality, the EZT structure also changes with the dimensionality; for instance the number of child nodes changes with the number of dimensions.

The number of bands created on each scale level by the D-dimensional wavelet transformation described in section 3.3.3 is 2^D, because each pass splits all bands in the currently processed scale level in two (compare with figures 3.6, 3.7 starting on page 43), so in the 2D case an average band is partitioned into four bands on each level, in 3D into eight bands etc. The number of child nodes on the coarsest level of the EZT equals the number of bands on the coarsest level minus 1 (the band containing the parent node itself, `cc1` in figure 3.13), i.e. $2^D - 1$, which is also the number of directions. The mapping function in equation (3.43) becomes a function $p_D : (\mathbb{Z}^D, \mathbb{Z}) \to (\mathbb{Z}, \mathbb{Z})^D$ in the general case:

$$p_D((x_1, \ldots, x_D), j) = [2^{J-j}x_1, 2^{J-j}(x_1 + 1) - 1] \times \cdots \times [2^{J-j}x_D, 2^{J-j}(x_D + 1) - 1] \quad (3.44)$$

and consequently a coefficient on level j covers the same area in the original data as 2^D coefficients on the next finer level $j + 1$, leading to 2^D child nodes for each *inner node*[23] that isn't also a root node. For a node at the relative position $(x_1, \ldots, x_D)$ in a band in (the non-top) level j, the child nodes on level $j + 1$ are therefore located at the relative positions

$$\text{children}(x_1, \ldots, x_D) = \{(2x_1 + z_1, \ldots, 2x_D + z_D) : z_i \in \{0, 1\}\} \quad (3.45)$$

in the next finer band in the same direction for all possible combinations of z_i resulting in legal positions (boundary effects may make some positions illegal). As a consequence, the node size and with it the size of the tree structure grow exponentially with the number of dimensions. This is a very undesirable property, but it can be avoided with careful implementation [21] by exploiting the fact that no leaf node has any children, as shown in section 3.4.3.5.

3.4.3.5 Implementational Issues

There are several problems implementing this general tree structure in an efficient way, like for instance the variable (worse yet: exponential) number of child nodes. In this section, I will propose a tree representation which has the following properties, some of which are also highly attractive for a non-generic version:

[23] An inner node is a node which has at least one child node.

- totally generic with respect to the number of dimensions;
- no hidden memory overhead per node by using a *node pool* for storing all tree nodes. If each node was allocated from the system heap individually, such hidden overhead – typically 4–8 bytes per heap block – couldn't be avoided;
- the tree structure contains no pointers, only integers, and could therefore also be put into ROM for special applications like a video compression board with fixed resolution, where RAM is a premium;
- the values at the tree nodes and the array data the tree is based on are one and the same, i.e. the tree references the array values. This avoids copy cycles from one representation into the other before encoding or after decoding;
- the memory consumption of the tree structure relative to the number of nodes *decreases* with the number of dimensions.

The memory overhead per node can be addressed by claiming a block of memory with a size large enough to hold all tree nodes and allocate nodes from there; this is the node pool. Because nodes are only allocated and never freed when building the tree structure, this is a trivial matter of incrementing a counter each time a node is allocated and doesn't require any complicated heap management functionality.

The memory structure of each node is still undefined at this point, so let's analyse the requirements: each tree node contains 2^D references to child nodes (which may be `NULL`) and one node value[24]. The naive implementation for an entity like this would be a structure containing 2^D pointers and a value, but there is a better way using self-describing integers for everything. First up, each node can be assigned a unique node number which doesn't necessarily have anything to do with the node's position. The `NULL` value can be expressed by a special integer `noChild`, such as the largest integer value possible. As for the node values, these can also be expressed as references, i.e. offsets into the coefficient array the tree is based on. If we store value references transposed by the maximum number of nodes in the node pool N_I, the integers i become self-describing references:

$$\begin{array}{rl} i = \texttt{noChild} & \texttt{NULL} \text{ reference; otherwise:} \\ 0 \leq i < N_I & \text{reference to a node} \\ N_I \leq i & \text{reference to a value} \end{array} \tag{3.46}$$

There are alternative approaches like using a specific bit to differentiate between the reference types, of course; but this encoding has the advantage that all references lie in a continuous numeric range without undefined values between minimum and maximum reference number.

It must be mentioned that this approach has the disadvantage of higher complexity, because in order to interpret a reference i encoded like this it first has to be compared

[24]The root nodes actually have one child reference less, however the current implementation doesn't use different node structures for root nodes, but merely marks one child reference as invalid there.

against N_I and then used to either calculate the address of the node by adding i times the node size to the start address of the node pool ($i < N_I$), or the address of the value by using $i - N_I$ as index into the coefficient array ($i \geq N_I$). But this method to encode nodes also has two major advantages:

- all attributes of a node have the same type and can therefore be represented by a subarray of length $2^D + 1$, which is a very convenient way to model this dynamic structure. The encoding chosen in this engine is to use the first entry in a node for the value reference and the remaining 2^D entries for the child node references, which allows some optimizations, but the decoder could also be made to work correctly without knowing the position of the value reference, since all references are self-describing;

- by encoding the values as references to values, tree structure and array data become equivalent and changes to one representation are immediately visible in the other. It also means that no copy cycle is needed to convert between representations, which compensates some of the higher complexity required for parsing a reference value.

Furthermore, the complexity of the compression stream (section 3.1.1) used to actually compress the symbols describing the tree must be taken into account, which is in most practical cases considerably higher than that of parsing the tree nodes (see the timings for decompressing the *lena* image in figure 4.2 on page 116).

Now let's examine how many entries N_I the node pool must be able to hold. Because only the inner nodes can have children, there should only be memory allocated for these nodes in the node pool, which raises the question of how to model the references to the leaf nodes on the penultimate level of the tree: since no memory is allocated for the leaf nodes, references to leaf nodes must have a different type than references to inner nodes. Because the leaf nodes contain no references to children but only to a value, we can simply replace the references to child nodes on the penultimate level with references to the child nodes' values; this is possible because all references are self-describing integers. The next question is how many inner nodes there are. As we already established, the D-dimensional wavelet transformation partitions the data it is applied to into 2^D bands of (approximately) the same size[25]. In other words, each band has about $\frac{1}{2^D}$ times the size of the data being transformed. For the following considerations regarding the number of inner nodes, we will assume that the wavelet transformation produced 2^D new bands with exactly the same size. The inner nodes are all within the c^D band on the finest level, i.e. there are $N_I = \frac{N}{2^D}$ inner nodes, the number of inner nodes *decreases exponentially* with the number of dimensions. At the same time the node size $2^D + 1$ increases exponentially; both effects cancel each other out, leading to total memory requirements (in integers) of

$$\text{mem}_{\text{node}} = N_I(2^D - 1) = \frac{2^D + 1}{2^D} N \tag{3.47}$$

[25]All 2^D new bands have exactly the same size if the spatial extent of the currently processed band is divisible by 2 in all dimensions; otherwise the sizes vary by 1 in all dimensions where this doesn't apply.

for the node pool. Note that equation (3.47) converges monotonously decreasing towards N for $D \to \infty$ and reaches a maximum of $\frac{3}{2}N$ for the 1D case (the 0D case for point data is a concept alien to $\mathcal{R}as\mathcal{D}a\mathcal{M}an$ and meaningless regarding wavelets and the EZT anyway). In other words: given a constant number of wavelet coefficients, the size of the tree structure decreases as the number of dimensions increases, which is a property (almost) too good to hope for.

This tree structure provides no directly available information about a coefficient's position any more[26], nor links from child to parent nodes. Neither is required for coding with this tree, however, so there is no point in adding them to the structure except for debugging purposes.

Another important issue are boundary effects for data cubes with edges which aren't powers of 2 long. The wavelet transformation partitions data of length $2^k + 1$ into $2^{k-1} + 1$ $(= \lceil \frac{2^k+1}{2} \rceil)$ average coefficients and 2^{k-1} $(= \lfloor \frac{2^k+1}{2} \rfloor)$ detail coefficients; this can lead to coefficients unreachable by the zerotree structure in its current form, as will be shown in the following example.

Consider 1D data with length 18. This will be split into 9:9 coefficients (with the number of average coefficients before the colon and the number of detail coefficients after it). Recursively applying this to the average coefficients results in partitions 5:4 and then 3:2 where we can stop in this example. This gives us a 1D partition into bands 3,2,4,9 coefficients long, which can also be seen for the 2D case in figure 3.17. Let's focus on the two finest bands at scale level 2 and 3 with lengths 4 and 9 respectively: the parent-child node relationship in equation (3.44) says that coefficient 0 in band 2 has coefficients 0 and 1 in band 3 as children, up to coefficient 3 in band 2, whose children are coefficients 6 and 7 in band 3; the last of the 9 coefficients in band 3 is unreachable, however, which has to be remedied to make the EZT work for arbitrary quality coding.

First up, we'll have to examine whether the area covered by the average coefficients on the coarsest level always completely covers the area of the original data according to equation (3.44). Using equation (3.34), we can calculate which area in the original data the average coefficients after l coarsening steps correspond to, using $|a^l|$ for the width of the band containing these average coefficients. Because according to equation (3.44) the width of the original data covered by the coefficients doubles with each coarser scale level, the width covered by the $|a^l|$ coefficients after l coarsening steps is $2^l|a^l|$, i.e.

$$\begin{aligned} 2^l|a^l| &= 2^l \left(\sum_{i=l}^{k} 2^{i-l} j_i + \left\lceil \frac{1}{l} \sum_{i=0}^{l-1} j_i \right\rceil \right) = \sum_{i=l}^{k} 2^i j_i + 2^l \left\lceil \frac{1}{l} \sum_{i=0}^{l-1} j_i \right\rceil \\ &\geq \sum_{i=l}^{k} 2^i j_i + \sum_{i=0}^{l-1} 2^i j_i = \sum_{i=0}^{k} 2^i j_i = |a^0| \end{aligned} \tag{3.48}$$

[26] Apart from its value offset, which can be converted back into a multidimensional position vector using equation (2.2), but that is a rather expensive operation.

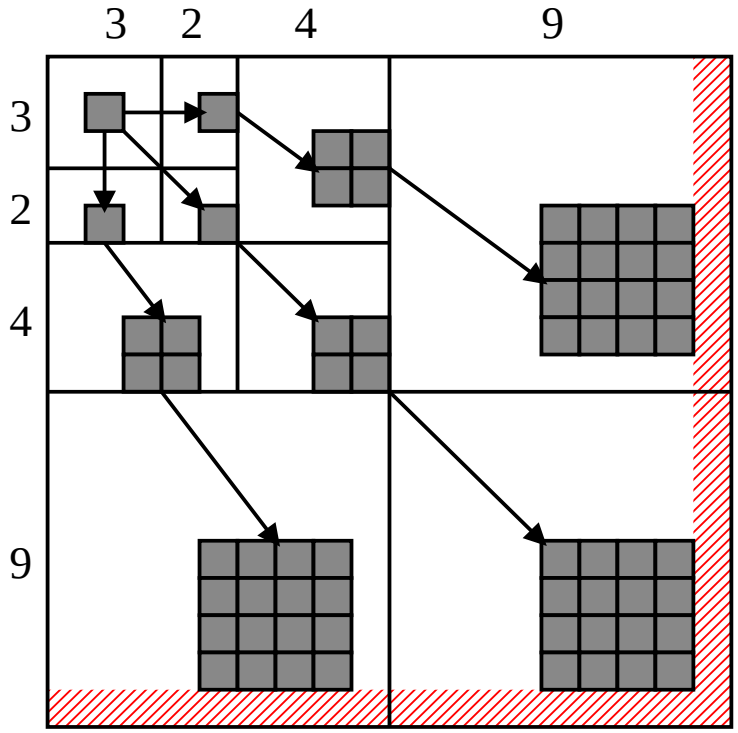


Figure 3.17: Zerotree boundary effects
This figure illustrates the problems with the EZT structure in case of data whose spatial extent is not a power of 2. It shows an array 18×18 *coefficients large and the band partition caused by the multiresolution wavelet analysis. If no special steps are taken, the hatched area is unreachable from the root band to the top left.*

because

$$2^l \left\lceil \frac{1}{l} \sum_{i=0}^{l-1} j_i \right\rceil = \begin{cases} 0 & j_0 = \cdots = j_{l-1} = 0 \\ 2^l & \text{otherwise} \end{cases}$$

and $2^l > \sum_{i=0}^{l-1} 2^i \geq \sum_{i=0}^{l-1} 2^i j_i$. That means that the original data is always covered completely by the average coefficients on any level, which naturally includes the coarsest one. The problem of unreachable nodes can therefore only be caused by some of the intermediate detail bands like in the example. A possible solution is therefore to virtually extend these detail bands so that the coarsest one is just as wide as the coarsest average band and each finer detail band is exactly twice as wide as its nearest coarser neighbour. Nodes which fall outside the actual limits of the band are purely virtual, have no node value and are merely present to allow linking to coefficients on finer levels; a node with no value can also be encoded easily with the self-describing integers introduced above by simply using `noChild` as the value reference. In the example, the bands would therefore have virtual widths $3, 3, 6, 12$ as opposed to the actual widths $3, 2, 4, 9$, and a possible route from the root of this virtual band partition to the last coefficient on the finest level (which used

to be unreachable) is $(c1, 2) \rightarrow \boxed{(d1, 2)} \rightarrow \boxed{(d2, 4)} \rightarrow (d3, 8)$, using the notation (*band-label*, *coefficient-number*) and framing purely virtual nodes. This is the approach currently implemented; it requires more memory due to the virtual band extensions, but allows efficient algorithms without too many exceptions by ensuring the tree has constant depth everywhere.

Shapiro's EZT publication also suggested using a list for the coordinates of nodes that haven't been encoded as significant yet and a list for the values of nodes that have been encoded as significant already. Because the additional memory requirements for these lists can become quite large (and one would even scale with the dimensionality because it contains coordinates), a different implementation was chosen using a binary array (*encoding map*) with the same size as the coefficient array, which signifies for each coefficient whether it has already been encoded as significant in a dominant pass or not. This requires one additional bit per coefficient, regardless of dimensionality, which is very little considering the zerotree algorithm operates on arrays of 64 bit floating point values. Instead of the list of node values already encoded, the coefficient array itself can be used and updated with each encoded node; this modifies the coefficient array, but that isn't required by any part of the compression engine any more once encoding has finished. While this approach slows down the subordinate pass for low quality coding where the number of encoded nodes is small compared to the number of coefficients (and therefore the iteration over the encoding map takes longer than the iteration over the explicit subordinate list), it performs well for higher quality and uses considerably less memory especially in the latter case. By casting the bit array to a byte array and iterating over that in the subordinate pass, one can even get a simple hierarchical index to encoded nodes at *no extra cost* by testing each byte against 0 and only checking the individual bits if the byte differs from 0, which can speed up the search for already encoded nodes by up to a factor of 8. Additional hierarchical levels could be added, but that would also increase the cost of marking a node as encoded, and because the complexity of the compression stream is still considerably higher than that of the subordinate pass, there is currently no urgent need for such an extension.

3.4.3.6 Aggregation for More Efficient Encoding

An important aspect of zerotree coding is the complexity of encoding a node's value. Determining whether a node is a zerotree root or an isolated insignificant node requires checking all the node's transitive children. If no information about a node's children was aggregated, the encoding algorithm would have to spend most of its time on this check. There is tremendous room for optimization especially in this respect, because in case the node currently processed turns out not to be a zerotree root, its children have to be processed and their transitive children – the current node's transitive grandchildren, which may already have been checked – have to be checked for significance again. An obvious choice for aggregation is the maximum absolute value of each inner node's transitive child nodes. That way, each node can be encoded in constant time by using its aggregated value to distinguish between zerotree root and isolated zero.

There is one potential problem, namely that encoding a node can cause aggregated

values above it to become inconsistent, i.e. when the node currently processed has the largest absolute value of all its parent's transitive children. Due to the encoding order this doesn't matter, however: the aggregated values are only used if the currently processed node is insignificant to determine whether it is an isolated insignificant node or a zerotree root. In the second case, its child nodes don't have to be processed, so the aggregated values remain unchanged and therefore the subtree remains consistent. Otherwise, the child nodes are encoded, but the aggregated value at the currently processed node is not needed any more, because that node was already encoded before its children were processed, nor is it needed above the current position because the same argument applies to all nodes between the root and the current position as well. Note also that this is true regardless of whether depth-first or breadth-first recursion is used. In other words, the aggregated values don't have to be updated every time a node in the subtree is encoded but only before a new dominant pass is started, i.e. at most once for each threshold value – if no nodes in a subtree were encoded in the previous pass, no update of aggregated values within the subtree is necessary, of course. If this was not the case, aggregation would cause increasing overhead the closer the currently processed node is to the leaf nodes due to the length of the aggregation path (directed towards the root). As it stands, the aggregation overhead is relatively small; an analysis of its benefits based on the number of nodes that have to be visited follows.

Assuming a tree with height L, dimensionality D and valid child nodes everywhere, we first calculate the total number of nodes in a subtree with a coarsest level coefficient in the root node. Due to the fact that each node has 2^D children (again assuming the same number of child nodes for the root nodes for simplicity), the number of nodes at depth $l+1$ in the tree is 2^D times the number of nodes at depth l, and since there is one node at the root of the tree, the total number of nodes N is

$$N = \sum_{i=0}^{L-1} 2^{iD} = \frac{2^{LD} - 1}{2^D - 1}. \tag{3.49}$$

Now let's consider the worst case when encoding a subtree: all inner nodes are insignificant, only the leaf nodes are not. This means that for each inner node the encoding algorithm has to check all children to determine whether the node is zerotree root or isolated zero, and as all inner nodes are insignificant, this continues up to the leaf nodes. That means that starting from the root node at depth 0, all nodes below that depth have to be visited, then the same for depth $1, 2, \ldots$. The number N^l of nodes from depth l to L can be calculated as the number of nodes in the entire tree minus the number of nodes in depths 0 to $l-1$:

$$N^l = N - \sum_{i=0}^{l-1} 2^{iD} = N - \frac{2^{lD} - 1}{2^D - 1} = \frac{2^{LD} - 2^{lD}}{2^D - 1}. \tag{3.50}$$

The worst case situation described above means that first all child nodes from depth 1 downwards have to be visited, then all those from depth 2 downwards etc., up to depth L, i.e. the total number V of visited nodes is

$$\begin{aligned} V &= \sum_{l=1}^{L} N^l = \sum_{l=1}^{L} \frac{2^{LD} - 2^{lD}}{2^D - 1} = \frac{2^{LD}}{2^D - 1} L - \frac{1}{2^D - 1} \sum_{l=1}^{L} 2^{lD} \\ &= \frac{2^{LD}}{2^D - 1} L - \frac{2^D}{2^D - 1} \sum_{l=0}^{L-1} 2^{lD} = \frac{2^{LD}}{2^D - 1} L - 2^D \frac{2^{LD} - 1}{(2^D - 1)^2} \\ &= \frac{2^D}{2^D - 1} \left(2^{(L-1)D} L - \frac{2^{LD} - 1}{2^D - 1} \right). \end{aligned} \tag{3.51}$$

Now let's set V into relation with the number of nodes in the subtree to get the complexity ratio of the naive approach compared to preaggregation, where each tree node only has to be visited once during encoding (not counting aggregation, because that may not be necessary before each pass):

$$\begin{aligned} \frac{V}{N} &= \frac{\frac{2^D}{2^D-1} \left(2^{(L-1)D} L - \frac{2^{LD}-1}{2^D-1} \right)}{\frac{2^{LD}-1}{2^D-1}} \\ &= \frac{2^D}{2^{LD} - 1} \left(2^{(L-1)D} L - \frac{2^{LD} - 1}{2^D - 1} \right) \\ &= \frac{2^{LD}}{2^{LD} - 1} \left(L - \frac{2^D - 2^{(1-L)D}}{2^D - 1} \right). \end{aligned} \tag{3.52}$$

For $L \to \infty$ this ratio converges towards L, because $\frac{2^{LD}}{2^{LD}-1} \to 1$ and $\frac{2^D - 2^{(1-L)D}}{2^D-1} \to \frac{2^D}{2^D-1} \ll L$. In other words, if preaggregated values are used for all inner nodes, only about $\frac{1}{L}$th the number of nodes in the current subtree have to be visited in the best case. In the worst case, all nodes are significant and the preaggregated values are not used. This means that all nodes in the current subtree are visited once during preaggregation and once during encoding, thereby losing efficiency compared to the naive approach without aggregation. However, as can be seen by the **ztreeroot** symbol, the EZT is apparently optimized for predominantly insignificant nodes. Therefore, as a consequence of preaggregation, the performance improvement will outweigh the overhead penalty at least in those situations where the EZT achieves good compression (which is typically the case for smooth data where wavelets are a good model).

3.4.3.7 Encoding Variants and Alphabets

I will continue this section on the Generalized Zerotree with some comments on coding variants and tree alphabets. The two-pass approach chosen in the original EZT publication is not the only conceivable way to encode coefficients in a zerotree structure, which can also be seen by the introduction of different alphabets, e.g. in [10]. The currently implemented zerotree engine already supports two different coding methods based on the zerotree structure and an arbitrary number can be added under the same framework by

simply deriving new classes with specific `encode()` and `decode()` methods. The coding variants currently available are

band1: this is a straightforward one-pass coder, i.e. there is only a dominant pass and no distinction between nodes that have or have not been encoded yet. Significant nodes are encoded with the alphabet of the dominant pass, and if they were significant their absolute values are reduced by the current threshold value. Subsequent passes operate on all nodes and do not ignore nodes that have already been encoded as significant in a previous (dominant) pass.

band2: this is the original two-pass approach as published by Shapiro [49] and already described in section 3.4.3.2. Using different compression streams for dominant and subordinate pass doesn't integrate well with the current architecture, however, so everything is written to one stream using a six-symbol alphabet (the regular four symbols for the dominant pass plus two additional symbols for the subordinate pass).

Both variants do the simpler depth-first recursion, which should also perform better because it orders the nodes by the strength of their correlation as implied by the EZT parent-child node linkage. In order to support both coding types, the aggregation algorithm has to work differently: for **band1** it has to use all node values, regardless of whether the nodes have already been encoded as significant or not, whereas for **band2** it must assume a value of 0 for all nodes that have already been encoded as significant in a previous dominant pass, because these nodes have to be ignored in future dominant passes and are therefore always seen as insignificant. Neither approach universally performs better, which is the best justification for having both implemented. Experimental results comparing both will follow in section 4.3.2.

3.4.3.8 Termination Criteria for Encoding

Zerotree coding successively refines the approximated data, so the higher the number of iterations, the closer the reconstructed data is to the original, and the larger the number of symbols needed to describe the data. Termination criteria for the encoder must allow the user to define a tradeoff between the number of symbols used to describe the data and the approximation quality, similar to the well-known quality level used in e.g. JPEG encoding. There are several termination criteria available for the Generalized Zerotree, which are mostly based on distortion measures defined in section 3.4, namely

SNR coding: terminate encoding as soon as the SNR of the wavelet coefficients is higher than a threshold SNR the user has to specify via a compression parameter;

PSNR coding: like SNR coding, but using the peak signal-to-noise ratio instead;

Residual coding: terminate encoding as soon as the maximum absolute difference between the actual wavelet coefficients and their current approximation falls below a threshold residual ε (per cell) the user has to specify via a compression parameter.

These termination criteria apply to the wavelet coefficients themselves, not the reconstructed data where distortion can differ considerably (see section 3.4.1). All termination criteria can be evaluated efficiently during encoding by updating precalculated values each time a coefficient is refined. For SNR and PSNR coding, the (peak) energy is constant, but the sum of squared errors it is divided by has to be updated every time a node is refined. To this end, the sum of squared errors is initialized to the total energy of the data (i.e. initially SNR = 1), because the initial approximation of all coefficients is zero, and every time a coefficient v_i is refined from an old approximation v_i' to a new one v_i'', the sum of squared errors is updated by $(v_i'' - v_i)^2 - (v_i' - v_i)^2$, i.e. it is not necessary to fully calculate equations (3.35) or (3.36) every time the termination criterion needs to be checked, but it can be done in constant time on each update. For residual coding, it suffices to check the current coding threshold T rather than having to iterate over all nodes to find the maximum value, because by definition at any time during coding all absolute values in the tree are smaller than $2T$. Because of this, there is very little overhead required to check any of these termination criteria, so tests can be made frequently, which means that for instance SNR and PSNR values can usually be met quite closely without excessive cost for checking the termination condition.

3.5 Predictors

Predictors belong to the model layer of a compression engine, because they transform the data according to a model into an equivalent representation promising better compression rates, but do not perform any compression themselves. As the name implies, a predictor expresses a cell value relative to an approximated (=*predicted*) value. The approximate value of any cell can be calculated by both encoder and decoder from the cell values already processed (*prediction context*), and therefore doesn't have to be stored. Consequently, only the difference of the actual cell value from the predicted value requires any storage. Provided the predictor's model matches the data well, most of the differences will be zero or at least concentrated around zero, which can improve compression considerably. On the other hand, if the model doesn't match the data, the differences will be large and may even compress worse than the original cell values. Because MDD represent a large variety of data types which don't follow a common data model[27], it is necessary to have a large number of predictor models available to be able to choose the best one for each MDD type. Predictors can be classified into two major kinds, interchannel (section 3.5.1) and intrachannel (section 3.5.2) predictors, which are orthogonal concepts and can therefore be combined arbitrarily. Not all predictors fall in these two categories, but those normally used in the compression context do. The names are based on the definition of a channel as all data belonging to the same base type within an MDD, so intrachannel predictors operate within one channel, whereas interchannel predictors operate across channels. Both intrachannel and interchannel predictors derive from a common abstract base class predictor

[27] e.g. volume data, which is typically smooth (small differences between neighbouring cell values), in contrast to abstract OLAP data which is rarely smooth (large differences between neighbouring cell values).

and both use neighbouring cell values for prediction, where the neighbourhood is either spatial (intrachannel) or across channels within each cell (interchannel); an overview of the predictor class hierarchy can be seen in figure 3.18. The different predictor types will be covered in the next two sections, followed by details on how predictors were integrated into the MDD compression engine and problems to take into consideration when using predictors in combination with lossy compression.

