The zerotree compression algorithm

Institute for Perception Research
PO Box 513, 5600 MB Eindhoven

28.10.1996

Rapport no. 1129

The zerotree compression algorithm

M.L.A. Bierman

Voor akkoord: Dr.ir. J.B.O.S. Martens

[illegible]

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Institute for Perception Research (IPO)

PROJECT REPORT

The Zerotree Compression Algorithm

by

M.L.A. Bierman (344953)

Supervisor : drs. A.M. van Dijk (IPO)

September 1996

Abstract

At the Institute for Perception Research (IPO) still images can be processed using Khoros. This report is created to show all the results obtained from implementing an image compression algorithm in Khoros. The algorithm that was implemented is called 'The Zerotree Compression Algorithm'. Normally, the algorithm uses hierarchical decompositions, but it has been altered to work with both polar and cartesian decompositions. The way it works and the implementation in Khoros is described thoroughly. Unfortunately, there was no time left to do some extensive testing for different input images and filtering types. Since the algorithm seems very promising, it is recommended to do more research on this algorithm and possible variations.

The project itself has been done in accordance with my study 'Electrical Engineering' at the Eindhoven University of Technology. It was carried out from March 1996 till October 1996 at IPO.

Contents

Chapter 1

Introduction

At the Institute for Perception Research (IPO) algorithms are designed for image processing. A part of this processing is about compressing images. Why would one want to compress images, you might ask? Well, because of the enormous amount of data involved with high quality pictures, they tend to be so space consuming that they overload computer systems everywhere. The most important answer to this space consumption is compression. Compressed images can be created and regained by a (de)coding program, which operates according to a specific scheme. In the course of time many different schemes have been developed and tested.

In 1993, a coding scheme called 'The Zerotree Compression Algorithm', was published ([Sha93]). Since it seemed to work so well, it was decided that the algorithm had to be implemented in Khoros, so that it could be used at IPO. Khoros is a software library with a number of programs that enables users to easily process images.
Here is where I come into account. I accepted to implement the algorithm in Khoros so that it could be used as a reference to their own research and to gain more insight in how to obtain highly compressed images with a quality that is still sufficient. To gain more insight in the algorithm, I will extensively describe it in Chapter 2.

In the paper that was published, one can find a description of a fully embedded coder, which decomposes images into local coefficients, codes these coefficients into a bytestream and compresses the bytestream arithmetically. Since at IPO Khoros is in use, there already are a number of programs that are, for instance, able to decompose images in (local) coefficients. Arithmetical compression is also implemented in Khoros, so it only was necessary to produce a program that could translate the decomposed image to a bytestream. The way the algorithm is implemented, can be read in Chapter 3. The C-code of the program is also added (Appendix B).

Since the paper is not too clear about some aspects of the coder, there are some differences between the coder I implemented and the one in the paper. In Chapter 4 these differences and the comparison between my results and the ones in the paper are described. This leads to some conclusions of course, which can be found in Chapter 5 together with a few recommendations for further research. While working at IPO, I could not help but notice the pleasant working atmosphere of which, among others, something will be told in Chapter 6.

Chapter 2

The zerotree algorithm

In 1993, J.M. Shapiro published a compression algorithm for single images, called 'the zerotree algorithm' ([Sha93]). The paper he published, also contains a description of the fully embedded coder, he developed. This coder has certain properties I will describe and obtains high compression ratios.

2.1 The embedded coder

The coder, described in [Sha93], consists of a number of operations which are applied after each other. This can be seen in figure 2.1

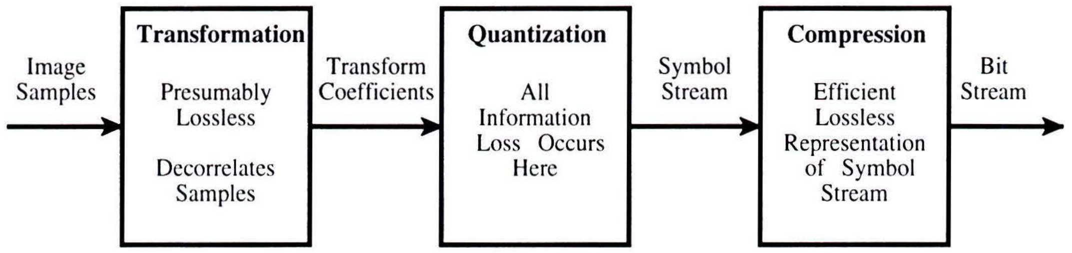


Figure 2.1: Basic components of a low-bit rate coder.

To create a low-bit rate coder, three basic components are required: a transformation, a quantizer and lossless data compression. These are the components shown in figure 2.1 and described in the next three sections.

2.1.1 The transformation

The goal of the transformation is to produce coefficients that are decorrelated. When this transformation is performed, one obtains an image decomposition, which is an array of coefficients that contains the same amount of information as was present in the original image. The decorrelation occurs, because the transformation removes statistical redundancies.

The transformation which was originally intended to be used with the zerotree algorithm is identical to a hierarchical subband system, where the subbands are logarithmically spaced in frequency and represent an octave-band decomposition (see also chapter 14 of [RJ91]). To obtain a hierarchical decomposition, first the image has to be divided in four subbands which are critically subsampled and second the subband with the low frequencies is again subdivided in four subbands according to the first step. If you repeat these steps three times for instance, you might get something like figure 2.2 when you would use the 'Lena' picture as can be seen in Appendix A as the original image.