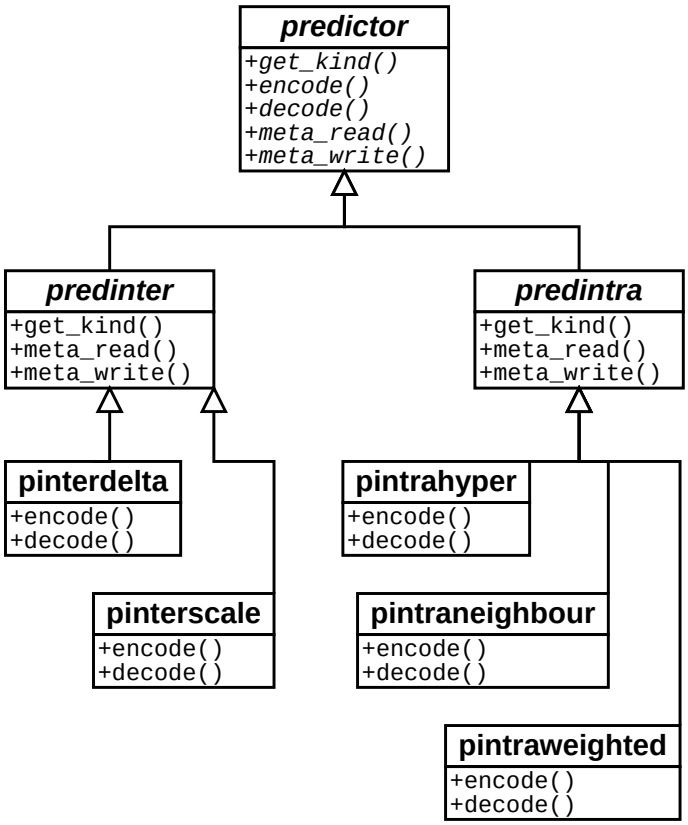


Figure 3.18: The predictor class hierarchy

There have to be some comments on the effect of calculating differences between integer values, because the range of the difference value is twice the range of the original values: for example when using 8 bit signed values 127 and -128, their difference is $127-(-128)=255$, which can only be represented as a 9 bit signed value, not an 8 bit one. But the predictors were designed to use the same data types for input and output, because this allows adding them as optional functionality at a very fundamental level in the architecture without having to make subsequent compression stages aware of the presence of predictors, since the data structure presented to them is the same with or without a predictor run. Fortunately, modulo effects cancel each other out, just like with Haar wavelets (see section 3.3.4.1), because when only the B least significant bits are of concern, aliasing errors of the difference values do not influence the B least significant bits of the result, since in case of an overflow the value appears shifted by $\pm 2^B$ and $(\pm 2^B) \bmod 2^B = 0$. For instance in the

above example, the value 255 would appear as -1 in 8 bit signed integer representation and $(127 - (-1)) \bmod 256 = 128 \bmod 256 = -128 \bmod 256 = (127 - 255) \bmod 256$. It is important to note that due to this, the sign bit of the difference value becomes meaningless, i.e. a negative difference value does not imply that the first value is smaller than the second one (see example). This effect causes considerable problems when using predictors in lossy compression, which will be covered in more depth in section 3.5.4.

Note that the current system does not allow changing the predictor type during the compression of one tile, so it isn't possible to switch between predictors for different channels. This has mostly to do with the way predictors were added to the compression engine, more about which will follow in section 3.5.3.

Predictors are sometimes also referred to as *filters* (e.g. in the context of the PNG format [41]), but that term will not be used in this work to avoid confusion with the wavelet filters covered earlier. Another term for compression with specific predictors commonly used in literature is *delta compression*, e.g. [22]; in order to make a clear distinction between the generation of the deltas and the compression of these deltas, the term predictor is used synonymously for the delta *generator*.

3.5.1 Interchannel Predictors

Interchannel predictors approximate the value of a channel at a given position from the values of the other channels at the same position. This type of predictor can improve compression considerably if the channels are correlated. A related application is the RGB $\rightarrow$ YUV transformation used in imaging [40]

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{3.53}$$

and its corresponding inverse operation. In YUV space, colours are modelled as overall brightness information Y (luminance) and two colour differences U,V (chrominance); the effects are that

- channels are correlated via their luminance, which typically concentrates most of the energy. Y essentially acts as the dominant predictor value (and green is the dominant contributor to the luminance);
- because the human eye is much less receptive to colour changes than to brightness changes, chrominance can be encoded with less precision than luminance without a noticeable impact on image quality.

Similar effects can be achieved with other data types as well, not just images. For instance temperature and pressure in a fluid flow simulation are often correlated, so there are also potential compression gains from using prediction for more generic MDD than just raster images.

Interchannel predictors provide a framework for correlating channels in a way similar to the above example. However, transformations like RGB $\leftrightarrow$ YUV require transformation matrices in $\mathbb{R}^{n_c \times n_c}$ (n_c = number of channels) for transformation and inverse transformation, and the resulting data types are real numbers rather than the original data types, which makes the addition of *optional* prediction to the compression engine complicated (because the data structure would be different if predictors were used) and can introduce loss due to numeric errors. In addition, specifying such a transformation matrix manually is complicated for the user, especially since the matrix must also have an inverse to allow reconstructing the original data in inverse prediction. A specialized RGB $\leftrightarrow$ YUV transformation is recommended for the future work section, but in this thesis the focus was on generic solutions. Therefore, the currently implemented system uses the simpler approach of user-defined channel correlations where each channel can be predicted by another channel (in contrast to a set of other channels) and the predicted channel's value is expressed as a predictor function $f_p(v_1, v_2)$ over both channels' values at the same position. In the simplest case, this is just the difference ($f_p(v_1, v_2) = v_1 - v_2$), which corresponds to a transformation matrix $(p_{i,j})$ with $p_{i,i} = 1$ and $p_{i,j} = -1$ if channel j predicts channel i, 0 otherwise. This matrix can be represented by n_c numbers which specify the predicting channel for each channel (with a special number for disabling prediction for a channel), and the error of applying the transformation is the machine precision, because the matrix coefficients are integers and at most one subtraction is performed per value (both things are true for the inverse transformation matrix as well, but with additions instead of subtractions). Predictor functions can also be more complex, however (see below). As a further restriction, only channels which have the same base type may be correlated, but because a correlation of channels which do not even have the same base type is very unlikely, this should not introduce complications in practical terms. Note that adding more complex transformations under the same framework is a trivial extension, i.e. the restrictions mentioned above are in the current implementation only rather than the design.

Because the deltas overwrite the predicted channel's values, the current interchannel predictors have to translate the channel correlation information into encoding/decoding channel orders. During encoding, channels must be processed in an order that ensures that if channel i is predicted by channel j, then i comes before j for all i, j; so if in an RGB image red is predicted by green and blue by red, then the order must be blue, red, green. During decoding, on the other hand, that order must be reversed, i.e. if a channel i is predicted by a channel j, then j comes before i, and using the same RGB example the order must be green, red, blue. Channel mapping and ordering is done in predinter scope, whereas derived classes provide the actual prediction codecs for all atomic base types. Currently available interchannel predictors are

pinterdelta: $f_p(v_1, v_2) = v_1 - v_2$, the predicted channel's values are stored as the differences to the predicting channel's values. This type of prediction is most efficient when the values of both channels cover a similar range, such as the colours in an RGB image and in contrast to the case where one channel's values are multiples of another channel's values. pinterdelta is lossless for integer types and can introduce

an error with a magnitude of the machine precision for floating point types.

pinterscale: $f_p(v_1, v_2) = (v_1 - m)s - v_2$, the predicted channel's values are first translated and scaled such that they cover the same range as the predicting channel's values ($v_1' = (v_1 - m)s$) and then stored as differences to the predicting channel's values ($= v_1' - v_2$). With the extreme values $v_{1,\max}, v_{1,\min}, v_{2,\max}, v_{2,\min}$ for both channels and the boundary conditions $f_p(v_{1,\min}, v_{2,\min}) = f_p(v_{1,\max}, v_{2,\max}) = 0$ we get

$$m = \frac{v_{1,\min}v_{2,\max} - v_{1,\max}v_{2,\min}}{v_{2,\max} - v_{2,\min}} \quad \text{and} \quad s = \frac{v_{2,\max} - v_{2,\min}}{v_{1,\max} - v_{1,\min}},$$

both of which are stored as predictor meta data for the inverse operation during decompression. This type of prediction is intended for channels which cover different ranges but are nonetheless correlated, for instance pressure and temperature in simulation data whose numeric ranges depend on the units used, but where typically an increase in pressure implies an increase in temperature. pinterscale is potentially lossy for all types due to floating point operations in the inverse prediction $f_p^{-1}(f_p(v_1, v_2), v_2) = \frac{1}{s}(f_p(v_1, v_2) + v_2) + m$; because these floating point operations are done in double precision, there is usually only loss for MDD over the double type, however.

While these predictors can't be expected to perform equally well as specialized image channel predictors due to their generality, they can be applied to far more generic data than RGB images, e.g. multichannel satellite image data as described in [57], for example.

3.5.2 Intrachannel Predictors

Intrachannel predictors approximate the values of a channel from the values of neighbouring cells in the same channel, which typically performs well if the data is smooth. That means intrachannel predictors exploit similar data properties as wavelet transformations or DCT do, but at a simpler level. In contrast to these transformations, intrachannel prediction is typically lossless. All intrachannel predictors in the compression engine derive from a common parent class predintra which manages information about which channels to use prediction on (saved as meta data for the decoder), whereas the actual prediction codecs are provided by derived classes. Naturally, there is a huge number of potential algorithms for intrachannel prediction, three of which have been implemented for evaluation and will be described below. Note that because the predictors overwrite the cell values with the differences to their predicted values, the order in which cells are iterated over during decoding must be the inverse order that was used during encoding for all intrachannel predictor types.

pintrahyper: this variant uses (orthogonal) hyperplanes for prediction, i.e. it slices through the MDD along the hyperplane normal and expresses the values in the currently processed hyperplane as differences to the nearest unprocessed hyperplane. An

orthogonal hyperplane is described by its normal dimension and the coordinate x_h where the hyperplane intersects this dimension's axis. The current implementation allows the user to specify the number of the normal dimension in case the cells are correlated stronger along this direction (such as the temporal dimension in a movie); this dimension number is stored as meta data for the decoder.

In the popular case of 2D images, hyperplanes are lines or columns of the image and hyperplane predictors can be found in e.g. the *delta-line* raster data format of the printer language PCL [38], which performs very well for rasterized text and similarily regular data, or some of the filters of the PNG image format [41] (filter types `SUB` for horizontal and `UP` for vertical prediction direction).

pintraneighbour: this predictor uses the arithmetic average of the values of neighbouring cells in all directions as approximation for a cell value; the cells used for prediction are limited to those whose coordinates differ by at most 1 in all dimensions (this is a cube containing $3^D - 1$ cells) and which have known values for the decoder to allow reconstructing the original (which eliminates some of these $3^D - 1$ cells, as explained below). The current implementation iterates back-to-front in all dimensions during encoding, so it has to iterate front-to-back in all dimensions during decoding. The order in which dimensions are iterated over is important as well to determine the maximum set of neighbouring cells with values known by the decoder: the default iteration order is last dimension first and first dimension last, which means that for example for a cell at $(x_1, \ldots, x_D)$, all cells at $(x_1 - 1, x_2 + z_2, \ldots, x_D + z_D)$ have values known to the decoder (where $z_i \in \{-1, 0, 1\}$). Cells with coordinates $(x_1 + 1, \ldots)$ don't qualify at all, and for cells with coordinates $(x_1, \ldots)$ the set of neighbouring cells with known values are at $(x_1, x_2 - 1, x_3 + z'_3, \ldots, x_D + z'_D)$ as well as $(x_1, x_2, x_3 - 1, x_4 + z''_4, \ldots, x_D + z''_D)$ and so forth up to the final cell at $(x_1, \ldots, x_{D-1}, x_D - 1)$. Formally, the coordinates of neighbouring cells with values known by the decoder are in the set $S_N(x_1, \ldots, x_D) = \{(x_1 + z_1, \ldots, x_D + z_D) : z_1 = \cdots = z_{i-1} = 0, \quad z_i = -1, \quad z_{i+1}, \ldots, z_D \in \{-1, 0, 1\}\}$. These sets are visualized for the 2D and 3D cases in figure 3.19.

Similar predictors can frequently be found in image compression, such as the `AVERAGE` and `PAETH` filters in the PNG standard [41], or to some extent the Q-coder in the lossless JBIG format [30] (which has a bigger prediction context, however). The pintraneighbour predictor uses the same weight for all neighbouring cell values irrespective of their relative position by calculating the arithmetic average value and rounding that to the nearest value that can be represented by the base type of the currently processed channel; because this rounded average, which is used as predictor value, is the same in encoder and decoder, the predictor is lossless (at least for integer types) even though the average value is not.

pintraweighted: this predictor works exactly like pintraneighbour, but uses different weights depending on the neighbouring cells' relative positions. The policy used is that the weight of a neighbouring cell is initialized with 1 and multiplied by a

`weightFactor` for every dimension where the coordinate differs from that of the currently predicted cell; `weightFactor` is $\frac{1}{2}$ by default but can be changed by the user via parameters. With this default value, the weight would be $\frac{1}{2}$ for the cell at relative position $(0, -1, 0)$, $\frac{1}{4}$ for the cell at relative position $(-1, 0, 1)$ and $\frac{1}{8}$ for the cell at relative position $(-1, 1, 1)$. This system ensures that diagonal neighbours have less influence on the predicted value than direct neighbours. With these weights w_i and the neighbouring cell values v_i, the predicted value is then calculated as the weighted average $\sum_i v_i w_i / \sum_i w_i$. Like in the pinterweighted predictor, calculating this average involves loss, but because the result is rounded to the same value by both encoder and decoder when determining the difference value, the predictor as a whole is lossless (for integer types).

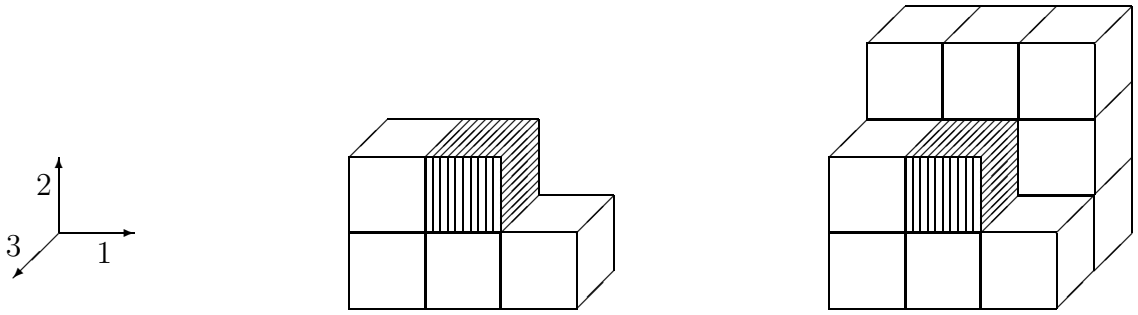


Figure 3.19: Neighbouring cells for some intrachannel predictors
This figure shows the maximum sets of neighbouring cells with a maximum distance of 1 in all dimensions and values known to the decoder for pintraneighbour and predintraweighted. To the left, axis orientation and iteration order of the decoder are shown, i.e. first horizontal (left to right), then vertical (bottom to top), followed by the normal to the paper (back to front) in the 3D case. In the center, the neighbourhood is shown for the 2D case and to the right for the 3D case; in both cases the currently processed cell is visualized hatched.

3.5.3 Predictors in the Compression Engine

As shown in figure 3.4 on page 37, predictors were added to the compression engine at a very high level in the class hierarchy within the first two child classes of the tilecompression root class. The reason for this decision was to make prediction uniformly available to all compression types without requiring special support by derived classes (albeit special steps must be taken for lossy compression, see section 3.5.4). Because interchannel prediction is a concept not covered at all by other components of the engine, the compression class tilecompinter with interchannel prediction comes first in the hierarchy and is shared by all compression types. The situation is somewhat different for intrachannel prediction, because wavelet transformations cover similar ground and combining both will therefore typically not improve compression, hence the wavelet class hierarchy derives directly from tilecompinter rather than tilecompredict, which contains both predictor types.

The current implementation only supports one predictor of each type per tile, so changing the predictor between channels is not an option. It is easily possible to extend tilecompinter and tilecompredict to support an arbitrary number of predictors, which would allow using different predictors for channels which are not correlated, because in this case each predictor can process all channels it controls in one pass, as none of these channels depends on ones controlled by other predictors. What can not be done with the mere addition of predictors to the tilecompression classes are channel correlations like 1→2, 2→3 and 3→4 where channels 1 and 3 are controlled by predictor P_1 and channels 2 and 4 are controlled by predictor P_2, because neither predictor can process all its channels independently. This sort of extension is more complicated and would require extending the predictors themselves. However, it seems unlikely that channels using different prediction strategies can be correlated, therefore this extension is directed to the future work section.

Predictors were added such that for lossless compression, they preprocess the data at the beginning of the `compress()` method, with interchannel prediction coming first and intrachannel prediction (if applicable) afterwards. The processed data (which has the same structure as the original data) is then passed to the compression object used via the virtual `do_compress()` method, which performs the actual compression without being aware of whether predictors were used or not. Predictor types and their meta data are stored along with the other modules' meta data in the compressed tile. For decompression, this process is reversed, i.e. first the predictor types and meta data are extracted from the compressed tile, then the data is decompressed by the compression object used with the `do_decompress()` method, then inverse intrachannel prediction (if applicable) followed by inverse interchannel prediction is applied, which results in the reconstruction of the original data in the lossless case. For lossy compression the use of predictors is more complicated and will be covered in the next section.

3.5.4 Predictors and Lossy Compression

Because the only lossy compression technique currently available is the wavelet engine, this section will concentrate on wavelets and prediction, although similar effects can appear in other lossy techniques (e.g. DCT). The main problem concerning the use of predictors in lossy compression is overflows for integer types when decoding: after the inverse wavelet transformation was applied, the values in the resulting floating point array don't necessarily fit within the range of the (integer) base type of the data due to lossy coding (see the quantizing wavelets in section 3.3.4.2), therefore the values are restricted to the range of the base type when this floating point array is converted back into an array over the actual base type to avoid aliasing errors. If predictors were used at the same point in the compression engine for lossless and lossy compression techniques, the following effect could occur for integer types:

A large value v_i is replaced by its difference $\delta = v_i - p_i$ to a predicted value p_i during prediction; the resulting data is then converted to a floating point array, wavelet-transformed and compressed with loss. During decoding, the data is first decompressed into a floating point array of wavelet coefficients, which is inverse-transformed into (an approximation of)

the original data in floating point representation and then converted into an array over the corresponding base type (at this stage the value range is restricted to that of the base type). However, the reconstructed values δ'_i and p'_i may differ from the ones used during encoding due to loss. In particular, it is possible that $p_i + \delta_i$ lies within the value range of the base type used, but the reconstructed value $v'_i = p'_i + \delta'_i$ falls outside of it and is aliased to another value (or the other way around in case of intentional overflows mentioned at the beginning of this section), e.g. for 8 bit unsigned types $v = 255$ but $v'_i = 257$ which gets aliased to 1. This kind of overflow can not be trapped, because as mentioned in section 3.5, the restriction of the difference value to the same number of bits as the base type renders the sign of the difference value meaningless, therefore it can't be decided whether an overflow is intentional as in the example at the beginning of this section, or an unwanted side-effect of the lossy compression. It is therefore not possible to restrict v'_i to the value range of the base type without maintaining the correct signs of the difference values in some way. Consider the example at the beginning of section 3.5, where a difference of 255 appeared as -1 in 8 bit signed representation, which was then subtracted from 127, resulting in 128, which appeared as -128 in 8 bit signed representation: if the difference value -1 was changed to 0 due to loss, the resulting error would obviously be unacceptably large.

The sign bits could be stored in a separate array; this doesn't integrate well with the compression engine and makes reconstructing the correctly signed values somewhat complicated. The alternative is to activate predictors at another point in the engine, within the compression class itself rather than outside of it. In the wavelet engine, the best place to do it is between wavelet transformation and the conversion *base type array* $\leftrightarrow$ *floating point array* (see section 3.3.4.2). Because in this case the prediction is applied to floating point values, no overflows and aliasing effects can occur[28] and the differences always have the correct sign; the differences may not be exact due to limited precision of the mantissa, but this effect is usually negligible compared to the compression-induced loss. The disadvantages of this approach are that it requires explicit support for predictors in all lossy compression classes as well as higher memory requirements, because the floating point arrays of channels predicting others have to be kept in memory until all channels predicted by them have been processed. Despite these problems, this is the most elegant approach, although it has to sacrifice the independence of the compression algorithms from the predictors that exists in the lossless case. Whether a compression algorithm is lossy or lossless is determined by a virtual function call to the tilecompression object; if it's lossless, predictors are handled in tilecompinter and tilecompredict scope and the compression algorithm can ignore prediction entirely, otherwise prediction has to be handled by the compression object internally.

[28]Bar pathological cases near the extreme values of the floating point types, which have little relevance in real life applications.

3.6 Dynamic Parameter System

Many compression algorithms have to be configured by the user via parameters, like for instance the compression level in *ZLib*, which is an integer value representing the tradeoff of time spent vs. compression rate[29]; other examples are quality settings like in JPEG or the signal-to-noise ratios to use for each channel in wavelet compression. These examples already illustrate the diversity of parameter types found in a complex compression engine, which makes it unfeasible to use fixed parameter types. For maximum flexibility, a dynamic parameter system is needed that is both easy to use and powerful enough to cover the requirements of each and every module within the compression engine[30].

The obvious choice regarding the flexibility required for the parameters is to encode them as strings, because any parameter type can be converted to some string representation and back. Since configuring all modules involved in the compression of an MDD calls for multiple parameters, there must be a way to identify each parameter, for instance via a keyword; therefore the parameter string format chosen is the following:

```
param_string  =  param_def [ , param_def ] *
   param_def  =  keyword = value
       value  =  integer | double | substring | "string"
     keyword  =  alphanumeric identifier name
     integer  =  string representation of integer value
      double  =  string representation of floating point value
   substring  =  any string not containing comma or whitespace
      string  =  any string
```

Values can currently have three different types: integer, floating point and string. Basically using strings as types and leaving the conversion to the module using the parameter in question would be sufficient, but because a large number of parameters are integers and floating point values (see appendix C), pushing the conversion into the parameter parser is often more convenient. String values can be in two formats: if the value contains whitespace characters or commas, it must be enclosed in double quotes to allow the parser to tell the difference between the end of the value and the beginning of the next keyword. Otherwise, the double quotes are redundant and may be omitted.

The parameter system is based on a class parseparams, which realizes the scanning of parameter strings for relevant parameters. Each module requiring parameters instantiates an object of this class and registers each parameter it understands with a keyword, a type descriptor and a reference to the variable used to store the parameter value, which is typically done in the constructor. Within a class hierarchy, the parseparams object is typically instantiated at root level and shared by all descendants, which merely have to register their parameters one by one; for instance the parameters used in an object of the daub4

[29] Legal values range from 0 to 9; with a level of 0, data is not compressed at all and the larger the value the harder the algorithm tries to improve the compression rate at the expense of getting slower.

[30] Use of the dynamic parameter system is not restricted to the compression module, but also extends to the related `conversion` module which handles data exchange formats like JPEG.

class (see figure 3.4 on page 37) were added to a parseparams object defined in tilecompression scope on all levels of the class hierarchy, in the constructors of classes tilecompression, tilecompinter, waveletcomp, qwaveletcomp and daub4. That way, each parseparams object maintains a list of keywords to scan for, depending on the module owning the object. When a parameter string is presented to such a module, it simply calls the parseparams object's `process()` method, which scans the parameter string for the registered keywords, extracting the values of keywords that are in its list and ignoring all others. When the call returns, the parameter variables on all levels of the class hierarchy contain the values found in the parameter string (or remain unchanged if no new value was given). This system allows using just one parameter string for all modules and leaving it to the individual modules to extract those values they consider relevant. For instance, the parameter with the keyword `wavestr` contains the name (a string) of the compression stream to use in wavelet compression, whereas the parameter with the keyword `zlevel` contains the *ZLib* compression level (an integer). Therefore, when the parameter string `wavestr=zlib, zlevel=9` is processed by a wavelet compression object, it will extract the name of the compression stream, use that to instantiate a *ZLib* stream in waveletcomp scope and pass the parameter string on to this newly created object, which will in turn extract the compression level 9. If the parameter string is `wavestr=rle, zlevel=9`, on the other hand, the `zlevel` parameter will be ignored entirely because rlestream doesn't understand it.

The current system operates directly on the parameter strings, which has the disadvantage of having to parse the string repeatedly on each call to the `process()` method[31]. Because the time taken for this call is usually very small compared to compression times, optimizing the parameter handling will not have much influence on the overall performance and is therefore assigned to future work. A possible way to speed up parameter processing is parsing the string once, building a table of all keywords found including a suitable index on it (e.g. hashing or alphabetic sorting) and subsequently scanning this table rather than the parameter string itself for parameter definitions.

3.7 Transfer Compression

An important requirement for the compression engine is that it can also be used for transfer compression, i.e. the compression of data in the client-server communication layer as shown in figure 3.4 on page 37 to reduce the data volume and consequently the transfer time. To achieve maximum efficiency and flexibility, this requires the same compression functionality on client and server, because that way tiles that are already stored in the database in a compressed format and need no trimming can be sent directly to the client without requiring decompression followed by compression with a dedicated transfer compression format, thereby saving substantial time. As this example illustrates, the speedup achievable

[31] Because normally only one parseparams object is used in a class hierarchy, there is typically no need for more than one call to its `process()` method; but in most cases several class hierarchies are involved in the compression of a tile, for example qwaveletcomp, wavequant, linstream and quantctrl or zerotree for lossy wavelets, each of which has to call `process()` once.

with transfer compression is governed by the following factors:

1. the compression time t_c on the sending side and the decompression time t_d on the receiving side (where the sender is not necessarily the server);

2. the compression ratio r achieved by the compression format chosen, defined as the size of the compressed data divided by the uncompressed size. As a general rule, the uncompressed data is used in preference if the compressed data is not shorter, which means we can always assume $0 < r < 1$;

3. the bandwidth B of the communication channel, i.e. the time taken to transfer data over the channel; in this section the unit used will be bytes per second.

Assuming uncompressed data with a length of m bytes, this results in total transfer times $\frac{m}{B}$ without transfer compression and $t_c + \frac{mr}{B} + t_d$ with sequential transfer compression, respectively $\max(t_c, t_d) + \frac{mr}{B}$ if compression and decompression can run in parallel; the abbreviation t_{cd} will be used to mean either $t_c + t_d$ or $\max(t_c, t_d)$ depending on whether compression and decompression are running sequentially or in parallel. The term $\frac{mr}{B}$ is the time taken to transfer the compressed data over the communication channel and since $0 < r < 1$ (only use the compressed data if it didn't expand in size), this is always less than the transfer time for the uncompressed data; consequently, the overhead for compression and decompression is the decisive factor when determining the feasibility of using transfer compression. Total transfer times benefit from transfer compression if $\frac{m}{B} > t_{cd} + \frac{mr}{B} \Leftrightarrow t_{cd} < \frac{m}{B}(1 - r)$. That means transfer compression can pay off if the compression ratio, the (de)compression times or the bandwidth are very small; small compression ratio and (de)compression times are mutually exclusive, because typically compression algorithms that achieve a substantial compression ratio are very complex and therefore expensive (see timing measurements in chapter 4), whereas fast and simple compression algorithms can't reduce the data size as much. Therefore the best compromise between high (de)compression overhead and low transfer times vs. low (de)compression overhead and high transfer times must be found, which means that the ratio of processing power to bandwidth is the decisive factor for transfer compression, and the larger this ratio, the more transfer compression speeds up transfer times[32], i.e. provided the bandwidth is small enough compared to the processing power, even very complex compression algorithms improve total transfer time ($B < \frac{m(1-r)}{t_{cd}}$). For instance over 100 MBit/s ethernet, even simple compression algorithms are unlikely to improve transfer times with processing power typically available today[33];

[32]The processing power could be measured in MIPS, but because no concrete threshold value for the power-to-bandwidth ratio is given in this section, any other unit will do. An important observation is that if a specific compression algorithm improves transfer times for a given reference ratio of processing power to bandwidth, this is also true for all combinations of these two factors whose ratio is at least as large as this reference ratio.

[33]Since the compression engine is tile-based, there is much potential regarding massively parallel architectures which – by compressing n tiles in parallel – can achieve near linear speedup. This is very important regarding storage compression of very large MDD, but it usually does not apply to transfer compression,

over 10 MBit/s ethernet, simple compression algorithms like *RLE* can already improve transfer times considerably, whereas over typical modem connections with about 50 kBit/s, even very complex compression algorithms like the wavelet-based techniques in section 3.3 can reduce total transfer times. The effects on overall performance of using the available compression algorithms for transfer compression will follow in the results chapter 4.4.

since in this case both sender and receiver must have similar processing power to avoid blocking. While a powerful, parallel database server is not unusual, the same can not be said for database clients, which therefore become the bottleneck.

Chapter 4

Evaluation and Results

In this section, the compression engine's major components will be tested and compared when applied to various kinds of MDD with 2–4 dimensions. The test MDD used, some of their properties and measuring conventions will be introduced in section 4.1. Next comes detailed analysis of lossless and lossy compression algorithms regarding runtime overheads and achievable compression rates in sections 4.2 and 4.3, both of which will end with a small conclusions section analysing the individual tests from a global perspective. This chapter closes with some measurements for transfer compression in section 4.4.

4.1 Test MDD and Conventions

The test MDD used for compression were chosen to cover a wide range of different types and data properties; they range from 2–4 dimensions and cover the base types `char`, `unsigned short`, `float` and `RGB` (= `struct { char red, char green, char blue }`). Each MDD will be given a name which will be used to identify it for the remainder of this chapter. All test MDD were processed as single tiles to minimize side-effects. The test MDD whose names end in "*_small*" were scaled to half their size in each dimension to make them more easily manageable: for instance the full tomogram (eight times the size of *tomo_small*) contains about 10 million cells, so for lossy wavelets the coefficient array alone would consume 80MB due to the change from `char` to the `double` base type). Figure 4.1 shows images of all test MDD except for *lena*, which can be seen in figure 3.5 on page 42.

2D: the 2D MDD are regular raster images; many of the compression techniques used in the engine originated in image compression, so raster images were chosen as a reference to compare how these techniques scale to the other MDD with higher dimensionality and different base types.

lena512: the standard greyscale *Lena* image (see figure 3.5) with the base type `char` and the spatial domain `[0:511,0:511]` (262144 bytes);

cnig: a colour satellite image of the spanish coastline with base type `RGB` and the spatial domain `[0:511,0:511]` (786432 bytes);

3D: the 3D MDD are spatial and spatio-temporal data typically used in entertainment, neuroscience, simulation and high performance computing.

tomo_small: a greyscale tomogram with base type `char` and the spatial domain `[0:127, 0:127, 0:76]` (1261568 bytes). The data is sparse, but contains a fair amount of noise;

brain_small: a greyscale tomogram with base type `unsigned short` and the spatial domain `[-72:-2,-75:16,-76:-2]` (979800 bytes). This MDD represents a standard brain scan as used by neuroscientists in e.g. the *NeuroGenerator* project [43];

movie_small: a short video sequence (i.e. 3D spatio-temporal data) with the base type `RGB` and the spatial domain `[0:59,0:79,0:59]` (864000 bytes). The first dimension is the temporal axis. It shows a scene from *The Jungle Book* © the Disney Corporation;

temperature: spatio-temporal data with the base type `float` and the spatial domain `[0:119,0:63,0:127]` (3932160 bytes). It shows the temperature distribution on a map of the earth over the seasons, where the first dimension is the temporal axis. The MDD was provided by *Deutsches Klimarechenzentrum* (*DKRZ*);

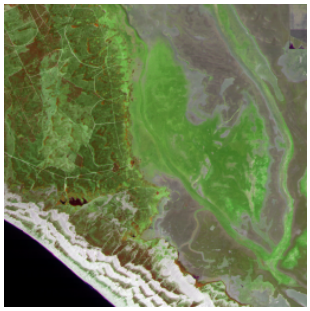
4D: there is only one 4D MDD available for testing, because there is a very limited supply of test data with this dimensionality suitable for all compression techniques (bar synthetically generated ones).

dkrz4d: spatio-temporal data with the base type `float` and the spatial domain `[0:14,0:31,0:31,0:63]` (3932160 bytes). It shows the temperature distribution in a volume over the seasons, where the second dimension is the temporal axis. This MDD is also courtesy of *DKRZ*.

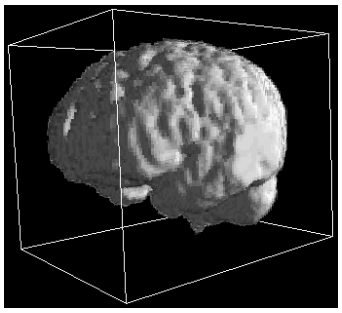
The conventions for all measurements are the following:

relsize (size of compressed data) are given in percent relative to the uncompressed size, i.e. a *relsize* of 100 means the compressed data is just as large as the uncompressed data, and the smaller the size, the better. The *relsize* is the *rate* times the size of the base type in bits times the number of cells in the MDD, in other words it scales proportionally with the rate. If compression algorithms are compared qualitatively, rate and *relsize* are equivalent; for quantitative comparisons, *relsize* is more meaningful in MDD compression, however. Note that the sizes given include any additional meta data that may be necessary for decompression;

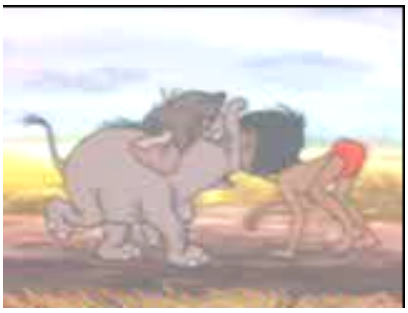
timings are given in microseconds (μs) unless noted otherwise. The abbreviations t_c and t_d are used for compression and decompression times. All timings were performed on a Sun Ultra 250 Enterprise Server with 640MB of main memory and two 400MHz *UltraSPARC-II* CPUs and are averages of at least 10 consecutive runs;



cnig

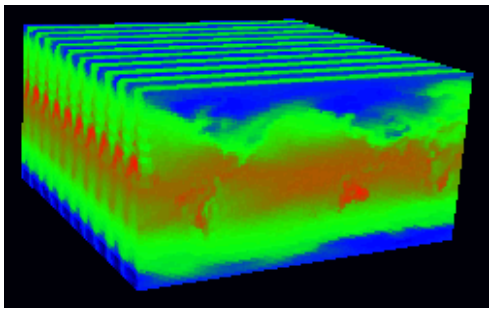


brain_small

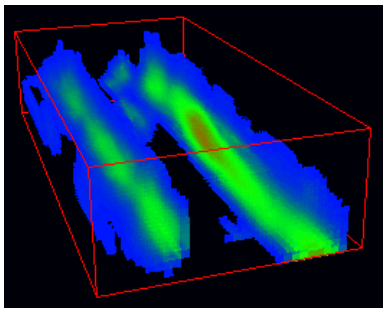


movie_small

tomo_small



temperature



dkrz4d

Figure 4.1: The test data sets

This figure shows images of the test MDD used in this chapter, visualized with rView *[19]. brain_small, tomo_small and dkrz4d were visualized with volume rendering, temperature with 3D texture mapping. dkrz4d was also projected down to 3D dimensions for the visualization.*

module timing (detailed listing of time spent on specific modules) is only done for major modules contributing to a compression operation, therefore the time given for the entire operation may be a little higher than the sum of the major components' time. To keep the tables compact, the following symbols will be used to abbreviate the modules involved:

- $\mathcal{T}$ the entire tilecompression object;
- $\mathcal{S}$ the linstream object;
- $\mathcal{P}$ the predictor(s), using $\mathcal{P}_e$ for interchannel and $\mathcal{P}_a$ for intrachannel predictors if both were used;
- $\mathcal{C}$ the compression core class itself without its helper classes (i.e. the component managing compression streams, meta data, wavelet transformations etc.); this is essentially everything not covered by the other modules above.

throughput (amount of uncompressed data processed per second) is abbreviated by θ_c for compression and θ_d for decompression and has the unit kilobytes per second. This is important for transfer compression, because all compression techniques whose throughput is below the bandwidth of the transfer medium can't improve transfer times, no matter what rates they achieve.

4.2 Lossless Compression

The lossless compression types are RLE, ZLib, SepRLE, SepZLib and HaarWavelet (with either *RLE*, *ZLib* or *ArithCoder* compression streams). *ZLib* streams are always run at maximum compression level for optimum rates (although this increases compression time). We will first examine the compression rates and the time taken for the compression/decompression operations and then evaluate the effects of predictors on the compression of these test MDD. The separating compression streams will only be used on those MDD over structured based types, because for atomic types they are equivalent to the non-separating variants. Because lossless compression has few parameters and most of all no rate-distortion considerations to take into account, there is little point in presenting the measurements graphically, so tables will be used instead.

4.2.1 Relative Sizes and Timings

4.2.1.1 RLE

RLE		*lena*	*cnig*	*tomo_small*	*brain_small*	*movie_small*	*temperature*	*dkrz4d*
relsize		105.802	100.173	29.8034	45.1894	101.797	100.047	100.155
t_c	$\mathcal{T}$	38328	78479	117452	60240	105290	173128	172926
	$\mathcal{C}$	1453	4112	1876	2196	4466	22737	23060
	$\mathcal{S}$	36645	74108	115311	57790	100528	150054	149520
θ_c		6679	9786	10489	15884	8014	22180	22206
t_d	$\mathcal{T}$	16712	15414	28008	11972	34712	32896	30786
	$\mathcal{C}$	4	24	22	22	24	22	18
	$\mathcal{S}$	16503	15032	27708	11666	34401	32590	30492
θ_d		15318	49825	43987	79923	24307	116732	124732

Observations: only acceptable results for sparse data (*tomo_small*, *brain_small*), with an average *relsize* of 83.281. Very fast: all decompression times faster than even 100Mbit Ethernet. Because the algorithm exploits the base type size, its decompression speed scales with the base type size for atomic base types (*brain_small*: 2; *temperature*, *dkrz4d*: 4). Decompression takes noticeably less time than compression with $\frac{t_c}{t_d}$ between 2.29 and 5.61.

4.2.1.2 ZLib

ZLib	*lena*	*cnig*	*tomo_small*	*brain_small*	*movie_small*	*temperature*	*dkrz4d*
relsize	79.4243	77.8596	24.6251	37.0645	67.8804	62.0784	71.6091
t_c $\mathcal{T}$	182160	492185	382313	429512	514109	10850199	5788265
$\mathcal{C}$	1211	3621	1805	2061	3426	14459	16352
$\mathcal{S}$	180698	488297	380253	427213	510423	10835423	5771614
θ_c	1405	1560	3222	2228	1641	354	663
t_d $\mathcal{T}$	24361	72741	63105	58921	67030	343957	359046
$\mathcal{C}$	13	17	12	13	14	17	19
$\mathcal{S}$	24157	72476	62877	58712	66785	343662	358738
θ_d	10509	10558	19523	16239	12588	11164	10695

Observations: achieves compression for all test MDD, but only *substantial* compression for sparse data where *relsize* does not differ dramatically from that achieved by RLE. The average *relsize* is 60.077, well below that of RLE. For most test MDD, the throughput is above 10Mbit Ethernet bandwidth, with the exception of the floating point base types (*temperature*, *dkrz4d*) where it's well below that for compression. Decompression throughput is relatively stable between 10 and 20 MB/s for all test MDD, but compression is a lot slower with $\frac{t_c}{t_d}$ ranging from 6.03 to 31.55, where the peaks are at the floating point types.

4.2.1.3 Channel Separation

Channel separation compresses each channel separately rather than interleaved like the normal (simple) compression objects. Whether this improves compression rates depends very much on the data.

SepRLE	*cnig*	*movie_small*
relsize	104.324	92.9567
t_c $\mathcal{T}$	216619	212737
$\mathcal{C}$	94752	104516
$\mathcal{S}$	119773	106117
θ_c	3545	3966
t_d $\mathcal{T}$	147790	141016
$\mathcal{C}$	89080	100425
$\mathcal{S}$	57843	40026
θ_d	5197	5983

SepZLib	*cnig*	*movie_small*
relsize	68.4723	65.3181
t_c $\mathcal{T}$	664018	597487
$\mathcal{C}$	91877	101567
$\mathcal{S}$	569990	493716
θ_c	1157	1412
t_d $\mathcal{T}$	157732	168178
$\mathcal{C}$	89901	100315
$\mathcal{S}$	67066	67431
θ_d	4869	5017

Observations: compared to the non-separating compression objects, the core classes $\mathcal{C}$ gain considerable complexity because they have to separate the channels before passing the data to the compression streams (or the other way around during decompression), which also lowers throughput. The *relsize* for SepRLE gets 4% worse for *cnig* and 8% better for *movie_small* compared to RLE, so on average it compresses better than interleaved *RLE* compression. SepZLib is better than ZLib in both cases, by almost 10% in the case of *cnig* and by about 2% in the case of *movie_small*.

4.2.1.4 Haar Wavelet

Because all wavelet types can be combined with arbitrary linstream objects for compression, this section contains measurements for all compression streams. The abbreviation $\mathbf{Haar}_R$ stands for *RLE* compression, $\mathbf{Haar}_Z$ for *ZLib* compression and $\mathbf{Haar}_A$ for *ArithCoder*. It must be noted that Haar wavelets are no longer lossless for floating point types due to the accumulation of arithmetic errors, typically introducting an SNR around 10^{13}.

$\mathbf{Haar}_R$	*lena*	*cnig*	*tomo_small*	*brain_small*	*movie_small*	*temperature*	*dkrz4d*
relsize	103.302	101.721	35.9521	52.0418	96.9093	100.125	100.163
t_c $\mathcal{T}$	131021	552084	1019142	347435	750913	1116372	1196337
$\mathcal{C}$	98972	459538	858695	265314	603203	924738	989248
$\mathcal{S}$	29484	85032	136027	69981	123066	157548	164609
θ_c	1954	1391	1209	2754	1124	3440	3210
t_d $\mathcal{T}$	90031	451120	857713	252925	626536	912916	992121
$\mathcal{C}$	78550	419289	782121	218038	542159	848627	906681
$\mathcal{S}$	10016	27593	51540	24441	61317	51824	63092
θ_d	2843	1702	1436	3783	1347	4206	3870

Observations: where actual compression is achieved, using an *RLE* stream for the compression of Haar wavelet coefficients results in worse compression rates than using the *RLE* stream on the untransformed data, with the exception of *movie_small* where the *relsize* is about 5% better. Compression and decompression times are very stable with $\frac{t_c}{t_d}$ between 1.19 and 1.56, but since the average *relsize* of 84.316 is higher than that of using RLE, the use of *RLE* streams for Haar wavelets can be discouraged.

$\mathbf{Haar}_Z$	*lena*	*cnig*	*tomo_small*	*brain_small*	*movie_small*	*temperature*	*dkrz4d*
relsize	76.062	78.1662	30.1502	46.0699	69.7862	58.9853	66.7483
t_c $\mathcal{T}$	317317	1048883	1434041	748286	1427384	12762465	11864787
$\mathcal{C}$	98919	458846	861395	267198	599655	917731	992210
$\mathcal{S}$	216378	584126	548061	468499	804587	11818378	10835894
θ_c	807	732	859	1279	591	301	324
t_d $\mathcal{T}$	113071	514581	969016	347175	728947	1301545	1413213
$\mathcal{C}$	78450	417758	781381	219479	539023	842490	895037
$\mathcal{S}$	33209	92738	164265	117573	168452	447503	497513
θ_d	2264	1492	1271	2756	1157	2950	2717

Observations: in most cases, the compression is worse than that of using *ZLib* compression on the untransformed data (with an average *relsize* of 60.853 compared to 60.077 for ZLib), with the exceptions of *lena*, *temperature* and *dkrz4d*. Since the transformation is no longer lossless for the floating point MDD, that leaves only *lena* as a clear winner regarding compression rate. Compression and decompression times are more evenly balanced than they are for ZLib, with $\frac{t_c}{t_d}$ between 1.48 and 9.81 (again peaking at the floating point types), which has mostly to do with the constant overhead of doing the (inverse)

wavelet transformations; this means considerably reduced throughput, however. Note that the *ZLib* compression stream takes longer here than in ZLib, the most extreme case being *dkrz4d* where it takes almost twice as long.

Haar$_A$		*lena*	*cnig*	*tomo_small*	*brain_small*	*movie_small*	*temperature*	*dkrz4d*
relsize		73.2304	76.2756	33.9151	51.5994	72.7648	60.6027	69.2243
t_c	$\mathcal{T}$	593893	2030025	2455507	1952243	2290628	7893826	8788333
	$\mathcal{C}$	101348	460363	862209	272556	600599	921756	990919
	$\mathcal{S}$	490081	1562574	1567905	1666940	1665825	6946222	7760482
θ_c		431	378	502	490	368	486	437
t_d	$\mathcal{T}$	704750	2441859	2874431	2467355	2918908	9968959	11279958
	$\mathcal{C}$	80231	418841	790652	224761	762082	953413	1000509
	$\mathcal{S}$	622966	2017207	2054973	2230076	2133628	9000415	10253718
θ_d		363	315	429	388	289	385	340

Observations: with the exception of *lena* and *cnig*, *relsize* is worse than with the *ZLib* compression stream (with an average *relsize* of 62.516), so on average *ZLib* compression is the better choice. Compression and decompression times have an almost constant ratio between 0.78 and 0.85. This is the only compression stream where decompression takes longer than compression, the reason for which is that finding the symbol belonging to an interval in the decoder is more complex than mapping a symbol to an interval in the encoder. The throughput is below that of using the *ZLib* compression stream in most cases, except for the compression of the floating point MDD; on average, the *ZLib* stream compresses faster and more efficiently than any of the other streams.

4.2.2 Predictor Usage

Comparing all possible combinations of predictors and tilecompression objects in this work is not an option due to the large number of configurations, so only a small selection will be compared. The tilecompression object used will always be ZLib or SepZLib (for interchannel predictors), likewise only one instance of both predictor types will be used.

4.2.2.1 Intrachannel Predictors

The intrachannel predictor chosen is *hyperplane*, which replaces a cell's value with the difference to the value of its nearest neighbour in a user-defined direction. This means there are as many possible hyperplane predictors as there are dimensions in the MDD being compressed. The following two tables give the results for the best and the worst hyperplane directions; "norm" is the number of the dimension which is normal to the prediction hyperplane (i.e. the prediction direction).

Worst	*lena*	*cnig*	*tomo_small*	*brain_small*	*movie_small*	*temperature*	*dkrz4d*
relsize	65.8661	65.7497	23.6967	38.0968	61.563	58.0192	71.3821
norm	0	1	1	0	0	0	1
t_c $\mathcal{T}$	249686	888877	645351	562910	894878	8858437	6717292
$\mathcal{C}$	1706	7163	8137	7223	7641	35662	38758
$\mathcal{P}_a$	50294	184895	246332	96239	182280	218897	202393
$\mathcal{S}$	197323	696329	390396	458977	704461	8603282	6475006
θ_c	1025	864	1909	1700	943	433	572
t_d $\mathcal{T}$	71866	249840	308155	153683	246628	550626	562904
$\mathcal{C}$	29	119	33	33	111	44	45
$\mathcal{P}_a$	49712	184269	245311	95064	178596	221678	204957
$\mathcal{S}$	21809	65040	62399	58182	67524	328336	356834
θ_d	3562	3074	3998	6226	3421	6974	6822

Best	*lena*	*cnig*	*tomo_small*	*brain_small*	*movie_small*	*temperature*	*dkrz4d*
relsize	62.5507	64.6717	23.2253	37.6137	52.0185	51.3851	55.5522
norm	1	0	0	1	1	2	3
t_c $\mathcal{T}$	284350	882677	647546	574233	1019717	12731292	13096505
$\mathcal{C}$	1623	6928	8569	7428	7486	34037	35385
$\mathcal{P}_a$	54533	152508	249216	97434	173578	516216	521418
$\mathcal{S}$	227827	722762	389256	468887	838153	12180475	12539140
θ_c	900	870	1903	1666	827	302	293
t_d $\mathcal{T}$	75668	216169	309573	154315	231450	794156	828622
$\mathcal{C}$	28	118	33	33	114	40	42
$\mathcal{P}_a$	54087	152233	248049	96494	170851	506569	514601
$\mathcal{S}$	21247	63422	61072	57381	60061	287021	312957
θ_d	3383	3553	3980	6201	3645	4835	4634

Observations: with the exception of *brain_small*, the hyperplane predictor improved compression rates for all test MDD in both worst and best case. This is also stressed by the average *relsize* of 54.91 in the worst and 49.574 in the best case, which is noticeably below the *relsize* of ZLib without predictors. Most noteworthy are probably the results for the floating point fields *temperature* and *dkrz4d*, where the predictor allowed reducing the data size by a further 11–16%, which is even below the *relsize* of Haar wavelets with *ZLib* – but in contrast to those, the predictor coding was lossless for these MDD[1]. During compression, the predictor takes little time compared to the compression stream, and relative to the total compression time the predictor takes between 2% for worst case *temperature* and 38% for *brain_small* (usually around 20%, but for the floating point data it's only 2–4% in both cases). Typically, ZLib takes longer to process the predicted data, however, so the compression throughput is only around half that without predictors. For decompression, ZLib has a much more stable throughput, which is also considerably higher than

[1]The hyperplane predictor is usually lossless even for floating point types if the floating point data is smooth, because that means the deltas are small and mantissa overflows do not occur.

the compression throughput, so the main complexity is then at the predictor, which takes 36–80% of the total time.

Regarding the best normal dimension, there is no clear trend visible: the two raster images have different optimums, so do the tomograms. The movie compresses best not when predicting along the temporal dimension, as one would naturally assume, but along the horizontal axis. Only the floating point fields seem to follow a common rule in that they compress best when predicting along their widest dimension. All in all, the only viable approach regarding intrachannel predictors and optimum hyperplane directions is experimentation at the moment.

4.2.2.2 Interchannel Predictors

Since the only test MDD with structured base types use the RGB type, the natural choice for interchannel prediction is the *delta* predictor, for example using green to predict the other two channels[2]. The compression algorithms used will be SepZLib and HaarWavelet with *ZLib* compression.

SepZLib		*cnig*	*movie_small*
relsize		76.7796	59.3979
t_c	$\mathcal{T}$	609660	724730
	$\mathcal{C}$	98350	108445
	$\mathcal{P}_e$	11326	12308
	$\mathcal{S}$	497463	601424
θ_c		1260	1164
t_d	$\mathcal{T}$	171659	178431
	$\mathcal{C}$	89836	100323
	$\mathcal{P}_e$	9508	10945
	$\mathcal{S}$	71425	66545
θ_d		4474	4729

Haar$_Z$		*cnig*	*movie_small*
relsize		79.2013	64.9763
t_c	$\mathcal{T}$	1055188	1860660
	$\mathcal{C}$	466417	608099
	$\mathcal{P}_e$	11248	12905
	$\mathcal{S}$	570704	1216940
θ_c		728	453
t_d	$\mathcal{T}$	522380	722712
	$\mathcal{C}$	418504	536276
	$\mathcal{P}_e$	11628	10223
	$\mathcal{S}$	88090	152778
θ_d		1470	1167

Observations: in both cases, *relsize* went down for *cnig* and up for *movie_small*, which implies that there is little channel correlation in *cnig* (there are indeed large areas of pure green) and substantial channel correlations in *movie_small*. The average *relsize* for SepZLib went up from 66.895 without to 69.089 with interchannel predictors, whereas for lossless Haar wavelets it went down from 73.976 without to 72.089 with interchannel predictors. The best results with the intrachannel hyperplane predictor for these two test MDD were achieved by SepZLib, however: 64.1214 (norm = 0) for *cnig* and 45.8479 (norm = 1) with

[2]While the predicting channel has little relevance in lossless compression, it can be very important in lossy mode where some channels can be encoded with less precision than others. The human eye is best at discerning shades of green, followed by red and least of all blue, so green should be chosen as the predicting channel and encoded with the highest precision, whereas red and blue (deltas or actual colours) can be encoded with less precision without affecting perceived image quality [40]. The advantage of encoding the deltas rather than the original channels with less precision is a better preservation of grey levels which are of particular importance.

the additional interchannel delta predictor for *movie_small* (for the full size *movie_full* cube, *relsize* is even better at 38.0751 with this configuration).

4.2.3 Conclusions for Lossless Compression

Although lossless compression has relatively few parameters, the addition of predictors and configurable compression streams results in a huge number of possible combinations, only a small, characteristic part of which was tested in this section. Looking at the results, the following general guidelines can be given for the compression and transformation techniques tested:

RLE: is obviously only viable for sparse data, as there is none or little compression achieved for other data types. RLE is very fast for all data types, and for sparse data its *relsize* is very close to the best achieved by the other compression techniques (e.g. for *tomo_small relsize* is only 5% worse than that of ZLib, but compression is more than three times faster).

ZLib: emphasized its status as *the* standard in lossless compression, having the best average *relsize* and good speed in most cases. Floating point data results in greatly reduced throughput when compressing, however.

Haar Wavelets: have rather mediocre results for the lossless case. The best average *relsize* is slightly worse than that of ZLib without predictors and considerably worse than ZLib with predictors, while taking noticeably more time. This is not all that surprising when looking at figure 3.8 on page 45, where the detail bands were enhanced, showing a lot of random noise: a lossy algorithm could ignore the noise and thereby achieve much better compression rates, but a lossless one has to preserve the noise, which greatly compromises its rate. Lossless Haar wavelets work well for some synthetic data (e.g. a checkerboard compresses to half the size of ZLib with Haar wavelets), but especially regarding the effects of intrachannel predictors on regular ZLib, there can be no recommendation for lossless Haar wavelets for the kind of data found in the test set.

Channel Separation: can often improve compression rates, especially in combination with *ZLib* compression, so this is always worth trying for structured base types despite the little overhead, although there can be no guarantee that the rate doesn't deteriorate.

Predictors: intrachannel predictors worked very well on most test MDD, even floating point fields (16% better for *dkrz4d* with ZLib, which corresponds to savings of 614kB compared to compressing the 4MB object without predictors). Considering the low overhead (only about 30–50% the time needed for the Haar transformations), it is highly recommended to try improving the compression rate with intrachannel prediction; only experimentation can determine the best correlation direction, however,

which varied even for data of the same dimensionality and axis interpretation. Interchannel predictors improved the rates only in half the test cases and not by as much as intrachannel prediction; but since they improve compression rates for at least one test MDD, their presence in the engine is justified.

When looking at the timings for interchannel and intrachannel predictors, it is obvious that intrachannel predictors take longer. The reason for that is that intrachannel predictors must be able to iterate through the data in non-default order, because the values used for prediction are read from the same channel the deltas are written to[3]; the time difference between interchannel and intrachannel predictors reflects the time difference between iteration in default and arbitrary order.

This leads to the following general guidelines for lossless compression, based on the test data:

1. if compression speed is highly critical: if the data is sparse, use RLE, otherwise turn off compression entirely;

2. if speed is less mandatory, use *ZLib* variants. For structured base types compare the compression rates achieved by ZLib and SepZLib;

3. try intrachannel predictors for any base type, in particular hyperplane prediction. Find the best normal dimension by comparing compression rates with all possible normals;

4. for structured base types, try interchannel prediction.

4.3 Lossy Wavelet Compression

Lossy compression in this engine means wavelet compression, so this section is dedicated to the quantizing wavelets described in section 3.3.4.2, whose filter coefficients are listed in appendix B. A lossy wavelet compression object consists of three major modules:

1. the core class, which is a direct descendant of qwaveletcomp in figure 3.4 and performs the basic tasks of channel separation, predictor and meta data management as well as the actual (inverse) wavelet transformation. There are currently 22 different lossy orthogonal wavelet classes available, falling into four groups (Haar, Daubechies, Least Asymmetric and Coiflet) with filter lengths between 2 and 30 taps;

2. the quantization engine, which converts the wavelet coefficient array into a quantized representation and back. There are currently three approaches possible: homogeneous band quantization (section 3.4.2) and Generalized Zerotree (section 3.4.3)

[3]This applies not only to hyperplane prediction but even more so to the averaging neighbours approaches (see the prediction context in figure 3.19 on page 95).

with one-pass or two-pass coding (section 3.4.3.7). All three variants require a quality parameter, which is the number of bits to use per coefficient in homogeneous band quantization, whereas for the zerotree coders it can be either minimum SNR, minimum PSNR or maximum residual per cell (section 3.4.3.8);

3. the compression stream used to actually (de)compress the quantized representation of the coefficient array. The default streams used will be *ZLib* for homogeneous band quantization and *ArithCoder* for zerotree coding. *ArithCoder* only performs well for small alphabets and uncorrelated symbol sequences (no patterns) and is therefore unsuitable for homogeneous band quantization; measurements made in this respect are not included in this work, however.

Listing results for all possible configurations in this thesis would obviously increase its volume out of proportion, so a representative subset must be found, which is necessarily tiny compared to all possibilities. Fortunately, most of the results follow very similar rules, so this is possible without over-generalization. There may be MDD other than the ones tested here where these rules no longer apply, however.

4.3.1 Relative Sizes and Timings

The wavelets used in this section are normally lossy, but the amount of loss can be limited via a quality parameter of the wavelet quantization module[4]. It is important to understand that the quality parameter concerns the quantization of the wavelet coefficients, not the reconstructed data; errors can accumulate in the inverse wavelet transformation, as explained in section 3.4.1, so SNR or maximum residual for wavelet coefficients can differ considerably from SNR and maximum residual of the reconstructed data. In order to avoid confusion, I will use SNR_w and RES_w when they apply to the wavelet coefficients (i.e. the quality parameters) and SNR and RES for the reconstructed data.

The measurements were done in such a way that starting from a given minimum quality, the test data was compressed with this quality, logging the current values of the quality parameter, the resulting (P)SNR, *relsize* and maximum difference values for the reconstructed data as well as timing information for all major modules. The quality parameter was then increased, the test data compressed again and so forth until a maximum quality was reached or the data could be reconstructed without loss (when timing the engine, measurements were always run up to a maximum quality parameter)[5]. The resulting data was then represented graphically in two kinds of diagrams, one for timing purposes and

[4]Provided this quality parameter is high enough, many MDD can be compressed without loss even with this part of the compression engine, although in this case compression rates can be far worse than when using a dedicated lossless compression technique.

[5]The maximum SNR_w was 2^{29} for timing measurements and 2^{50} for size-snr analysis; for homogeneous band quantization, the maximum number of bits used was 32 in both cases. SNR coding started with an SNR_w of 2, which was doubled after each iteration until the maximum SNR_w was reached; for homogeneous band quantization, coding started with 2 bits per coefficient and that value was incremented until the maximum number of bits was reached.

one for rate-distortion analysis. Axes representing (P)SNR or residual values are always plotted logarithmically.

For timing graphs, the SNR_w is plotted on the abscissa and the time taken for each major module on the ordinate; these graphs will be called *quality-time* graphs. When comparing graphs for different configurations, the one whose curve is lowest is faster, i.e. better as far as runtime considerations are concerned.

The efficiency of a lossy compression algorithm is determined by its *relsize* and its distortion, both of which should be as small as possible. For the graphs in this section, the *relsize* is plotted on the abscissa and the SNR of the reconstructed data on the ordinate[6]; this kind of graph will be called a *size-snr* graph. Note that since for lossless reconstruction the SNR would be ∞, this case can't be plotted. Another type of graph plots *relsize* vs. maximum residual of the reconstructed data (*size-residual*). When comparing these graphs for two algorithms a and b, a is better than b if

1. a's *relsize* for lossless reconstruction is smaller than b's
2. a's *size-snr* graph is above that of b
3. a's *size-residual* graph is below that of b

Since these conditions are rarely all true at the same time, it depends on the user's requirements which algorithm is better than another. If the user wants lossless reconstruction, then condition 1 is the only one of importance; otherwise it depends on the quality the user wants (some wavelets work better for low rates, some better for high ones) and whether the average distortion should be minimal (condition 2) or the maximum residual per cell (condition 3).

In this section, *size-snr* and *quality-time* comparison graphs will be given for all test MDD using one-pass zerotree coding with SNR termination; zerotree coding was chosen because it provides the best rates (see section 4.3.2) and SNR termination because it best represents the average distortion. The comparisons will be made across a selection of wavelet filters from each group and all filter lengths. Additional diagrams will be given where appropriate to stress extraordinary properties. Because the time taken depends on the quality parameter, direct comparison with the lossless results is not possible; to allow some reference nonetheless, time consumed and throughput will be given for the Daubechies 4-tap wavelet with $\mathrm{SNR}_w = 4096$.

4.3.1.1 lena

Figure 4.2 shows the *quality-time* graphs for compression and decompression of *lena* with the Daubechies 4-tap wavelet. The diagrams visualize the time consumed by each of the major modules involved: the core class (daub4), the arithmetic coder (*ArithComp*, *ArithDecomp*), building the zerotree structure (*ZBuild*), the zerotree coder (*ZEncode*, *ZDecode*)

[6]The SNR is basically the inverse of the distortion, so it should be maximal to achieve minimal distortion.

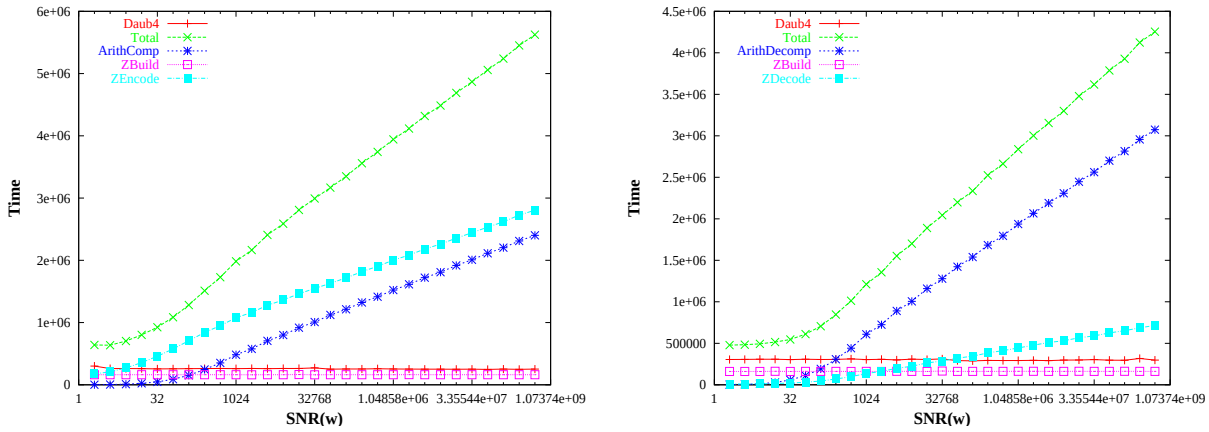


Figure 4.2: Lena detailed timings for Daubechies 4-tap wavelets

and the resulting total time. As expected, the timings for the core class and the building of the tree structure are constant and only zerotree coder and arithmetic coder vary with the SNR_w. As can be seen, *ZEncode* takes most of the time during compression, whereas *ZDecode* takes very little time during decompression; the reason for that lies in the aggregation passes during encoding described in section 3.4.3.6, which are not necessary during decoding (*ZEncode* would take even longer if the aggregated values weren't available). The arithmetic coder is actually faster compressing than decompressing, which was already observed in section 4.2.1.4. We can also see that for higher quality ($\mathrm{SNR}_w \geq 1024$ in this case), the graph approximates a perfectly straight line, so the time is proportional to the logarithm of the SNR_w. This is true for all test MDD and all filters once SNR_w has reached a certain value, although Haar wavelets have a rather jagged curve. The total times for compression and decompression are comparable, with compression taking about 25% longer than decompression for higher quality. The total time for the Daubechies 4-tap wavelet with $\mathrm{SNR}_w = 4096$ is 2407487μs for compression and 1552698μs for decompression, leading to throughput values $\theta_c = 106$ and $\theta_d = 165$.

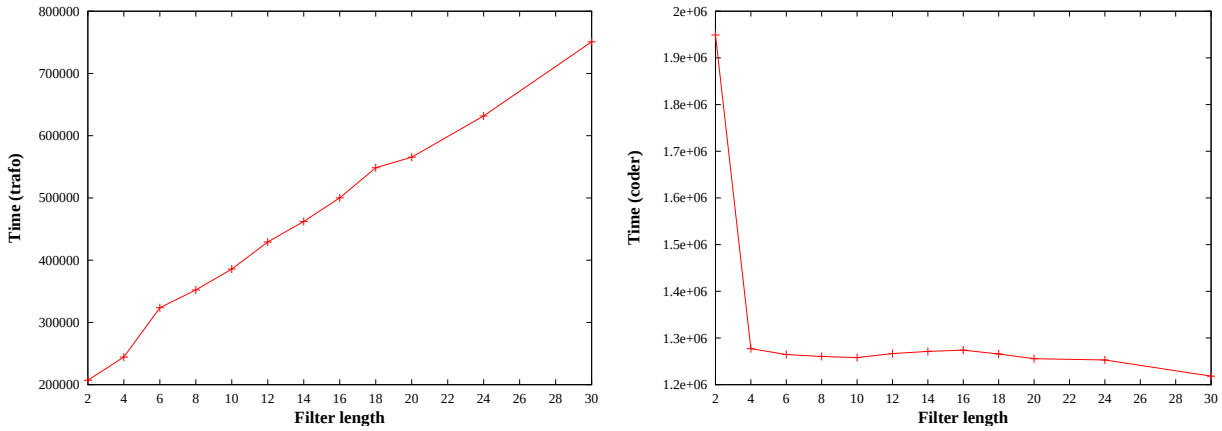


Figure 4.3: Lena timing comparisons for wavelet transformation and zerotree coder

Figure 4.3 compares the time taken by the wavelet core class (left) and the zerotree encoder (right) for wavelet filters with length 2–30. The timings for the core class are almost on a perfect line, with only one major exception at filter length 6, the reason for which is that from this filter length on the generic orthowavelet class is used, whereas there are specialized (and therefore faster) classes for filter lengths 2 and 4. The timings for the zerotree coder show that the coder requires approximately 50% more time for the Haar wavelet (filter length 2) than for any of the other wavelets. The reason for that is that in this case the significant coefficients in the tree are more evenly spaced out than for the other filters where they are concentrated around small values, so the aggregation must be performed more often: even one significant coefficient in the currently encoded subtree (starting in a node on the coarsest scale level) requires running the aggregation code for the entire subtree before the next dominant pass in the current implementation. If many coefficients have similar values, many of them will be encoded in the same dominant pass, so the time taken for the aggregation is small compared to the encoding time. If the values cover a wide range, on the other hand, only few coefficients will be encoded during each pass and the aggregation time becomes dominant, as is apparently the case for Haar wavelets. Because each node in a D-dimensional zerotree has 2^D children, this effect becomes more pronounced in higher dimensional spaces, as can be seen in figures 4.9 and 4.15. Note that this is an effect intrinsic to zerotree quantization and doesn't happen in homogeneous band quantization. Also note that this doesn't mean that zerotree coding has exponential complexity, because even if all subtrees have to be aggregated after one pass, the number of nodes visited for the aggregation equals the number of cells; the exponential complexity only occurs locally in a subtree for the extreme case when only one coefficient in the subtree is encoded during a dominant pass. Improving the zerotree aggregation algorithm to minimize the aggregation path is therefore an attractive optimization for the future work section. When decoding, on the other hand, the total time consumed when using Haar wavelets is typically at the lower end of the scale.

Figure 4.4 shows the *quality-time* comparison graph for 11 representative wavelet classes. The curves are very close to each other, so using longer filters does not affect performance dramatically because the compression time is dominated by zerotree coder and arithmetic coder (compare with figure 4.2). The general trend is that shorter filters take less time, the only exception being the Haar wavelet, for reasons explained above and shown in figure 4.3 to the right.

Figure 4.5 shows the *size-snr* comparison graph for the same 11 wavelet classes. The differences between the wavelets are even smaller than the timing differences for this MDD, the winners by a very close margin being Daubechies 6-tap and Least Asymmetric 8-tap wavelets, whereas the loser (by an equally close margin) is the Haar wavelet.

4.3.1.2 cnig

Figure 4.6 shows timing and *size-snr* comparison graphs for a selection of wavelets for the *cnig* MDD. The graphs are very similar to the ones for *lena* in figures 4.4 and 4.5, therefore they are printed as smaller versions. The only notable difference to *lena* is that the Haar

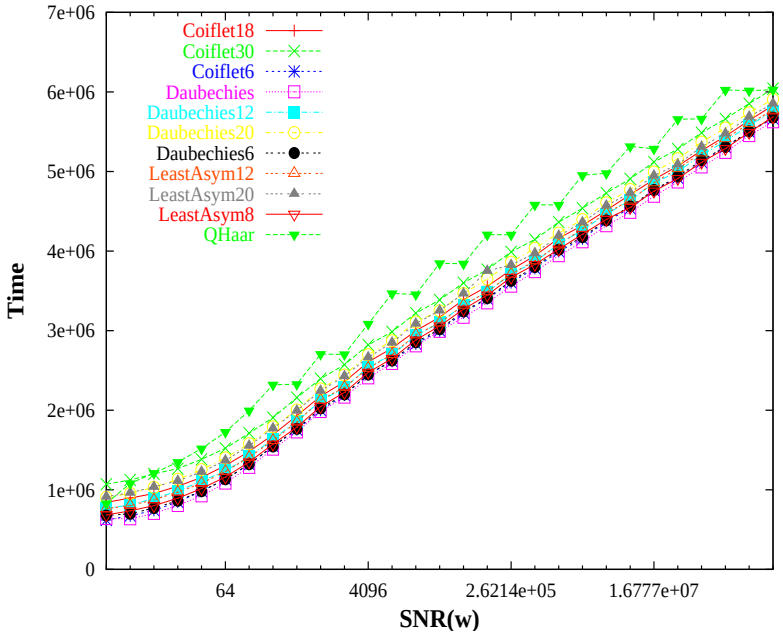


Figure 4.4: Lena timing comparisons

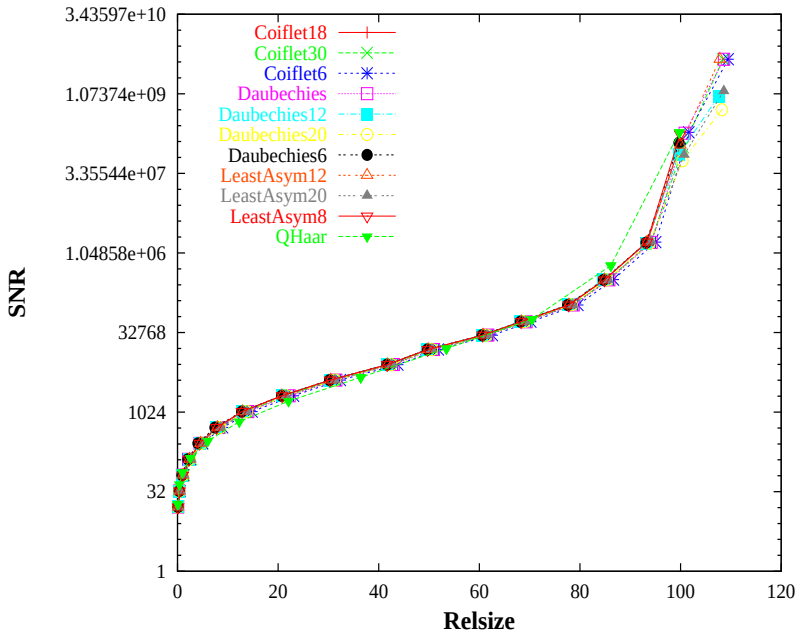


Figure 4.5: Lena size-snr comparisons

wavelet beats most of the other wavelets for *relsize* higher than 50. The total time for the Daubechies 4-tap wavelet with $\text{SNR}_w = 4096$ is $6955268\mu s$ for compression and $4362677\mu s$ for decompression, leading to throughput values $\theta_c = 110$ and $\theta_d = 176$, which are also very similar to *lena*. Another property in common with *lena* is that the *relsize* for lossless

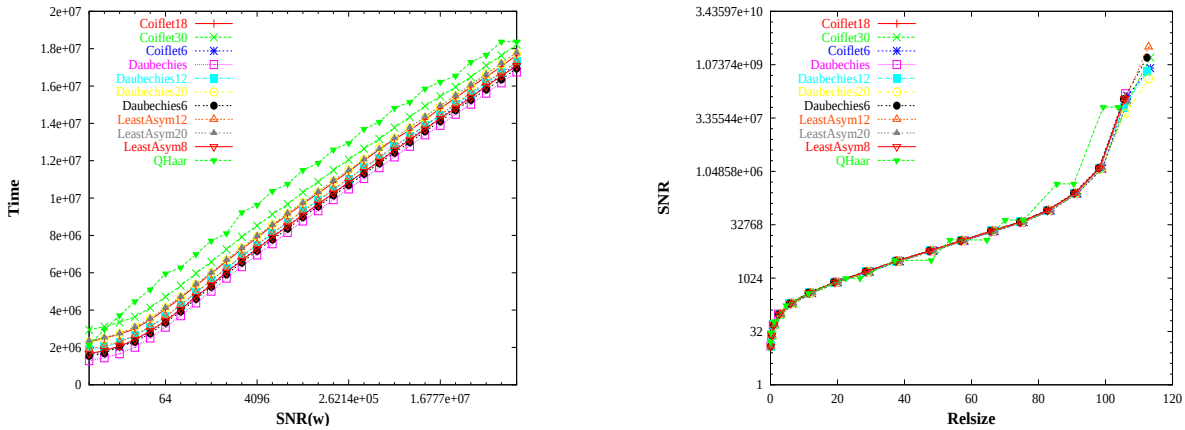


Figure 4.6: CNIG timing and size-snr comparisons

reconstruction is larger than 100 and considerably above that achievable with a dedicated lossless compression algorithm like in section 4.2.2.1.

4.3.1.3 tomo_small

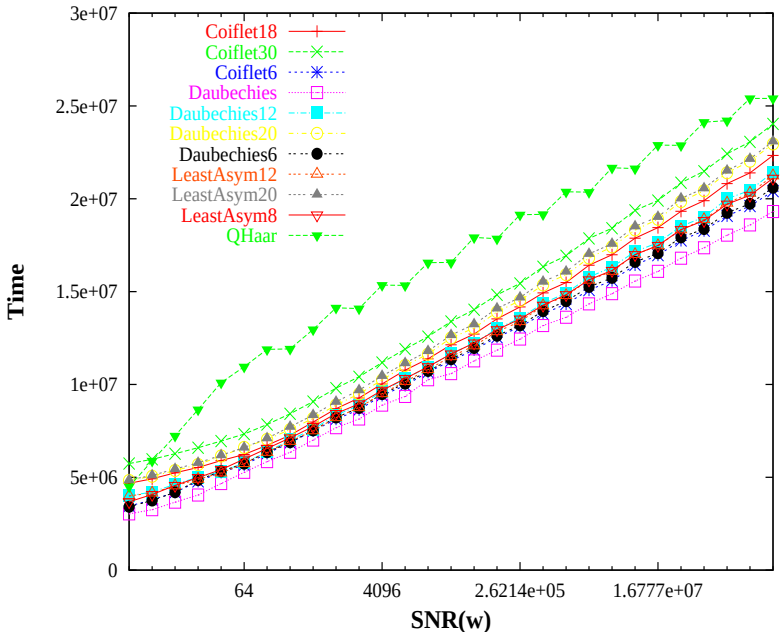


Figure 4.7: Tomogram timing comparisons

Figure 4.7 shows timing comparisons for the compression of *tomo_small*. As observed previously in section 4.3.1.1, the total time taken for encoding with Haar wavelets is higher than with the other filters, and since the tomogram is 3D rather than 2D, the effect is more pronounced. Apart from Haar wavelets, the time taken depends on the filter length, so

Daubechies 4-tap filters are the fastest. The total time for the Daubechies 4-tap wavelet with $\text{SNR}_w = 4096$ is 8884309μs for compression and 4387475μs for decompression, leading to throughput values $\theta_c = 139$ and $\theta_d = 281$, where the θ_d is considerably higher than in the 2D cases, which is most likely caused by the sparsity of the data (see also θ_d for *movie_small*, which is also 3D, but not sparse).

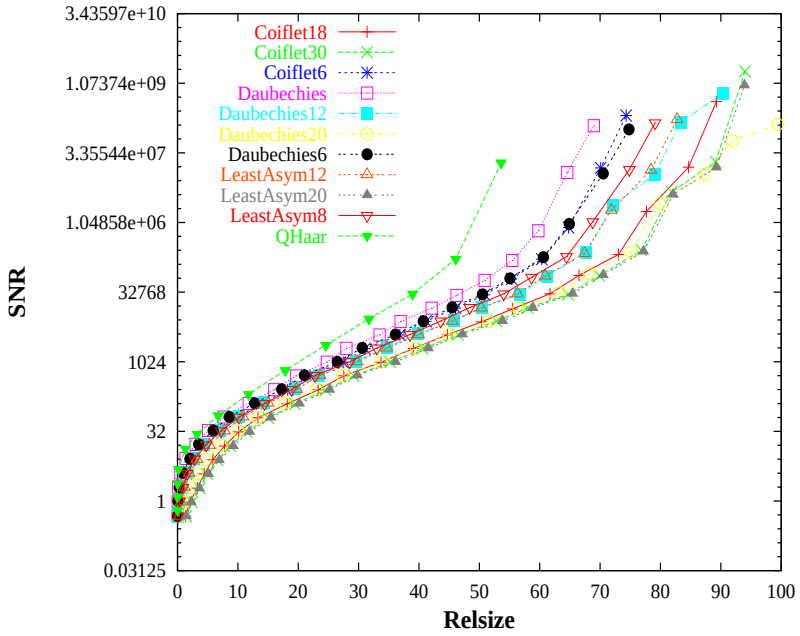


Figure 4.8: Tomogram size-snr comparisons

Figure 4.8 shows the *size-snr* comparison graphs for the compression of *tomo_small*. This tomogram is rather sparse and contains a certain amount of noise, which poses a problem for longer (and therefore smoother) wavelet filters. It therefore comes as no surprise that the shortest filter (Haar) performs best, its curve being above all others for all sizes and allowing lossless reconstruction with a *relsize* of 60.0927; the other wavelets perform worse in direct proportion to their length for the most part (only for very high quality does the longest filter (Coiflet 30-tap) perform slightly better than the second longest filters with 20 taps. Clearly, Haar wavelets are the best choice for sparse and noisy data. If lossless reconstruction is required, an actual lossless compression algorithm should be chosen, however, since even the best lossless *relsize* achievable with wavelets is about 2.5 times higher than that of ZLib and even twice as large as that of RLE (see sections 4.2.1.1 and 4.2.1.2).

4.3.1.4 brain_small

Figure 4.9 shows the timing comparisons for the compression of *brain_small*. All in all, the compression times are very similar for all wavelets with the exception of Haar, which takes about twice as long as most of the other wavelets (for reasons already explained in

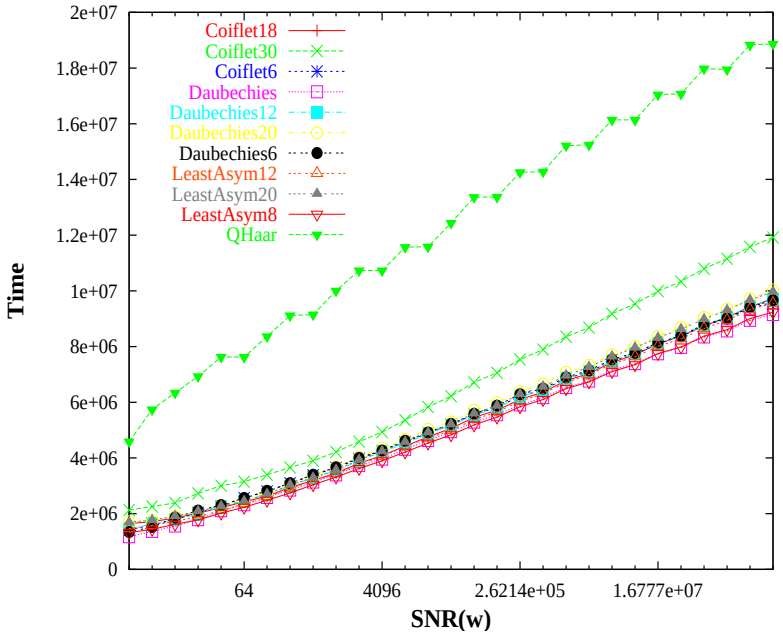


Figure 4.9: Brain timing comparisons

section 4.3.1.1) and the Coiflet 30-tap wavelet due to its length. The total time for the Daubechies 4-tap wavelet with $\text{SNR}_w = 4096$ is 3960733μs for compression and 2088218μs for decompression, leading to throughput values $\theta_c = 242$ and $\theta_d = 458$, which are approximately twice as large as they were for *tomo_small*. This is fairly straightforward to understand because the data is converted to a floating point array for wavelet transformation and quantization, the size and complexity of which only depends on the number of cells, not the original base type. Since each cell[7] of *brain_small* has twice the size of a cell in *tomo_small*, but the complexity of encoding a cell is about the same, the throughput will be approximately twice as large as that of *tomo_small*.

Figure 4.10 shows the *size-snr* comparison graph for *brain_small*. Similar to the one for *tomo_small*, the Haar wavelet universally performs best, allowing lossless reconstruction at a *relsize* of 81.7542, the other filters performing worse in direct proportion to their length. But as with *tomo_small*, lossless reconstruction can be achieved far more efficiently with a dedicated lossless algorithm by a factor of more than 2. Depending on which levels of distortion are acceptable, far better results can be achieved by lossy wavelets, however, which will be discussed in more detail in section 4.3.6.

4.3.1.5 movie_small

Figure 4.11 shows the timing comparisons for the compression of *movie_small*. Compared to *tomo_small*, the speed penality for the Haar wavelet is even more severe, taking almost

[7]Or rather more precisely: each atomic type within each cell, because wavelet techniques always do channel separation.

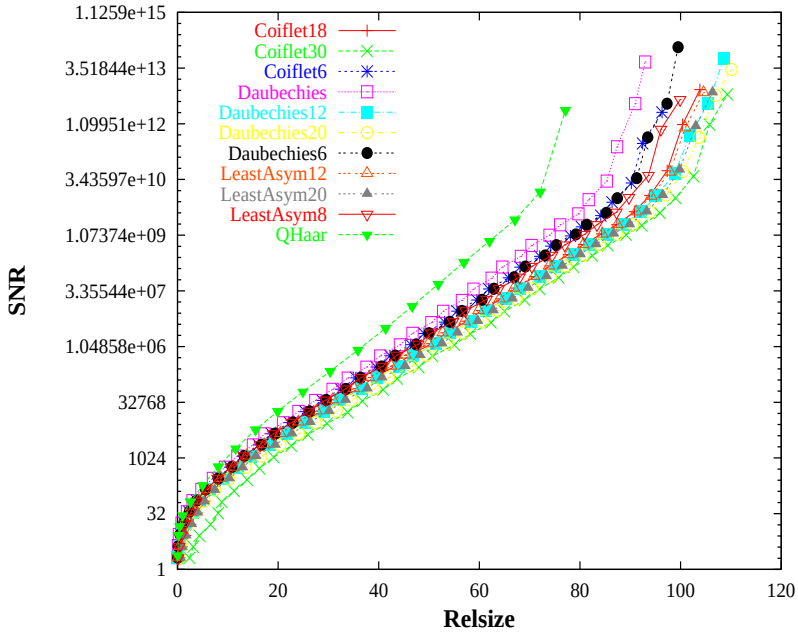


Figure 4.10: Brain size-snr comparisons

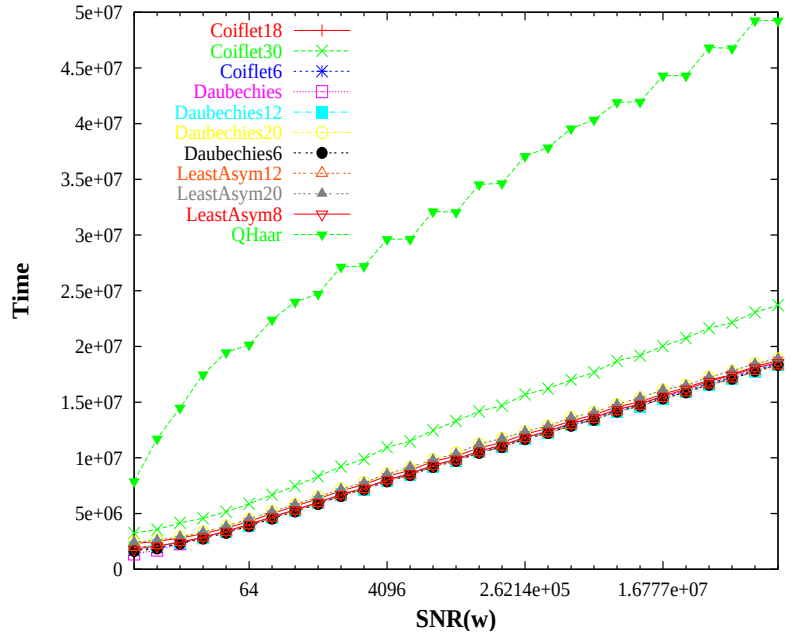


Figure 4.11: Movie timing comparisons

three times as long as most other wavelets, which indicates very evenly distributed wavelet coefficient values. Also the longest filter (Coiflet 30-tap) takes noticeably longer than the others, which have almost identical timing values. The total time for the Daubechies 4-tap wavelet with $\mathrm{SNR}_w = 4096$ is $8036931\mu s$ for compression and $5030343\mu s$ for decompres-

sion, leading to throughput values $\theta_c = 105$ and $\theta_d = 168$. As expected, the compression throughput is very close to that of *tomo_small* again, whereas the decompression throughput is considerably lower and closer to *lena* and *cnig* – a consequence of *tomo_small* being sparse, in contrast to *movie_small*.

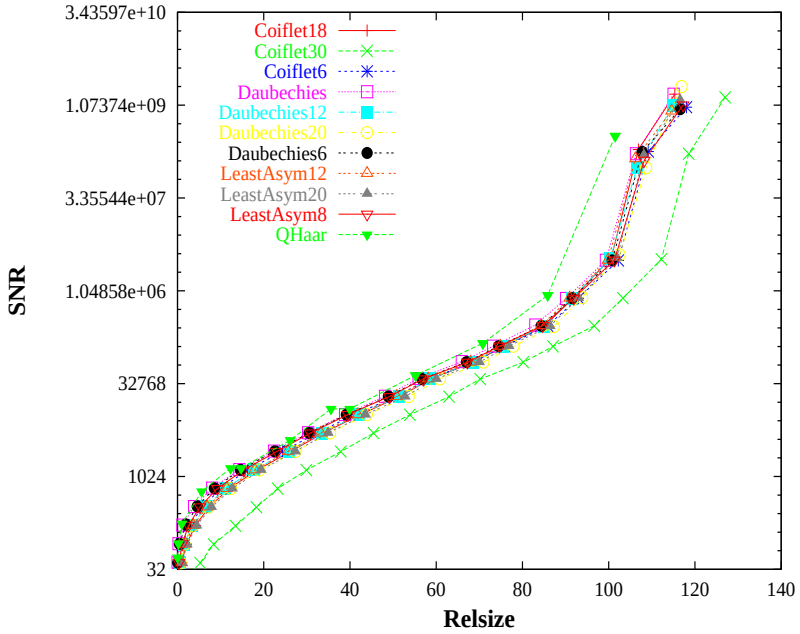


Figure 4.12: Movie size-snr comparisons

Figure 4.12 shows the *size-snr* comparison graph for the compression of *movie_small*. In contrast to *tomo_small*, most wavelets perform equally well, once again with the exception of the Haar wavelet which performs best in particular for *relsize* above 70, and the Coiflet 30-tap wavelet which has once again the worst performance. The reason for the particularily inferior performance of Coiflet30 for this MDD lies most likely in its spatial extent $60 \times 80 \times 60$, which becomes $15 \times 20 \times 15$ after two coarsening steps, which still allows applying wavelet filters with a maximum length of 20 taps along one dimension; since this is not true for Coiflet30, the multiresolution analysis for this filter has to terminate one level earlier than for all the others and therefore can't concentrate the energy as well on the coarse scale levels as the other wavelets. The best *relsize* for lossless reconstruction can once again be obtained with the Haar wavelet at 114.512; however this is yet again more than twice the *relsize* of using a dedicated lossless compression algorithm with predictors.

4.3.1.6 temperature

Figure 4.13 shows the timing comparisons for the compression of *temperature*. The time required for encoding data transformed with the Haar wavelet relative to the time for the other wavelets is smaller for this MDD than it was for *movie_small* and is only about 50% higher on average. Apart from this exception, the timing values for the other wavelets once

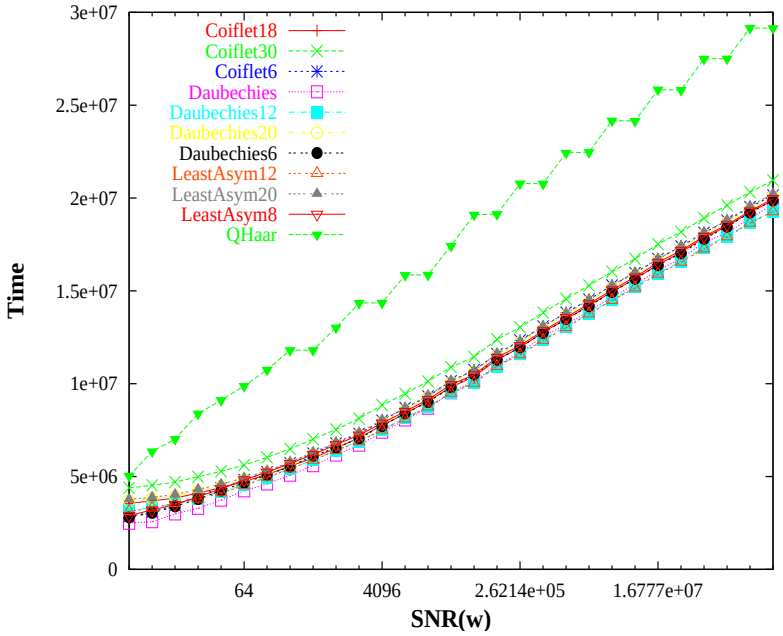


Figure 4.13: Temperature timing comparisons

again scale proportionally with the filter length. The total time for the Daubechies 4-tap wavelet with $\text{SNR}_w = 4096$ is 7388013μs for compression and 3695840μs for decompression, leading to throughput values $\theta_c = 520$ and $\theta_d = 1039$, which are again approximately twice as large as those of *brain_small*, in accordance with another doubling of the base type size. The compression throughput is thus even slightly higher than that of ZLib (see section 4.2.1.2), although the decompression throughput is by an order of magnitude below that of ZLib.

Figure 4.14 shows the *size-snr* comparison graph for the compression of *temperature*. Most of the time, all wavelets perform simularily well, with clearly visible exceptions only for high quality. Closer inspection reveals that with the exception of the Coiflet 6-tap wavelet the Haar wavelet is beat by a small margin by filters with lengths ≤ 12 for *relsize* up to around 60. The Coiflet6 wavelet has constant SNR $\approx 10^{12}$ for *relsize* higher than 50, which can only be caused by the accumulation of numerical errors during zerotree coding for very high SNR_w values. This effect can also be observed in *dkrz4d*, whose origin is the same laboratory as this MDD's, so the data from this laboratory obviously has some properties which cause convergence problems with the Coiflet6 wavelet. The best *relsize* for lossless reconstruction is once again achieved by the Haar wavelet at 71.1636, however. This is still higher than the best *relsize* achieved by dedicated lossless compression algorithms, but not by as much as in previous cases. The only other wavelet to achieve lossless reconstruction within the maximum SNR_w of 2^{50} was the Daubechies 20-tap wavelet, but with a considerably higher *relsize* of 122.618.

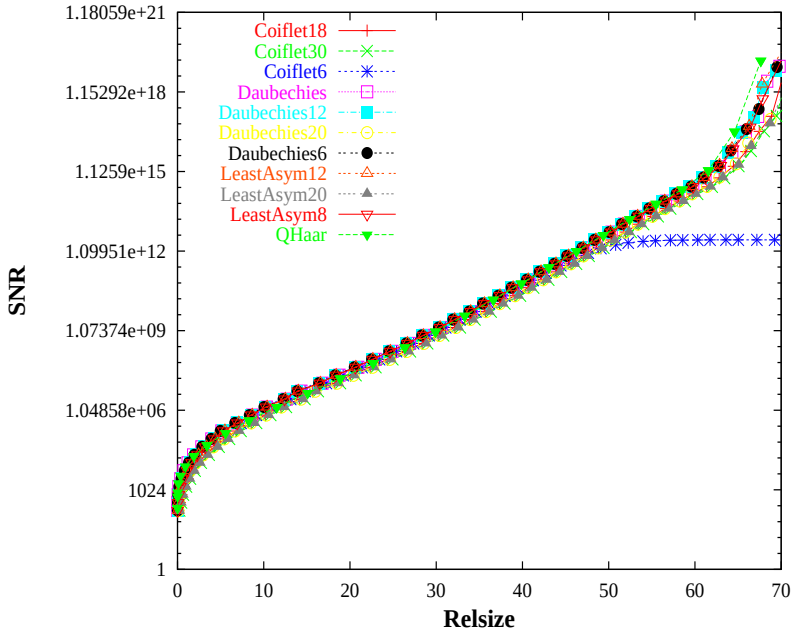


Figure 4.14: Temperature size-snr comparisons

4.3.1.7 dkrz4d

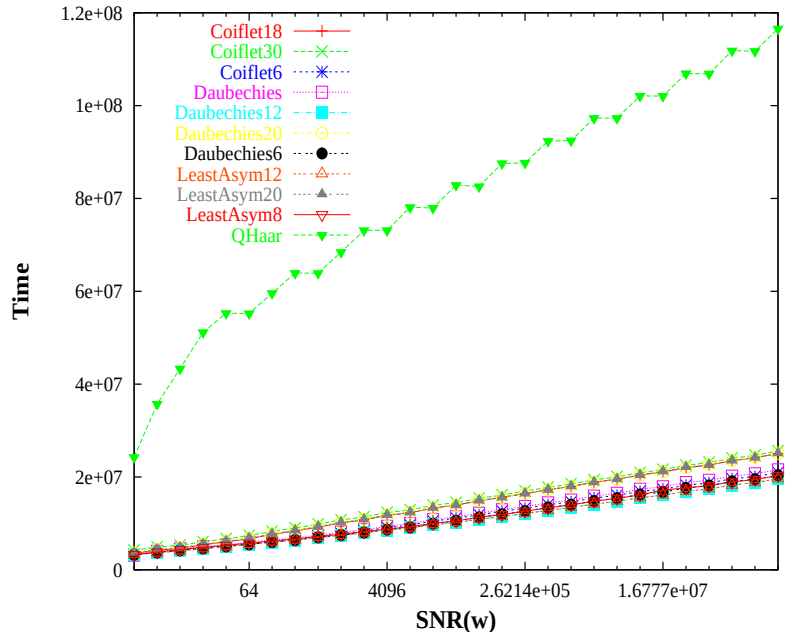


Figure 4.15: DKRZ timing comparisons

Figure 4.15 shows the timing comparisons for the compression of *dkrz4d*. The speed penalty for the Haar wavelet is the largest so far, because of the exponential complexity

of aggregating the node values described in section 4.3.1.1, taking more than five times as long as the other wavelets on average. Another (relatively small) timing difference exists between filters longer and shorter than 15 taps. The wavelet transformation as such actually takes less time for the longer filters than it does for the shorter ones, because they can't be applied along the dimension with width 15 on the finest level; this causes a similar absence of energy concentration as in the case of Haar wavelets and consequently longer overall compression time. The total time for the Daubechies 4-tap wavelet with $\mathrm{SNR}_w = 4096$ is $9366951\mu s$ for compression and $4648919\mu s$ for decompression, leading to throughput values $\theta_c = 410$ and $\theta_d = 826$.

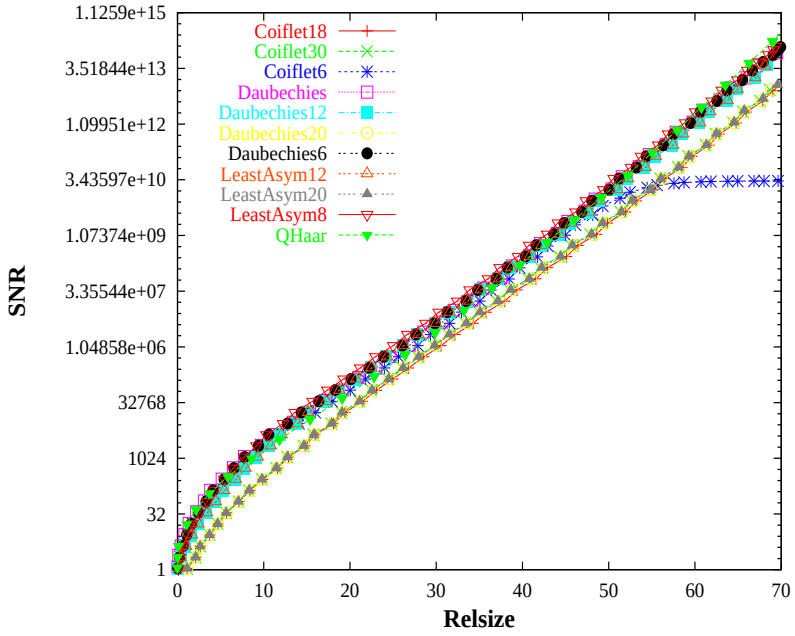


Figure 4.16: DKRZ size-snr comparisons

Figure 4.16 shows the *size-snr* comparison graph for the compression of *dkrz4d*. Same as the timing graph in figure 4.15, there are two major branches, the lower one with filters longer than 15 taps and the upper one with filters shorter than that. The longer filters perform noticeably worse, which is not suprising for an MDD that has an average length of 30 cells along each dimension. Haar wavelets perform worse than most filters with a maximum length of 12 taps for *relsize* up to 40 and worse than filters with a maximum length of 8 taps for *relsize* up to 60; also the same SNR convergence problem of the Coiflet 6-tap wavelet can be observed here that happened with the *temperature* MDD, albeit at a lower SNR of approximately $3 \cdot 10^{10}$. No wavelet achieves lossless reconstruction within the maximum SNR_w of 2^{50}, the closest being the 8-tap Least Asymmetric wavelet with a maximum residual per cell of $2.4 \cdot 10^{-5}$ at a *relsize* of 65.3511 – which is better than the *relsize* achieved by the lossless ZLib without predictors.

4.3.2 Quantization Comparisons

As described in section 3.4 and towards the beginning of section 4.3, there are three wavelet quantization techniques: homogeneous band quantization and zerotree coding with one or two passes. Measurements so far were taken with one-pass zerotree coding only; the purpose of this section is to compare this quantization with the other two.

Homogeneous band quantization, described in section 3.4.2, is the oldest and simplest technique: the bands are partitioned into equivalence groups by a band iterator and within each group all coefficients are quantized with a constant number of bits. The partitioning chosen here is the default *leveldet*, where all bands in the same scale level and with the same degree of detail are within the same band equivalence group (see section 3.4.2.1). The bit allocation chosen for this test was an identical number of bits for all band equivalence groups; this is not optimal, as for instance all-detail bands can often be quantized to 0 bits without noticeably affecting distortion levels (while reducing the rate), but choosing a constant number of bits reduced the number of parameters to one and thus allowed a straightforward comparison with SNR zerotree coding.

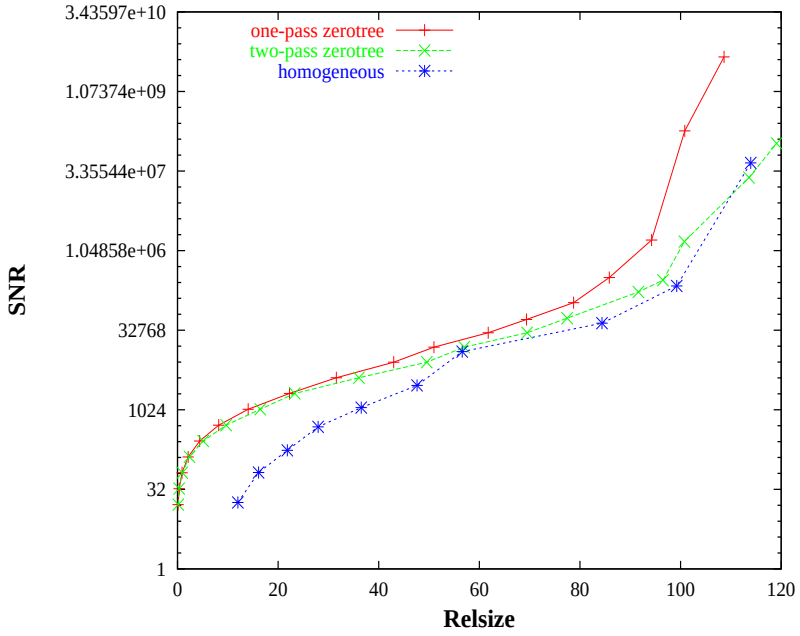


Figure 4.17: Lena size-snr quantization comparisons

Figure 4.17 shows *size-snr* comparisons between one-pass and two-pass zerotree coding and homogeneous band quantization of the *lena* MDD transformed with the Daubechies 4-tap wavelet. As can be clearly seen, the two zerotree variants have very similar size-snr curves for *relsize* up to 20. For *relsize* up to 80, the two curves are still very close, but one-pass coding has an obvious edge over two-pass coding. Beyond a *relsize* of 80, one-pass coding performs noticeably better than two-pass coding for this MDD, allowing lossless reconstruction at a *relsize* of 114.907, in contrast to two-pass coding where the *relsize* for

lossless reconstruction is 133.496. The much simpler homogeneous band quantization is clearly beaten by both zerotree coders except for the two-pass zerotree at very high quality; it must be noted that there is still plenty of room for improvement by adding a sophisticated statistical model and dedicated quantizers to homogeneous band quantization, as described in section 3.4.2.3.

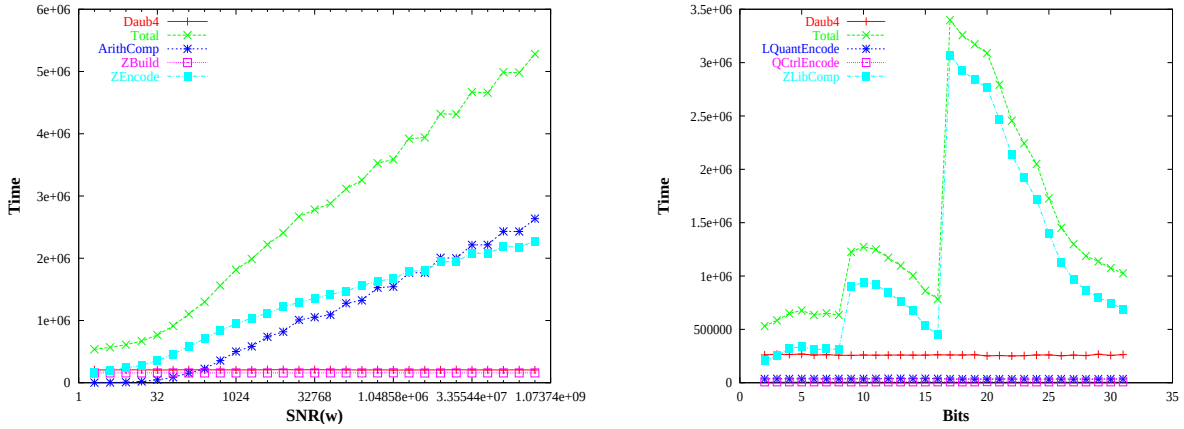


Figure 4.18: Lena detailed quantization timing comparisons

Figure 4.18 shows detailed timing measurements for the quantization of the Daubechies4-transformed *lena* image for two-pass zerotree coding (left) and homogeneous band quantization (right). The timings for two-pass zerotree coding are very similar to those of the one-pass variant in figure 4.2, with a maximum time under $6 \cdot 10^6 \mu s$ in both cases. In contrast to one-pass coding, the complexity of the arithmetic coder exceeds that of the zerotree encoding algorithm for higher quality, however. The reason is that in two-pass coding each node is only encoded as significant once in the dominant pass, and only the dominant pass needs aggregated node values. So as coding proceeds, fewer and fewer nodes remain that haven't been encoded as significant in a dominant pass yet and only when one of those is encoded as significant for the first time does the aggregation algorithm need to be rerun for that subtree; therefore, the complexity for the zerotree coder diminishes for high quality. That of the (adaptive) arithmetic coder increases, however, because now it operates on a six-symbol alphabet rather than the four-symbol alphabet of the one-pass coder: since the adaptive coder has to maintain an ordered list of probabilities for all symbols, it has higher complexity for large alphabets than it has for small ones.

The timings for band quantization to the right show two characteristic peaks of the *ZLib* compression stream for 9 and 17 bits. The reason for this is that the quantizer uses the nearest integer with at least as many bits as the quantization requires to hold the quantized value, and at 9 and 17 quantization bits this integer type changes from 8 to 16 bits and from 16 to 32 bits respectively, consequently the peaks are common to all test MDD. The *ZLib* compression stream used here obviously takes much more time for compressing integer values longer than 8 bits where the most significant bits are cleared than where they are filled (compare the time taken for 9 bits with that for 16 bits, or

even more extreme that for 17 bits compared to that for 31 bits). The reason for that lies most likely in the fact that e.g. a 16 bit type where the most significant byte is mostly unused appears as an interleaved sequence of bytes covering the full range (least significant bytes) and bytes covering only a very small range around 0 (most significant bytes); there will be many identical most significant bytes, therefore the dictionary coder has to check more potential pattern matches than if the significant bytes covered a larger range and consequently there were fewer identical values. As can be seen in figure 4.18, the time taken for quantization (*LQuantEncode* + *QCtrlEncode*) is insubstantial and by orders of magnitude smaller than that of all other modules; the total time is dominated by the *ZLib* compression stream, especially around 10 and 17 bits per coefficient. The total coding time is still noticeably smaller than that of the zerotree coders even in the most extreme case around 17 bits with $3.5 \cdot 10^6 \mu s$. For decompression, the speed advantage of homogeneous band quantization is much more severe due to *ZLib*'s faster decompression speed, leading to approximately 10 times the decompression speed of the zerotree coders.

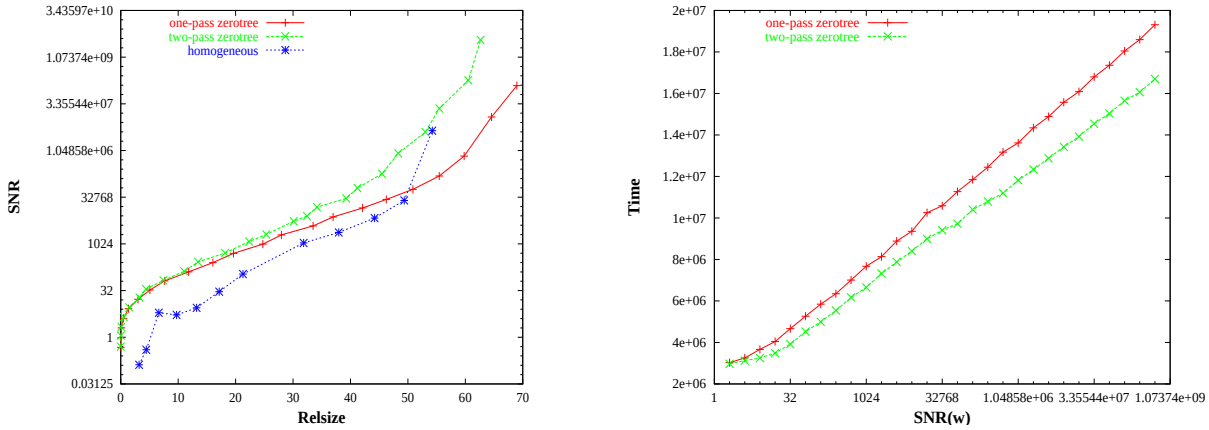


Figure 4.19: Tomogram quantization comparisons

Figure 4.19 shows the *size-snr* comparison graphs for the three quantization approaches applied to the Daubechies4-transformed *tomo_small* data cube (left) and the timing comparison between the one-pass and the two-pass zerotree coder (right). In this case, the two-pass zerotree coder performs best throughout the entire range, achieving lossless reconstruction at a *relsize* of 80.7758 in contrast to homogeneous band quantization with a *relsize* of 90.6414 and the one-pass coder with a *relsize* of 96.6089. The time taken by the two-pass coder is again very similar to that of the one-pass coder (where the two-pass coder is slightly faster).

The *size-snr* graphs for the other MDD show the following behaviour: for *cnig* and *movie_small* the one-pass coder wins, for *brain_small* the two-pass coder wins, and for *temperature* and *dkrz4d* the curves are almost identical with a tiny advantage for the two-pass coder. Since both *tomo_small* and *brain_small* are sparse MDD, this implies that the two-pass coder works best for sparse data, whereas the one-pass coder works best for densely populated data. The runtime differences of the two approaches are typically small

(on average the two-pass coder is around 10% faster). Homogeneous band quantization has clearly inferior performance almost everywhere, but has speed advantages, especially for decompression.

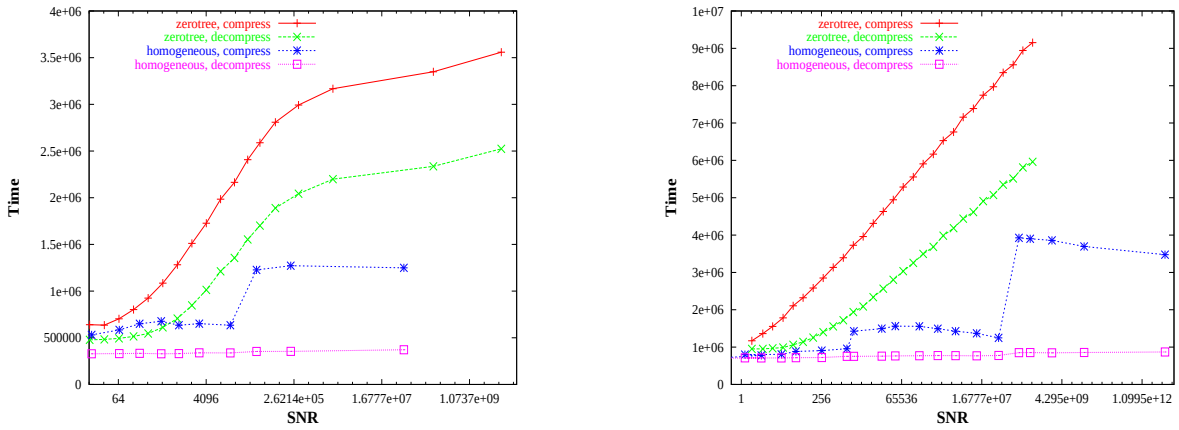


Figure 4.20: Quantization timing comparisons

Figure 4.20 compares the times taken for compressing and decompressing the *lena* (left) and *brain_small* (right) MDD with one-pass zerotree coding and homogeneous band quantization for given SNR values[8]). The SNR values are plotted logarithmically on the horizontal axis and the time taken for this SNR is plotted on the vertical axis. The characteristic peaks caused by the *ZLib* compression stream for homogeneous band quantization are again clearly visible, in particular for *brain_small* which shows both peaks due to its larger base type (i.e. in contrast to *lena*, *brain_small* could not be reconstructed without loss in less than 17 bits per coefficient). Homogeneous band quantization is noticeably faster at compression in either case, and dramatically faster at decompression with a curve almost independent of the SNR (i.e. the time taken for *ZLib* decompression is very small compared to the time taken for the inverse wavelet transformation). Whether this compensates the inferior rates is for the user to decide.

4.3.3 Compression Stream Comparisons

So far, all zerotree variants used the adaptive arithmetic coder introduced in section 1.2. The modular design also allows exchanging that compression stream for another one, however (or even a concatenation of streams). In this section, the performance of the adaptive arithmetic coder vs. a *ZLib* compression stream will be examined for some test MDD.

Figure 4.21 shows *size-snr* comparisons (left) and the timing comparisons (right) for the compression of the Daubechies4-transformed *lena* image with the adaptive arithmetic coder vs. *ZLib* compression for both one-pass and two-pass zerotree coding. Regarding the rate, the arithmetic coder performs better than the *ZLib* stream in both cases, where

[8]Note that these are SNR values, not SNR_w values, i.e. they belong to the reconstructed data in contrast to figure 4.18 where the horizontal axis represents the quality parameter SNR_w.

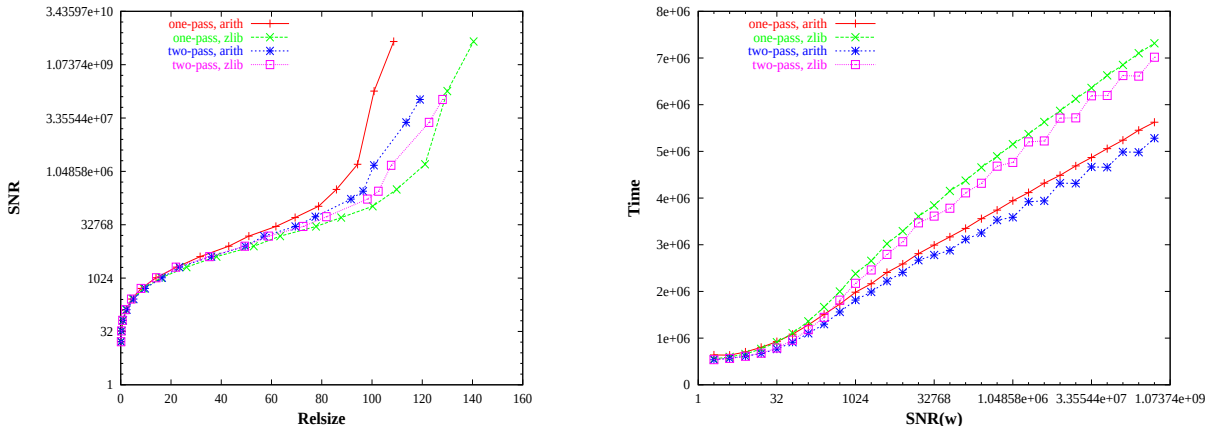


Figure 4.21: Lena compression stream comparisons

the advantage is more pronounced for the one-pass coder. As far as timings go, *ZLib* compression also takes noticeably longer than the adaptive arithmetic coder in both cases; this is caused by the tiny alphabet size, which speeds up the adaptive part of the coder enormously.

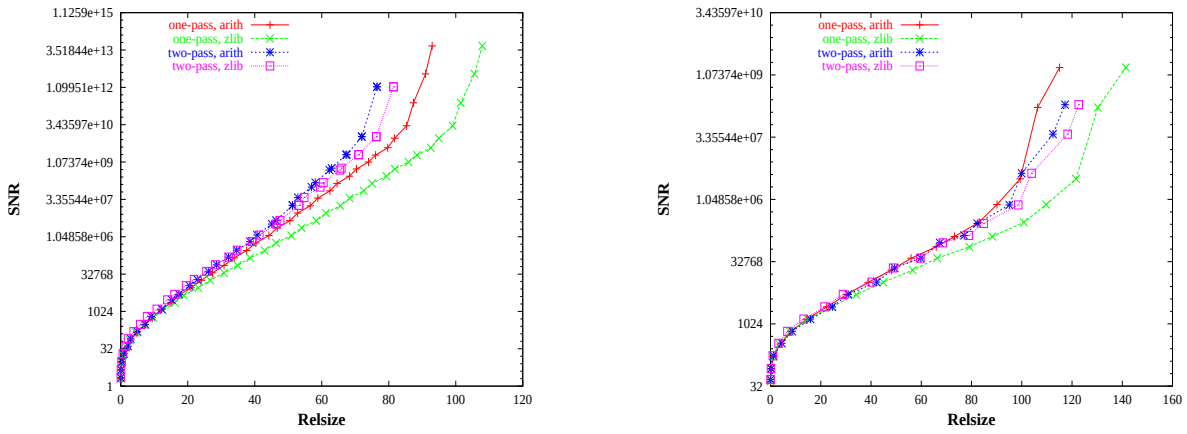


Figure 4.22: Brain and movie compression stream comparisons

Figure 4.22 shows the same *size-snr* comparison graphs for the *brain_small* (left) and *movie_small* (right) MDD. We can see that in all cases the adaptive arithmetic coder works better than the *ZLib* compression stream for the same type of zerotree coder, irrespective of which zerotree coder performs better for this particular MDD. This is true for all test MDD, thereby establishing the adaptive arithmetic coder as the universally best compression stream for zerotree coding.

4.3.4 Error Propagation

As established in section 3.4.1, the quantization error of the wavelet coefficients can accumulate during synthesis and thereby become considerably larger. The worst case error

propagation described there is typically by orders of magnitude larger than the actual one, however – for instance for the *lena* image with the Haar wavelet, equation (3.42) states a maximum error amplification factor of 3578 ($D = 2$, $H = \sqrt{2}$, and $j = 9$ scale levels). Although this factor is much larger than the actual one, there is usually an amplification of the maximum error. In order to find out typical amplification factors, the actual error amplification of some of the test MDD was measured by zerotree residual coding with $\mathrm{RES}_w = 10$ (i.e. no quantized wavelet coefficient differed from its exact value by more than 10) and checking the maximum difference of the data reconstructed from these quantized wavelet coefficients to the original data.

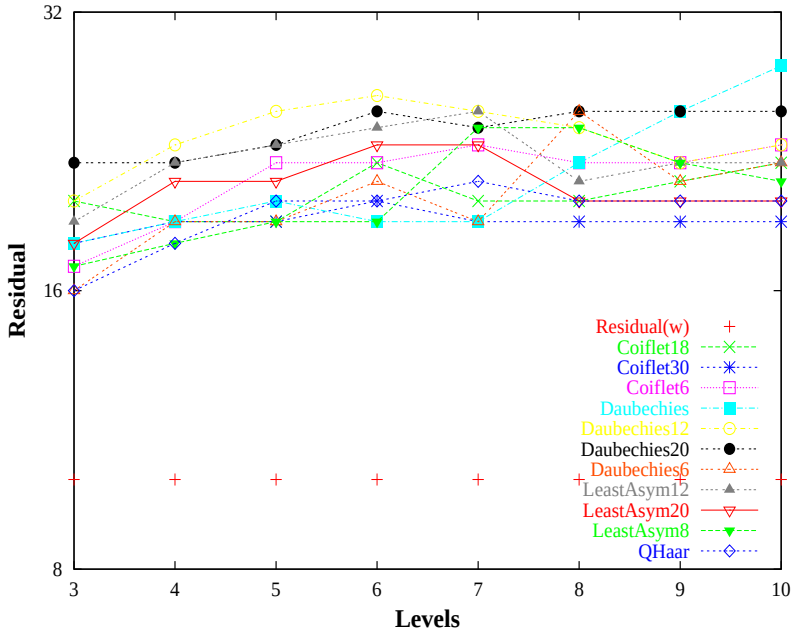


Figure 4.23: Lena error propagation comparisons

Figure 4.23 shows error propagation comparisons for a selection of wavelets when applied to the *lena* image. The horizontal axis represents the number of scale levels used, the vertical axis the maximum residual per cell in the reconstructed data. The quality parameter RES_w used for the wavelet coefficients is represented by the dots at vertical position 10. It is clearly visible that the error in the reconstructed data is larger than that of the quantized coefficients, so the error was amplified during synthesis – however nowhere near as much as the theoretical limit of 3578, but only by an average factor of 2. We can also see that the error grows with the number of scale levels for the most part, also in accordance with equation (3.42). There is no visible correlation between the filter length and the error magnitude, however, as e.g. for 10 scale levels the Daubechies 4-tap wavelet produces the largest error, followed by the Daubechies 20-tap wavelet, whereas the Coiflet 30-tap wavelet has the smallest error there.

Figure 4.24 shows the error propagation comparisons for the *tomo_small* (left) and *brain_small* (right) MDD. We can see that in both cases the average is somewhat higher

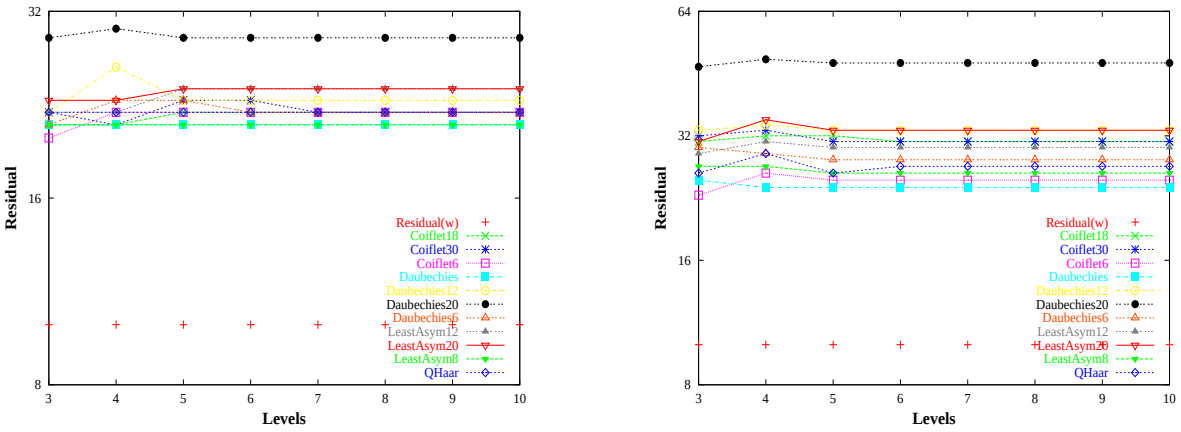


Figure 4.24: Tomogram and Brain error propagation comparisons

than it was for *lena*, which is again as expected from equation (3.42) due to the higher dimensionality. In particular the highest error is now magnified by a factor of approximately 3. The reason why the curves are constant after 7 respectively 6 scale levels is that these two MDD have shorter spatial extents than *lena*, so fewer scale levels are possible; attempting to code with more scale levels than physically possible reverts to coding with the maximum number of scale levels, hence the curves are constant in this area.

4.3.5 Predictor Usage

In contrast to the other compression techniques, wavelets can only be combined with interchannel predictors (see section 3.5). When the two test MDD with structured base types (*cnig*, *movie_small*) are preprocessed with the *delta* predictor, using green to predict the other two colours, the differences in the size-snr curves are very small. When comparing the *relsize* for lossless reconstruction with the Haar wavelet, the *relsize* is slightly higher for *cnig* (119.8 instead of 111.9 without the predictor) and slightly lower for *movie_small* (111.744 instead of 114.512), which mirrors the behaviour found in lossless compression (see section 4.2.2.2).

In the special case of lossy compression of RGB data (images, video), it is common practice to transform the channels (e.g. the RGB → YUV transformation introduced in section 3.5.1) and encode the resulting channels with different quality levels (typically the highest quality is used for the Y channel [60]), due to colour sensitivity of the human eye. Similar techniques can be used with wavelets and interchannel prediction, e.g. different SNR_w values for each channel. However, the evaluation of the results is a problem of image perception (and therefore highly application-specific), whereas this work concentrates on generic solutions, therefore this approach will not be analysed in more depth here; the interested reader is referred to [61], for example.

4.3.6 Is Lossy Good Enough?

Ideally, compression is lossless, but unfortunately Shannon's entropy equation (1.1) sets a hard limit on the compression rate achievable without loss. This rate can be improved by removing correlations via a model layer, e.g. by predictors, but there rarely is a model that allows a reduction of the rate by orders of magnitude; this is true especially for data that contains noise, because noise is random and truly random symbols do not compress at all[9]. Since this noise doesn't contain any information, a loss of the noise during compression can't lose any information either, so the compression is *logically* lossless, despite the *physical* loss of detail information. Therefore lossy compression is highly attractive if a certain noise floor level can be specified for the data.

Whether such a noise floor exists depends on the data; most MDD that are attractive for wavelet compression belong to the following categories:

sampled analogue data: most raster images, scientific measurements and similar types fall into this category, as do all the test MDD. The presence of noise is very likely in this kind of data, as can also be seen in figure 3.8, which means that loss up to a certain threshold is perfectly acceptable;

simulation data: numerical simulations typically have to solve partial differential equations and since there are no scalable, stable, exact solutions to this kind of problem, this is done using iterative approximations with a certain quality level, much like the SNR_w and RES_w parameters used in the compression engine. This implies the presence of a noise floor inversely proportional to the exactness of the solution.

Therefore, lossy compression can in many cases be good enough indeed, although using it efficiently is considerably more complex than lossless compression is. What degree of loss is acceptable is highly application-dependent, as residals and signal-to-noise ratios perfectly acceptable for images may be totally unacceptable for numerical simulation data, or quality requirements in medical images may be considerably higher than quality requirements in multimedia images.

Furthermore, the quality measure used plays an important role, the most useful ones being SNR and RES. The SNR sets a limit on the average difference of cells in the original data and the reconstructed lossy data, whereas the RES sets a limit on the maximum difference. Picking the more restrictive quality RES and assuming a signal-to-noise distance of n_s bits (i.e. the ratio of the largest value to the smallest relevant value is 2^{n_s}), we can achieve the following *relsize* for the *brain_small*, *temperature* and *dkrz4d* MDD with Haar and Daubechies 4-tap wavelets (using one-pass zerotree quantization again):

[9] A direct consequence of equation (1.1), as by definition random numbers are uncorrelated (i.e. there is no data model) and have the same probability per symbol.

Wavelet	n_s	*brain_small*	*temperature*	*dkrz4d*
Haar	8	35.8862	15.0835	22.7958
	10	46.6676	22.674	29.8596
	12	56.9473	29.9099	33.2047
	16		39.8643	49.1803
Daub4	8	44.2029	14.3602	23.045
	10	56.5514	20.9524	28.5096
	12	68.2509	28.6993	35.4545
	16		42.2255	48.3789

For *brain_small*, only $n_s = 8$ with the Haar wavelet shows a small improvement over the best lossless *relsize* of 37.0645 (see section 4.2.1.2). For the floating point MDD, on the other hand, rates can be improved considerably within reasonable noise floor limits, which can be seen in the following table, where the best *relsize* of the previous table is set in relation to the best *relsize* achievable with lossless compression and optional predictors (see section 4.2.2.1).

n_s	*brain_small*	*temperature*	*dkrz4d*
8	0.9682	0.2795	0.4103
10	1.2591	0.4078	0.5132
12	1.5364	0.5585	0.5977
16		0.8217	0.8709

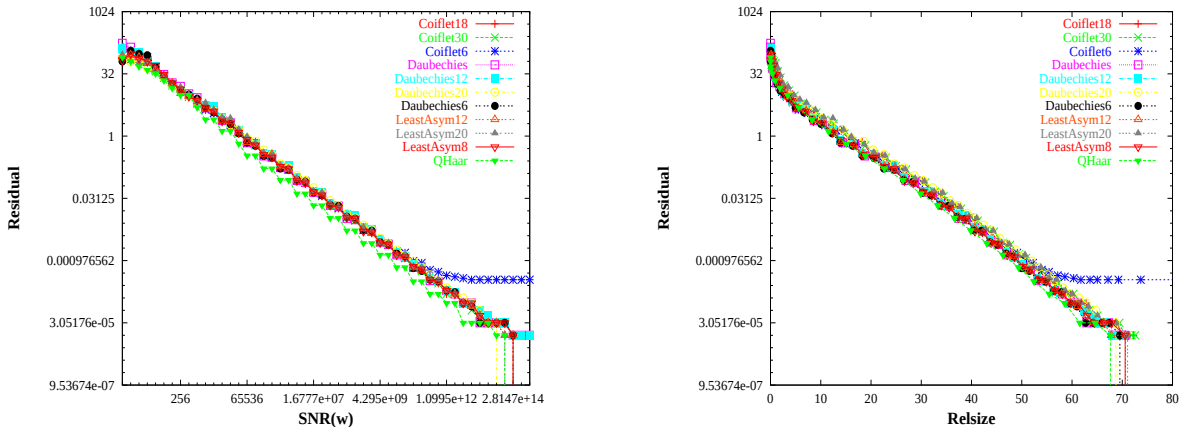
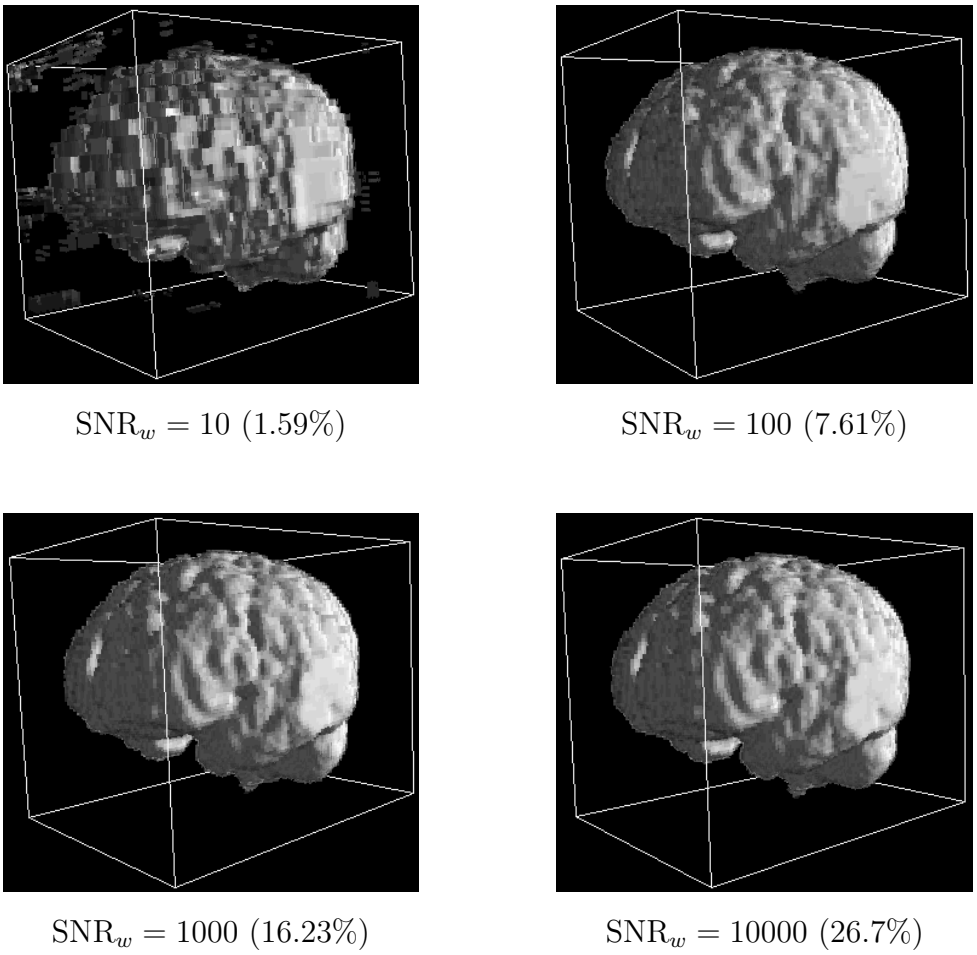


Figure 4.25: Temperature *quality-residual* and *size-residual* graphs

So even with n_s as high as 16, the *relsize* of *temperature* and *dkrz4d* is lower with lossy compression than it is with lossless compression. For $n_s = 10, 12$ the size of the data compressed with lossy techniques is around 40-60% that of using lossless techniques for these MDD. So provided a signal-to-noise distance of 10–12 bits is acceptable, the rates for lossy compression can improve upon the lossless ones considerably, even with the more restrictive RES quality measure. Figure 4.25 shows *quality-residual* (left) and *size-residual*

(right) graphs for *temperature* with one-pass SNR zerotree coding, which can be used to determine what quality parameter SNR_w is needed for a desired maximum residual (*quality-residual* graph) and what *relsize* can be achieved with that residual (*size-residual* graph).



$\text{SNR}_w = 10$ (1.59%) $\text{SNR}_w = 100$ (7.61%)

$\text{SNR}_w = 1000$ (16.23%) $\text{SNR}_w = 10000$ (26.7%)

Figure 4.26: Lossy coding of Brain

Figure 4.26 shows the *brain_small* MDD coded at different quality levels ($10 \leq \text{SNR}_w \leq 10000$). The resulting data cubes were visualized with the $\mathcal{RasDaMan}$ client application *rView* [19] using volume rendering. The lowest quality image ($\text{SNR}_w = 10$) has noticeable compression artefacts, but the basic shape can still be recognized and the compressed data has a *relsize* of only 1.59%. There is hardly any visible difference between the higher quality images or the original in figure 4.1 on page 105, so for visualization purposes a compression with $\text{SNR}_w = 100$ is sufficient, which yields *relsize* 7.61% and is therefore about five times smaller than the best lossless size.

4.3.7 Conclusions for Lossy Compression

Lossy compression is considerably more complex to handle than lossless compression due to the new concept of distortion and the parameters required to control it. The size-snr curves displayed in section 4.3.1 allow weighing compressed sizes against the distortion introduced by loss for all the test MDD and can – for example in combination with SNR_w-SNR and SNR_w-size tables for zerotree coding – be used to determine what SNR_w one has to use to be able to reconstruct the data with the desired SNR or *relsize*. It is important to stress the difference between SNR_w and RES_w, the quality parameters used for encoding the wavelet coefficients, and SNR and RES which compare the original data with the lossy reconstruction. In most cases RES $>$ RES_w, i.e. errors in the wavelet coefficients are amplified during wavelet synthesis, in particular for high dimensionality or a large number of scale levels, although the actual error amplification measured is by orders of magnitude smaller than the maximum theoretical amplification calculated in equation (3.42). This was studied in section 4.3.4, where typical amplification factors between 2 and 3 were measured. Common observations from the *size-snr* graphs in section 4.3.1 are

- often the differences between the rates achieved with the various wavelets are very small, in particular for *lena*, *cnig*, *temperature* and *dkrz4d*. Long filters typically only have clear advantages for very smooth data and low quality;
- on average the best size-snr ratio is achieved by the Haar wavelet, in particular for high quality. Even when the Haar wavelet is not the best one, it is usually not far from the optimum, at least for high quality. It performs particularily well for sparse or noisy data (*tomo_small*, *brain_small*). Overly long filters (more than 12 taps) hardly ever perform better than shorter ones;
- although lossless reconstruction is possible in most cases, the rates are always inferior to those of a dedicated lossless compression technique like ZLib with appropriate predictors, even for floating point arrays;
- the time taken for zerotree-encoding Haar-transformed data is noticeably higher than it is for the other wavelets, in particular for high-dimensional spaces (*dkrz4d*), due to the current implementation of the aggregation code;
- with the exception of the Haar wavelet, the time taken for zerotree encoding and decoding is comparable, they are not largely asymmetrical like in *ZLib*-based techniques;
- the maximum throughput values measured for zerotree coding at $SNR_w = 4096$ are below the bandwidth of standard 10MBit/s Ethernet for all test MDD ($t_c \leq 520$, $t_d \leq 1039$). This is an important observation for transfer compression in section 4.4.

The comparison of the three quantization approaches in section 4.3.2 showed that the two zerotree coding variants achieved the best overall size-snr ratios, where for some test

MDD the one-pass coder performed better (*lena*, *cnig*, *movie_small*), for others the two-pass coder (*tomo_small*, *brain_small*) and for the rest both had almost identical performance. Judging by the test MDD, the two-pass coder is superior for sparse data, but more tests need to be made for any final statements in this respect. Homogeneous band quantization is far behind in terms of size-snr ratio[10], but is much faster, in particular for decompression; on the other hand, it must be noted that homogeneous band quantization was used in its simplest form, with an identical number of bits and a uniform linear quantizer for all bands, and that this can be improved with a sophisticated statistics module like in [12, 13]. It is unlikely that this will better the rates achievable with zerotree coding, but it might provide a faster alternative with comparable rates.

Regarding the choice of the compression stream to use for zerotree coding, the default adaptive arithmetic coder proved to be the best alternative in section 4.3.3, being both faster and achieving better rates than the *ZLib* stream it was compared with; this allows a clear recommendation to use the default compression stream and effectively reduces the parameters of the wavelet engine by one.

The wavelet engine is by far the most complex module of the compression engine and needs many more parameters than its lossless counterpart. Whether it offers advantages over lossless compression depends mostly on the kind of data it is applied to and what amount of distortion is acceptable for that data. Section 4.3.6 gave some examples of how the wavelet engine can provide results far superior to the rates possible with any lossless compression system currently available in the compression engine, depending on the level of distortion, and how distortion does not necessarily mean loss of information as long as the distortion is within the data's noise floor. The question of when to use wavelets is therefore highly application-specific, and can not be answered in general without detailed information on acceptable distortion levels. Even then some experimentation is required regarding what wavelet filter and quantization technique to use for optimum rates, because amplification of the coefficient quantization error makes it impossible to predict the error of the reconstructed data from the quantization error apart from the worst case factor in equation (3.42), which is typically by orders of magnitude larger than the actual amplification factor.

4.4 Transfer Compression

Besides savings in storage space, compression can also speed up the entire DBMS due to reduced communication- and IO times. As shown in section 3.7, it depends on the compression and decompression times (t_c, t_d) as well as the size of the uncompressed data m, the compression ratio r and the bandwidth B of the communication channel whether the use of compression speeds up total system performance. Let again the abbreviation t_{cd} stand for either $t_c + t_d$ for sequential compression and decompression or $\max(t_c, t_d)$ for parallel operation. Then transfer compression improves performance if $t_{cd} < \frac{m}{B}(1 - r)$, or

[10]With the exception of very high quality, where it sometimes manages to beat one, but never both zerotree coders.

$\frac{m}{t_{cd}} > \frac{B}{1-r}$. $\frac{m}{t_{cd}}$ is the throughput of the compression and decompression modules involved in the transfer compression; a compression algorithm can therefore be ruled out for transfer compression completely and irrespective of its *relsize* if its throughput for compression or decompression is smaller than the communication channel's bandwidth – or in other words: if compression and decompression take longer than transferring the uncompressed data over the communication channel, transfer compression with this algorithm can't benefit system performance regardless of the compression rate it achieves. Thus the throughput values listed in sections 4.2 and 4.3 can be used to filter out techniques feasible for transfer compression. Another logical filtering step is ignoring all techniques which didn't compress the data at all.

We will now check which techniques have potential for transfer compression, assuming a standard 10MBit/s (= 1250kB/s) ethernet network. As noted in section 4.3.7, throughput values for all lossy wavelets are lower than that on the reference machine (at least with zerotree quantization), so lossy wavelets can be ruled out entirely for transfer compression with the given hardware (this statement will eventually have to be reevaluated once the hardware has improved sufficiently). This leaves only lossless compression with the following candidates:

RLE: the throughput is considerably higher than the ethernet bandwidth, but it achieves only compression for *tomo_small* and *brain_small*, so RLE transfer compression should only be attempted for those two test MDD;

ZLib: the throughput is higher than the ethernet bandwidth for all test MDD but *temperature* and *dkrz4d*; closer inspection reveals that only the transfer times for *tomo_small* and *brain_small* can improve with transfer compression because *relsize* is too large for the other MDD.

SepZLib and SepRLE are also candidates for *movie_small*, but can be ruled out because for both of them $t_c + \frac{mr}{B} > mB$ (the time for transferring the raw data is 675313μs, SepRLE takes 840195μs and SepZLib takes 1038384μs for compression, decompression and data transfer). The same is true for all Haar wavelets, leaving only those two compression techniques above as candidates for transfer compression, with *tomo_small* and *brain_small* as test MDD; predictors will be ignored at this point. Because other overhead is added to the pure (de)compression and transfer times, there is no guarantee that a theoretically advantageous technique will also perform well in real life, but the theoretical threshold is a necessary minimum requirement for improvements.

The following table contains the transfer compression times for *tomo_small*, *brain_small*, *tomo_full* and *brain_full* when using RLE or ZLib for transfer compression in $\mathcal{RasDaMan}$. The *tomo_full* and *brain_full* MDD were originally used to create *tomo_small* and *brain_small* by scaling them to half their size in each dimension and are included in these measurements to increase the data volume for the measurements by a factor of 8 and thereby reduce the impact of not compression-related DBMS overhead on the measurements. Timings are in milliseconds:

Type	*tomo_small*	*brain_small*	*tomo_full*	*brain_full*
NoComp	1597	1218	13456	8985
RLE	990	841	7905	5585
ZLib	3240	1933	17280	11851

Obviously, RLE improves the total response time in all cases for these MDD, saving 30-40% of the transfer time for the uncompressed data. ZLib slows down the total response time, but the relative slowdown is considerably smaller for the full MDD than it is for the scaled down versions.

Due to the symmetry of the compression engine, it is also possible to transfer tiles that were stored compressed at the server and are fully contained in the query box directly to the client, so there is no compression overhead, only decompression at the client. Under these conditions, ZLib can potentially speed up the total response time for all test MDD and RLE once again for *tomo_small* and *brain_small* because its decompression throughput is high enough and it manages to compress these two test MDD. In this case, database IO will also be sped up, because the compressed tiles are smaller and therefore take less time to load from disc, but since IO throughput is by orders of magnitude larger than the standard ethernet bandwidth, this effect is small compared to the improved communication times. The following table shows the total transfer times when the data was already stored compressed in the database; timings are again in milliseconds:

Type	*lena*	*cnig*	*tomo_small*	*brain_small*	*movie_small*	*temperature*	*dkrz4d*
NoComp	427	929	1597	1218	1080	4494	5350
RLE			818	765			
ZLib	388	831	777	719	856	3361	4742

In this case, the situation is reversed: not only does ZLib improve total response times in all cases due to its asymmetric compression/decompression times (see section 4.2.1.2), it even beats RLE because its lower transfer time caused by a better *relsize* has more effect on the total time than its lower decompression throughput. Therefore it can be recommended to store all data compressed with ZLib by default, provided the data is read more often than written to, since decompression overhead was outweighed by improved transfer times for all test MDD.

Chapter 5

Conclusions and Future Work

The Road goes ever on and on
Out from the door where it began.
Now far ahead the Road has gone,
Let others follow it who can!
Let them a journey new begin,
But I at last with weary feet
Will turn towards the lighted inn,
My evening-rest and sleep to meet.

J.R.R. Tolkien, *The Lord of the Rings*

Multidimensional Discrete Data appears in many different application areas, including such diverse fields as multimedia, medicine or scientific computing, each with specialized tools and storage formats. The array DBMS $\mathcal{R}asDa\mathcal{M}an$ unites generic multidimensional arrays under the MDD concept and provides common database services like transaction management and a query language. Due to the enormous size typically found in MDD, compression becomes an important component to reduce storage requirements and potentially improve system performance by also reducing communication- and IO times.

In this thesis, the requirements for an MDD compression engine were analysed and used to develop a modular design introduced in chapter 3, including a dynamic parameter system for flexible configuration of the system's various components. The engine has a standard two-layer architecture found in most advanced compression techniques, with a model layer exploiting MDD properties and a compression layer for the actual compression of the transformed data with tranditional compression techniques. The work concentrated on the development of model layers for MDD compression with different levels of complexity, in particular

Channel Separation for structured base types, i.e. compressing values for each channel separately instead of using the raw tile format, which interleaves channels;

Predictors which calculate approximate values for some cells and express these cells' values relative to the approximate values (deltas). If the predictor matches the data

model, most of these deltas will cover a very small range around 0, which allows more efficient compression. Approximate values can be calculated using spatially neighbouring cells (intrachannel predictors) or neighbours across channels (interchannel predictors);

Wavelets which transform the channels individually into multiresolution representations with coarse approximations and various levels of detail information which allow successively refining these coarse approximations until the original data can be reconstructed. Provided the wavelet base matches the data well, most of the detail information will be sparse or near sparse and allow far more efficient compression than the untransformed data, in particular lossy compression where small detail coefficients are zeroed to further improve compression rates.

On the subject of wavelets, the thesis concentrated on the generalization of multiresolution wavelet transformations and the efficient quantization of the resulting wavelet coefficient arrays. This led to a wavelet engine architecture consisting of three major components, transformation, quantization and compression, all of which can be exchanged freely due to modular design. The quantization problem gave birth to the Generalized Zerotree described in section 3.4.3, a hierarchical structure exploiting relationships between wavelet coefficients on neighbouring scale levels, which hitherto existed only in specialized implementations for 2 and 3 dimensions; the older and simpler homogeneous band quantization is still available as a less expensive but also less efficient alternative.

The thesis closed with an evaluation of the performance of the compression engine's major components on a set of seven test MDD from different application areas, covering various base types and dimensionality between 2 and 4 in chapter 4. For lossless compression, techniques based on the standard compression library *ZLib* provided the best rates, which could often be improved considerably by the application of predictors or channel separation. For the lossy wavelet engine, comparisons were made between the different wavelet filters, the quantization types and the compression streams. In contrast to the lossless case, there is usually no clear winner which performs equally well on all test MDD, neither is it possible to guarantee a level of reconstruction quality without excessive restrictions on the quality of the coefficient quantization because of error amplification in the wavelet synthesis stage, so experimentation is required to achieve optimum results. The results measured for lossy compression provide valuable heuristics for the lossy compression of various MDD categories, however. They also prove that the rates possible with lossless compression can be improved considerably in the lossy case, depending on what distortion levels are acceptable.

But especially in compression, work is never truly finished. Topics for future work can be divided into the two major categories *Compression* and *Integration*:

Compression: this area concerns improvements of the compression engine as such, independent of the DBMS; this includes the following issues:

- improve the efficiency of the adaptive arithmetic coder, e.g. based on the work in [37];

- add more predictors, for example a RGB $\leftrightarrow$ YUV transformation for RGB data;
- add more specialized compression techniques, for instance an approach for the lossless compression of floating point values described in [22];
- improve boundary treatment for wavelet transformations. The current periodic approach can introduce singularities at the boundaries which cause large detail coefficients and compromise efficiency, in particular for low rates. Promising solutions are the mirror boundary condition or the use of shorter filters at the boundaries;
- add more wavelet transformations, for instance biorthogonal wavelets often used in image compression [51], or wavelet packets [36];
- improve homogeneous band quantization via a sophisticated statistics module and more specialized quantizers like in [12, 13] to obtain a less expensive alternative to zerotree coding with comparable rates;
- improve the zerotree aggregation code to address the speed penalties measured in particular for encoding Haar-transformed wavelet coefficients. Another zerotree improvement is the addition of some newer tree alphabets, for instance the one in [10];
- add a *rate* or *relsize* termination criterion to the zerotree coder. This would allow encoding an MDD by size rather than by quality, but will require extensions of the compression streams to get meaningful information about the compressed size while the stream is still active;
- use different compression streams for dominant and subordinate pass in zerotree coding for better compression rates;
- add alternative wavelet quantization modules, for example a generalized SPIHT [46].

Integration: this area concentrates on improving the integration of the compression engine into the DBMS. That means in particular extending the query optimizer to exploit compression where possible, such as

- smart materialization of compressed data by exploiting properties of the compression technique used. That means e.g. only decompressing those channels that are actually needed and ignoring those that will be discarded later on via type projections, if the compression technique does channel separation. Another example would be limiting the reconstruction quality or resolution, for instance when the data will be scaled down, which can easily be achieved for lossy wavelet compression techniques with zerotree coding;
- execute operations directly on compressed tiles without an explicit decompression-compression cycle. This is usually only feasible for very simple compression types like *RLE*; but some operations are possible in far more

complicated cases, for example multiplying every cell in a zerotree-coded MDD with a constant value can be done by the compression engine in just *one* operation, namely multiplying the initial zerotree threshold value stored as meta data with this constant value;

- add a cost model for compression to the query optimizer. This would allow automating the use of transfer compression, or the optional internal use of compression for communication between parallel nodes, among other things.

Bibliography

[1] P. Baumann, P. Furtado, R. Ritsch, N. Widmann: *The RasDaMan Approach to Multidimensional Database Management.* Proceedings of the SAC'97, San Jose, California, 1997

[2] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, N. Widmann: *The Multidimensional Database System RasDaMan.* Proceedings ACM SIGMOD International Conference on Management of Data, 1998

[3] P. Baumann: *A Database Array Algebra for Spatio-Temporal Data and Beyond.* NGITS 99, Zikhron Yaakov, Israel, 1999 (Springer LNCS 1649)

[4] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, N. Widmann: *Spatio-Temporal Retrieval with RasDaMan.* Proc. Very Large Data Bases (VLDB), Edinburgh, 1999, pp. 746-749.

[5] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, N. Widmann: *Cross-Dimensional Sensor Data Management.* Proceedings of the Fourth International Airborne Remote Sensing Conference and Exhibition, 1999

[6] C.F. Barnes, E.J. Holder: *Successive Approximation Quantization with Generalized Decoding for Wavelet Transform Image Coding.* 27th Asilomar Conference on Signals, Systems and Computers, 1993

[7] R. Bayer: *The Universal B-Tree for Multidimensional Indexing: General Concepts.* World-Wide Computing and its Applications '97 (WWCA '97), Tsukuba, Japan, 1997

[8] L. Bottou, P. Haffner, P. Howard, P. Simard, Y. Bengio, and Y. LeCun: *High Quality Document Image Compression with DjVu.* Journal of Electronic Imaging, 1998

[9] R. Cattell: *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, San Mateo, California, USA, 1997

[10] Y. Chen, J. Shapiro: *Three-Dimensional Subband Coding of Video Using the Zero-Tree Method.* SPIE Symposium on Visual Communications and Image Processing, 1996

[11] Z. Chen, P. Seshadri: *An Algebraic Compression Framework for Query Results.*

[12] C. Chrysafis, A. Ortega: *Context-Based Adaptive Image Coding.* 30th Asilomar Conference on Signals, Systems and Computers, 1996

[13] C. Chrysafis, A. Ortega: *Efficient Context-Based Entropy Coding for Lossy Wavelet Image Compression.* Data Compression Conference (DCC), Snowbird, Utah, 1997

[14] C. Chrysafis: *Wavelet Image Compression Rate Distortion Optimizations and Complexity Reductions.* PhD thesis, Faculty of the Graduate School, University of Southern California, 2000

[15] C.K. Chui: *An Introduction to Wavelets.* Academic Press Inc, 1992

[16] W.P. Cockshott, D. McGregor, N. Kotsis, J. Wilson: *Data Compression in Database Systems.* DEXA Workshop on Advanced Database Applications 98

[17] G.V. Cormack: *Data Compression on a Database System.* Communications of the ACM, Dec.1985, Volume 28

[18] I. Daubechies: *Ten Lectures on Wavelets.* CBMS-NSF Regional Conference Series in Applied Mathematics, CBMS 61, 1992

[19] A. Dehmel: *Visualizing Multidimensional Raster Data with rView.* Database and Expert Systems Applications (DEXA) 2000, Springer-Verlag, Heidelberg

[20] A. Dehmel: *Designing a Compression Engine for Multidimensional Raster Data.* Database and Expert Systems Applications (DEXA) 2001, Springer-Verlag, Heidelberg

[21] A. Dehmel: *Encoding Multidimensional Wavelet Coefficients Using the Generalized Zerotree.* 35th Asilomar Conference on Signals, Systems and Computers, 2001

[22] V. Engelson, P. Fritzson, D. Fritzson: *Lossless Compression of High-Volume Numerical Data from Simulations.* Linköping Electronic Articles in Computer and Information Science ISSN 1401-9841, Vol. 5 (2000): nr 011

[23] A. Fournier et. al.: *Wavelets and their Applications in Computer Graphics.* SIGGRAPH '95 Course Notes

[24] P. Furtado: *Storage Management of Multidimensional Arrays in Database Management Systems.* PhD thesis, Technical University of Munich, 1999.

[25] M. Griebel, F. Koster: *Adaptive Wavelet Solvers for the Unsteady Incompressible Navier-Stokes Equations.* J. Malek, M. Rokyta (Eds.), Advances in Mathematical Fluid Mechanics, Springer Verlag, also as Preprint No. 669 (2000), University of Bonn

[26] HDF homepage: `http://hdf.ncsa.uiuc.edu/`

[27] D.A. Huffman: *A Method for the Construction of Minimum Redundancy Codes.* Proceedings of the IRE, 40:1098-1101, 1951

[28] I. Ihm, S. Park: *Wavelet-Based 3D Compression Scheme for Very Large Volume Data.* Graphics Interface '98, pp107-116, Vancouver, Canada, 1998

[29] The International Organization for Standardization (ISO): *Database Language SQL.* ISO 9075, 1992(E)

[30] JBIG Committee: *Coding of Still Pictures (JBIG, JPEG)* ISO/IEC JTC 1 / SC 29 / WG 1, N 1359, July 1999.

[31] B.J. Kim, Z. Xiong, W.A. Pearlman: *Low Bit-Rate, Scalable Video Coding with 3D Set Partitioning in Hierarchical Trees.* IEEE Transactions on Circuits and Systems for Video Technology, Vol. 10, 2000

[32] F. Koster, M. Griebel, N. Kevlahan, M. Farge, K. Schneider: *Towards an Adaptive Wavelet-Based 3D Navier-Stokes Solver.* E. Krause, E. Hirschel (Eds.) Notes on Numerical Fluid Mechanics (1998)

[33] Lena Soderberg, *Playmate of the Month*, Playboy Magazine, November 1972

[34] W.E. Lorensen, H.E. Cline: *Marching Cubes: A High Resolution 3D Surface Construction Algorithm.* ACM Computer Graphics, Volume 21, Number 4, 1987

[35] S. Mallat: *A Wavelet Tour of Signal Processing.* Academic Press 1999 (2nd edition)

[36] F.G. Meyer, A. Averbuch, J.O. Strömberg, R.R. Coifman: *Fast Wavelet Packet Image Compression.* Data Compression Conference (DCC) 98, Snowbird, Utah, 1998

[37] A. Moffat, R.M. Neal, I.H. Witten: *Arithmetic Coding Revisited.* ACM Transactions on Information Systems, Vol. 16, No. 3, 1998

[38] *PCL 5 Printer Language Technical Reference Manual.* Order No. 5961-0509, Hewlett-Packard Co., P.O. Box 1145, Roseville, CA 95678, 1992

[39] S.M. Perlmutter, K.O. Perlmutter, P.C. Cosman: *Vector Quantization with Zerotree Significance Map for Wavelet Image Coding.* 29th Asilomar Conference on Signals, Systems and Computers, 1995

[40] Charles A. Poynton: *Frequently Asked Questions about Color.* `http://www.inforamp.net/~poynton/ColorFAQ.html`

[41] G. Randers-Pehrson et al.: *PNG (Portable Network Graphics) Specification, Version 1.2.* `http://www.libpng.org/pub/png/pngdocs.html`, July 1999

[42] R. Ritsch: *Optimization and Evaluation of Array Queries in Database Management Systems.* PhD thesis, Technical University of Munich, 1999.

[43] P. Roland, J. Young, T. Lindeberg, G. Svensson, J. Frederiksson, H. Halldorsson, L. Forsberg, T. Risch, P. Baumann, A. Dehmel, K. Zillers: *A Database Generator for Human Brain Imaging.* TRENDS in Neurosciences, Vol.24, No.10, October 2001

[44] L. Rosenblum et al: *Scientific Visualization – Advances and Challenges.* Academic Press Ltd., 1994

[45] A. Said, W.A. Pearlman: *An Image Multiresolution Representation for Lossless and Lossy Compression.* SPIE Symposium on Visual Communications and Image Processing, Cambridge, MA, 1993.

[46] A. Said, W.A. Pearlman: *A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees.* IEEE Transactions on Circuits and Systems for Video Technology, Vol. 6, June 1996

[47] K. Sayood: *Introduction to Data Compression.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[48] C.E. Shannon: *A Mathematical Theory for Communication.* Bell System Technical Journal, 27:379-423, 623-656, 1948

[49] J. Shapiro: *Embedded Image Coding using Zero-Trees of Wavelet Coefficients.* IEEE Transactions on Signal Processing, pp. 3445-3462, 1993

[50] J. Shapiro: *Smart Compression using the Embedded Zerotree Wavelet (EZW) Algorithm.* 27th Asilomar Conference on Signals, Systems and Computers, 1993

[51] A.N. Skodras, C.A. Christopoulos, T. Ebrahimi: *JPEG2000: The Upcoming Still Image Compression Standard.* Proceedings of the 11th Portuguese Conference on Pattern Recognition (RECPA), 2000

[52] N. Strobel, S.K. Mitra, B.S. Manjunath: *Progressive-Resolution Transmission and Lossless Compression of Color Images for Digital Image Libraries.* Image Processing Laboratory University of California, Santa Barbara 93106.

[53] *TIFF Revision 6.0 Specification*, p42; Aldus Corporation, Seattle, 1992

[54] J.R.R. Tolkien: *The Lord of the Rings.* George Allen and Unwin, London, 1954–1955

[55] A. Trott, R. Moorhead, J. McGinley: *Wavelets Applied to Lossless Compression and Progressive Transmission of Floating Point Data in 3D Curvilinear Grids.* IEEE Visualization, 1996.

[56] G. Uytterhoeven: *Wavelets: Software and Applications.* PhD thesis, Department of Computer Science, K.U. Leuven, 1999

[57] V.D. Vaughn, T.S. Wilkinson, L.S. Kalman: *Multispectral Image Compression for Future LANDSAT Remote Sensing Systems.* 27th Asilomar Conference on Signals, Systems and Computers, 1993

[58] J.D. Villasenor, B. Belzer, J. Liao: *Wavelet Filter Evaluation for Image Compression.* IEEE Transactions on Image Processing, 1995

[59] J.S. Vitter, M. Wang: *Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets.* SIGMOD '99, Philadelphia PA.

[60] G.K. Wallace: *The JPEG Still Picture Compression Standard.* Communications of the ACM No.4, Vol 34, Apr 1991.

[61] A.B. Watson, G.Y. Yang, J.A. Solomon, J. Villasenor: *Visual Thresholds for Wavelet Quantization Error.* SPIE Proceedings Vol. 2657, The Society for Imaging Science and Technology, 1996

[62] T. Westmann, D. Kossmann, S. Helmer, G. Moerkotte: *The Implementation and Performance of Compressed Databases.* Reihe Informatik 3 / 1998

[63] I.H. Witten, R.M. Neal, J.G. Cleary: *Arithmetic Coding for Data Compression.* Communications of the ACM, June 1987, Vol 30, Number 6

[64] J. Ziv, A. Lempel: *A Universal Algorithm for Data Compression.* IEEE Transactions on Information Theory, IT-23(3), 1977

[65] J. Ziv, A. Lempel: *Compression of Individual Sequences via Variable-Rate Coding.* IEEE Transactions on Information Theory, IT-24(5), 1978

[66] Z. Zhu, R. Machiraju, B. Fry, R. Moorhead: *Wavelet-Based Multiresolution Representation of Computational Field Simulation Datasets.* Proceedings of the 8th IEEE Visualization Conference, 1997

[67] ZLib homepage: `http://www.info-zip.org/pub/infozip/zlib/`

Appendix A

Proof for Lossless Haar Wavelets

This part of the appendix proves the assertion stated in section 3.3.4.1 that lossless Haar transformations for integer types are possible without having to use larger integers to hold either the average coefficients $c_i^j = \frac{1}{2}(c_{2i}^{j+1} + c_{2i+1}^{j+1})$ or the detail coefficients $d_i^j = \frac{1}{2}(c_{2i}^{j+1} - c_{2i+1}^{j+1})$. It is possible to solve this problem for integer types by storing $c'^j_i = (s_r(2c_i^j) + o_d 2^{B-1}) \bmod 2^B$ and $d'^j_i = (2d_i^j) \bmod 2^B$, where $s_r(x)$ performs a bitwise, sign-preserving shift-right[1] of x, B is the number of bits in the base type and $o_d = 0$ if $2d_i^j$ can be represented in B bits or $o_d = 1$ otherwise. Then the original values can be restored using $c_{2i}^{j+1} = (c'^j_i + s_r(d'^j_i + 1)) \bmod 2^B$ and $c_{2i+1}^{j+1} = (c'^j_i - s_r(d'^j_i)) \bmod 2^B$. Note that this special coding format is only used for integer types, whereas for floating point types c_i^j and d_i^j are used – in this case, lossless transformation can no longer be guaranteed if either value can't be represented exactly in the machine's precision, however!

Regarding lossless reconstruction despite the $s_r()$ operator in c'^j_i, we can first establish that this can only involve loss if c_{2i}^{j+1} is even and c_{2i+1}^{j+1} is odd or the other way around. In these cases, we can write the odd coefficient as the sum of the nearest smaller even value plus 1. Because a wavelet transformation is a linear operation (i.e. given two sequences (s_i), (t_i), $(\mathcal{W}(s_i + t_i))(a, b) = (\mathcal{W}s_i)(a, b) + (\mathcal{W}t_i)(a, b))$, we can express the wavelet transformation of c_{2i}^{j+1}, c_{2i+1}^{j+1} as the wavelet transformation of the two nearest smaller even values (where the $s_r()$ operator is lossless) plus the wavelet transformation of the pair $(1, 0)$ in case c_{2i}^{j+1} was odd or $(0, 1)$ in case c_{2i+1}^{j+1} was odd. Therefore proving the correctness of the above encoding format for values $(0, 1)$ and $(1, 0)$ is sufficient to prove correctness for all possible pairs of even/odd values. For input values $(0, 1)$ we get $c'^j_i = s_r(1) = 0$ and $d'^j_i = -1$. Reconstruction is done using $c_{2i}^{j+1} = c'^j_i + s_r(d'^j_i + 1) = 0 + s_r(-1 + 1) = 0$ and $c_{2i+1}^{j+1} = c'^j_i - s_r(d'^j_i) = 0 - s_r(-1) = 1$, which are the correct input values. For input values $(1, 0)$ we get $c'^j_i = s_r(1) = 0$ and $d'^j_i = 1$; the reconstructed values are $c_{2i}^{j+1} = 0 + s_r(1 + 1) = 1$ and $c_{2i+1}^{j+1} = 0 - s_r(1) = 0$, which are the correct input values too, thereby proving the encoding to be correct.

[1] The operator $s_r(x)$ is used instead of a division plus floor/ceiling rounding, because they have different meaning for negative numbers, i.e. dividing the integer number -1 by 2 has the result 0, whereas $s_r(-1) = -1$.

Regarding the fact that $2d_i^j$ can become too large to be represented in B bits, we have to analyse what happens in case of an overflow: positive values too large to be represented in B bits (i.e. $2^{B-1} \leq 2d_i^j < 2^B$) appear as negative numbers which are by 2^B smaller than the actual value, whereas negative values too small to be represented in B bits (i.e. $-2^B \leq 2d_i^j < -2^{B-1}$) appear as positive numbers which are by 2^B larger than the actual value. For instance the value ± 200 appears as ∓ 56 when using a signed 8 bit representation. This means that in case of an overflow $d'^j_i = 2d_i^j \pm 2^B$ and therefore $s_r(d'^j_i) = d_i^j \pm 2^{B-1}$; however, when only the B least significant bits are of interest like in this case, adding or subtracting 2^{B-1} has the same effect because $(x + y) \bmod 2^B = (x - (2^B - y)) \bmod 2^B$ and therefore $(x + 2^{B-1}) \bmod 2^B = (x - (2^B - 2^{B-1})) \bmod 2^B = (x - 2^{B-1}) \bmod 2^B$. Thus by adding 2^{B-1} to the average value we can compensate the case when $2d_i^j$ causes an overflow in B bits. For example encoding the (unsigned) values $(200, 0)$ in 8 bits results in $c'^j_i = 100 + 128 = 228$ and $d'^j_i = -56$ (see above; note that the average coefficients have the same sign as the original data, whereas the detail coefficients are always signed), which leads to reconstruction values $c_{2i}^{j+1} = (228 + s_r(-56 + 1)) \bmod 256 = (228 - 28) \bmod 256 = 200$ and $c_{2i+1}^{j+1} = (228 - s_r(-56)) \bmod 256 = (228 + 28) \bmod 256 = 256 \bmod 256 = 0$ and we're indeed able to reconstruct the correct values $(200, 0)$ in the least significant 8 bits.

Appendix B

Wavelet Filters

This part of the appendix lists the wavelet filter coefficients h_i for the various wavelet types used in the compression engine. All filter coefficients are normalized such that $\sum h_i = \sqrt{2}$ and $\sum h_i^2 = 1$, as required to use the same coefficients for transformation and inverse transformation without additional rescaling. In addition to the filter coefficients, the sum over their absolute values is also given for each filter; this is useful for the error propagation analysis in section 3.4.1.

B.1 Daubechies Wavelets

These are the filter coefficients for Daubechies wavelets with 4–20 taps. They can be found in e.g. [18, 47].

Daubechies 4-tap	
h_0	0.4829629131445341
h_1	0.8365163037378079
h_2	0.2241438680420134
h_3	-0.1294095225512604
$\sum \lvert h_i \rvert$	1.6730326074756159

Daubechies 6-tap	
h_0	0.3326705529500825
h_1	0.8068915093110924
h_2	0.4598775021184914
h_3	-0.1350110200102546
h_4	-0.0854412738820267
h_5	0.0352262918857095
$\sum \lvert h_i \rvert$	1.8551181501576572

Daubechies 8-tap	
h_0	0.2303778133088964
h_1	0.7148465705529154
h_2	0.6308807679398587
h_3	-0.0279837694168599
h_4	-0.1870348117190931
h_5	0.0308413818355607
h_6	0.0328830116668852
h_7	-0.0105974017850690
$\sum \lvert h_i \rvert$	1.8654455282251383

Daubechies 10-tap	
h_0	0.1601023979741929
h_1	0.6038292697971895
h_2	0.7243085284377726
h_3	0.1384281459013203
h_4	-0.2422948870663823
h_5	-0.0322448695846381
h_6	0.0775714938400459
h_7	-0.0062414902127983
h_8	-0.0125807519990820
h_9	0.0033357252854738
$\sum \lvert h_i \rvert$	2.0009375600988957

Daubechies 12-tap	
h_0	0.1115407433501095
h_1	0.4946238903984533
h_2	0.7511339080210959
h_3	0.3152503517091982
h_4	-0.2262646939654400
h_5	-0.1297668675672625
h_6	0.0975016055873225
h_7	0.0275228655303053
h_8	-0.0315820393174862
h_9	0.0005538422011614
h_{10}	0.0047772575109455
h_{11}	-0.0010773010853085
$\sum \lvert h_i \rvert$	2.1915953662440892

Daubechies 14-tap	
h_0	0.0778520540850037
h_1	0.3965393194818912
h_2	0.7291320908461957
h_3	0.4697822874051889
h_4	-0.1439060039285212
h_5	-0.2240361849938412
h_6	0.0713092192668272
h_7	0.0806126091510774
h_8	-0.0380299369350104
h_9	-0.0165745416306655
h_{10}	0.0125509985560986
h_{11}	0.0004295779729214
h_{12}	-0.0018016407040473
h_{13}	0.0003537137999745
$\sum \lvert h_i \rvert$	2.2629101787572639

Daubechies 16-tap	
h_0	0.0544158422431072
h_1	0.3128715909143166
h_2	0.6756307362973195
h_3	0.5853546836542159
h_4	-0.0158291052563823
h_5	-0.2840155429615824
h_6	0.0004724845739124
h_7	0.1287474266204893
h_8	-0.0173693010018090
h_9	-0.0440882539307971
h_{10}	0.0139810279174001
h_{11}	0.0087460940474065
h_{12}	-0.0048703529934520
h_{13}	-0.0003917403733770
h_{14}	0.0006754494064506
h_{15}	-0.0001174767841248
$\sum \lvert h_i \rvert$	2.1475771089761433

Daubechies 18-tap	
h_0	0.0380779473638778
h_1	0.2438346746125858
h_2	0.6048231236900955
h_3	0.6572880780512736
h_4	0.1331973858249883
h_5	-0.2932737832791663
h_6	-0.0968407832229492
h_7	0.1485407493381256
h_8	0.0307256814793385
h_9	-0.0676328290613279
h_{10}	0.0002509471148340
h_{11}	0.0223616621236798
h_{12}	-0.0047232047577518
h_{13}	-0.0042815036824635
h_{14}	0.0018476468830563
h_{15}	0.0002303857635232
h_{16}	-0.0002519631889427
h_{17}	0.0000393473203163
$\sum \lvert h_i \rvert$	2.3482216967582961

Daubechies 20-tap	
h_0	0.0266700579005473
h_1	0.1881768000776347
h_2	0.5272011889315757
h_3	0.6884590394534363
h_4	0.2811723436605715
h_5	-0.2498464243271598
h_6	-0.1959462743772862
h_7	0.1273693403357541
h_8	0.0930573646035547
h_9	-0.0713941471663501
h_{10}	-0.0294575368218399
h_{11}	0.0332126740593612
h_{12}	0.0036065535669870
h_{13}	-0.0107331754833007
h_{14}	0.0013953517470688
h_{15}	0.0019924052951925
h_{16}	-0.0006858566949564
h_{17}	-0.0001164668551285
h_{18}	0.0000935886703202
h_{19}	-0.0000132642028945
$\sum \lvert h_i \rvert$	2.5305998542309207

B.2 Least Asymmetric Wavelets

These are the filter coefficients for the Least Asymmetric wavelets with 8–20 taps. They can be found in e.g. [18].

Least Asymmetric 8-tap	
h_0	-0.0757657147896339
h_1	-0.0296355276460688
h_2	0.4976186676266057
h_3	0.8037387518083272
h_4	0.2978577956066951
h_5	-0.0992195435773194
h_6	-0.0126039672623100
h_7	0.0322231006041961
$\sum \lvert h_i \rvert$	1.8486630689211563

Least Asymmetric 10-tap	
h_0	0.0273330683451645
h_1	0.0295194909260734
h_2	-0.0391342493025834
h_3	0.1993975339769955
h_4	0.7234076904038076
h_5	0.6339789634569490
h_6	0.0166021057644243
h_7	-0.1753280899081075
h_8	-0.0211018340249298
h_9	0.0195388827353869
$\sum \lvert h_i \rvert$	1.8853419088444219

Least Asymmetric 12-tap	
h_0	0.0154041093273377
h_1	0.0034907120843304
h_2	-0.1179901111484105
h_3	-0.0483117425859981
h_4	0.4910559419276396
h_5	0.7876411410287942
h_6	0.3379294217282401
h_7	-0.0726375227866000
h_8	-0.0210602925126954
h_9	0.0447249017707482
h_{10}	0.0017677118643983
h_{11}	-0.0078007083247650
$\sum \lvert h_i \rvert$	1.9498143170899580

Least Asymmetric 14-tap	
h_0	0.0026818145681163
h_1	-0.0010473848889657
h_2	-0.0126363034031523
h_3	0.0305155131666126
h_4	0.0678926935015956
h_5	-0.0495528349370399
h_6	0.0174412550871095
h_7	0.5361019170907605
h_8	0.7677643170045544
h_9	0.2886296317509771
h_{10}	-0.1400472404427000
h_{11}	-0.1078082377036144
h_{12}	0.0040102448717032
h_{13}	0.0102681767084966
$\sum \lvert h_i \rvert$	2.0363975651253980

Least Asymmetric 16-tap	
h_0	0.0018899503329007
h_1	-0.0003029205145516
h_2	-0.0149522583367926
h_3	0.0038087520140601
h_4	0.0491371796734768
h_5	-0.0272190299168137
h_6	-0.0519458381078751
h_7	0.3644418948359564
h_8	0.7771857516997479
h_9	0.4813596512592012
h_{10}	-0.0612733590679088
h_{11}	-0.1432942383510542
h_{12}	0.0076074873252848
h_{13}	0.0316950878103452
h_{14}	-0.0005421323316355
h_{15}	-0.0033824159513594
$\sum \lvert h_i \rvert$	2.0200379475289640

Least Asymmetric 18-tap	
h_0	0.0010694900326538
h_1	-0.0004731544985879
h_2	-0.0102640640276849
h_3	0.0088592674935117
h_4	0.0620777893027638
h_5	-0.0182337707798257
h_6	-0.1915508312964873
h_7	0.0352724880359345
h_8	0.6173384491413522
h_9	0.7178970827642257
h_{10}	0.2387609146074181
h_{11}	-0.0545689584305765
h_{12}	0.0005834627463312
h_{13}	0.0302248788579895
h_{14}	-0.0115282102079848
h_{15}	-0.0132719677815332
h_{16}	0.0006197808890549
h_{17}	0.0014009155255716
$\sum \lvert h_i \rvert$	2.0139954764194874

Least Asymmetric 20-tap	
h_0	0.0007701598091030
h_1	0.0000956326707837
h_2	-0.0086412992759401
h_3	-0.0014653825833465
h_4	0.0459272392237649
h_5	0.0116098939129724
h_6	-0.1594942788575307
h_7	-0.0708805358108615
h_8	0.4716906668426588
h_9	0.7695100370143387
h_{10}	0.3838267612253822
h_{11}	-0.0355367403054689
h_{12}	-0.0319900568281631
h_{13}	0.0499949720791560
h_{14}	0.0057649120455518
h_{15}	-0.0203549398039460
h_{16}	-0.0008043589345370
h_{17}	0.0045931735836703
h_{18}	0.0000570360843390
h_{19}	-0.0004593294205481
$\sum \lvert h_i \rvert$	2.0734674063120626

B.3 Coiflet Wavelets

These are the filter coefficients for Coiflet wavelets with 6–30 taps. They can be found in e.g. [18].

Coiflet 6-tap	
h_0	-0.0727326176845812
h_1	0.3378976539641091
h_2	0.8525719987812369
h_3	0.3848648371898933
h_4	-0.0727329632844261
h_5	-0.0156557277419281
$\sum \lvert h_i \rvert$	1.7364557986461744

Coiflet 12-tap	
h_0	0.0163873364635975
h_1	-0.0414649367819499
h_2	-0.0673725547222730
h_3	0.3861100668229387
h_4	0.8127236354492816
h_5	0.4170051844236111
h_6	-0.0764885990786583
h_7	-0.0594344186467304
h_8	0.0236801719464430
h_9	0.0056114348194203
h_{10}	-0.0018232088707114
h_{11}	-0.0007205496433577
$\sum \lvert h_i \rvert$	1.9088220976689729

Coiflet 18-tap	
h_0	-0.0037935128644910
h_1	0.0077825964273254
h_2	0.0234526961418362
h_3	-0.0657719112818552
h_4	-0.0611233900026726
h_5	0.4051769024096150
h_6	0.7937772226256168
h_7	0.4284834763776167
h_8	-0.0717998216193117
h_9	-0.0823019271068856
h_{10}	0.0345550275730615
h_{11}	0.0158805448636158
h_{12}	-0.0090079761366615
h_{13}	-0.0025745176887502
h_{14}	0.0011175187708906
h_{15}	0.0004662169601129
h_{16}	-0.0000709833031381
h_{17}	-0.0000345997728362
$\sum \lvert h_i \rvert$	2.0071708419262930

Coiflet 24-tap	
h_0	0.0008923136685824
h_1	-0.0016294920126020
h_2	-0.0073461663276432
h_3	0.0160689439647787
h_4	0.0266823001560570
h_5	-0.0812666996808907
h_6	-0.0560773133167630
h_7	0.4153084070304911
h_8	0.7822389309206136
h_9	0.4343860564915321
h_{10}	-0.0666274742634348
h_{11}	-0.0962204420340021
h_{12}	0.0393344271233433
h_{13}	0.0250822618448678
h_{14}	-0.0152117315279485
h_{15}	-0.0056582866866115
h_{16}	0.0037514361572790
h_{17}	0.0012665619292991
h_{18}	-0.0005890207562444
h_{19}	-0.0002599745524878
h_{20}	0.0000623390344610
h_{21}	0.0000312298758654
h_{22}	-0.0000032596802369
h_{23}	-0.0000017849850031
$\sum \lvert h_i \rvert$	2.0759968540210383

Coiflet 30-tap	
h_0	-0.0002120808398259
h_1	0.0003585896879330
h_2	0.0021782363583355
h_3	-0.0041593587818186
h_4	-0.0101311175209033
h_5	0.0234081567882734
h_6	0.0281680289738655
h_7	-0.0919200105692549
h_8	-0.0520431631816557
h_9	0.4215662067346896
h_{10}	0.7742896037334737
h_{11}	0.4379916262173834
h_{12}	-0.0620359639693546
h_{13}	-0.1055742087143175
h_{14}	0.0412892087544753
h_{15}	0.0326835742705106
h_{16}	-0.0197617789446276
h_{17}	-0.0091642311634348
h_{18}	0.0067641854487565
h_{19}	0.0024333732129107
h_{20}	-0.0016628637021860
h_{21}	-0.0006381313431115
h_{22}	0.0003022595818445
h_{23}	0.0001405411497166
h_{24}	-0.0000413404322768
h_{25}	-0.0000213150268122
h_{26}	0.0000037346551755
h_{27}	0.0000020637618516
h_{28}	-0.0000001674428858
h_{29}	-0.0000000951765727
$\sum \lvert h_i \rvert$	2.1289452161382330

Appendix C

Compression Parameters

This part of the appendix explains all parameters of the compression engine's dynamic parameter system described in section 3.6, ordered by module. The first column contains the parameter keywords, the second their types, the third their default values and the last a short description. Parameter strings for the engine are formed as comma-separated pairs of *`keyword`*=*`value`*, as explained in section 3.6. For example `snr=8000` activates signal-to-noise termination for zerotree quantization with a threshold value of 8000, `predintra=hyperplane,hypernorm=2` activates the hyperplane intrachannel predictor for all channels using the third dimension as hyperplane normal and `predinter=delta, intermap="0:1,2:1"` activates the delta interchannel predictor and uses the second channel to predict the first and the third.

Keyword	Type	Default	Description
`zlevel`	int	6	The *ZLib* compression level (0 . . . 9)

Figure C.1: Parameters for *ZLib* compression stream

Keyword	Type	Default	Description
`wavestr`	string	`zlib`	The compression stream name to use: `none`, `rle`, `zlib` or `arith`
`mrlevels`	int	0	Maximum number of hierarchical levels to do multiresolution analysis on. 0 means all levels
`banditer`	string	`leveldet`	The name of the band iterator. Possible values: `isolevel`, `leveldet` and `isodetail`

Figure C.2: Parameters for wavelets

Keyword	Type	Default	Description
`wqtype`	string	`zerotree`	The wavelet quantization type to use. Possible values are `perband` and `zerotree`. `zerotree` is much more efficient regarding storage space, but involves rather much overhead. `perband` is simpler but compresses worse and is harder to configure
`enhancelv`	int	0	Number of hierarchical levels (starting from coarsest one) over which to enhance wavelet-coefficients at the boundary (to avoid boundary artefacts at low rates). 0 disables it
`qwavdbg`	int	0	Level for debug-version: perform additional statistics during encoding; higher levels do everything lower levels do. 1: residual and logarithmic histogram; 2: linear histogram for all bands; 3: effect of zeroing each band on signal-to-noise ratio

Figure C.3: Parameters for lossy wavelets

Keyword	Type	Default	Description
`qrtype`	string	`const`	The quantizer type (statistics module) setting the bits per band. Values are `const`, `linear`, `expnt`, `gauss` and `custom`, where `custom` can't be selected directly (see `relqbits`)
`qntype`	string	`linear`	The quantization type, either `linear` or `expnt`
`bitscale`	double	1.0	Set the scaling factor of the bit allocation curve relative to the size of the base type
`cutoff`	double	1.0	The band number relative to the total number of wavelet bands starting from which all bands are ignored (quantized to 0)
`nullzone`	double	0.0	Coefficients whose absolute value is smaller than `nullzone` are set to 0
`relqbits`	string	—	The string is a comma-separated ($\Rightarrow$ the string must be enclosed in double quotes) list of floating point values, encoding the number of bits relative to the number of bits in the base type each wavelet band should be encoded with. Implicitly sets `qrtype` to `custom`

Figure C.4: Parameters for *homogeneous* lossy wavelet quantization

Keyword	Type	Default	Description
`predinter`	string	—	The name of the interchannel predictor which correlates cells across channels (→ atomic types within structured types). Possible values: `delta` (normal difference) and `scaledelta` (rescale to same dynamic range, then use difference)
`predintra`	string	—	The name of the intrachannel predictor which correlates cells with their spatial neighbours. Possible values: `hyperplane` (predict from previous hyperplane, e.g. previous scanline in a 2D image), `neighbours` (predict from arithmetic average of all neighbouring cells already seen) and `weighted` (additionally weigh the neighbouring cell values according to their offset in each dimension)
`intermap`	string	—	Comma-separated list of channel mappings of the form *ch-num*`:`*pred-num*, where *ch-num* is the number of the channel being mapped and *pred-num* is the number of the channel it's predicted by. e.g. typical for RGB images, where green predicts red and blue: `0:1,2:1`
`intralist`	string	!	Comma-separated list of channel numbers where intrachannel prediction is used. If the first character is an exclamation mark, the inverse list is used
`hypernorm`	int	0	For intrachannel `hyperplane` prediction: the number of the dimension orthogonal to the (moving) predictor hyperplane, i.e. the direction in which prediction takes place
`predweight`	double	0.5	For intrachannel `weighted` prediction: the amount by which a cell weight is multiplied for each offset $\neq 0$ in a dimension, starting with a weight of 1

Figure C.5: Parameters for predictors

Keyword	Type	Default	Description
`zttype`	string	`band1`	The type of the zerotree coder to use. `band1` is one-pass with a four symbol alphabet, whereas `band2` is two-pass with dominant pass (four symbol alphabet) and subordinate pass (two symbol alphabet)
`snr`	double	1e4	The signal-to-noise ratio to use for the encoding of wavelet coefficients. Higher values mean better quality
`psnr`	double	0	The peak signal-to-noise ratio to use for the encoding of wavelet coefficients. Higher values mean better quality, 0 disables it
`residuum`	double	0	The maximum residual per cell to use during encoding. Lower (positive) values mean better quality, 0 disables it

Figure C.6: Parameters for *zerotree* lossy wavelet quantization

Keyword	Type	Default	Description
`exactformat`	int	0	If `exactformat` is 0, the transfer format will only be used by the server for transfer compression if the data was stored uncompressed in the database; it it's $\neq 0$, the server will always repack the data to the exact format requested by the client

Figure C.7: Parameters for client-server communication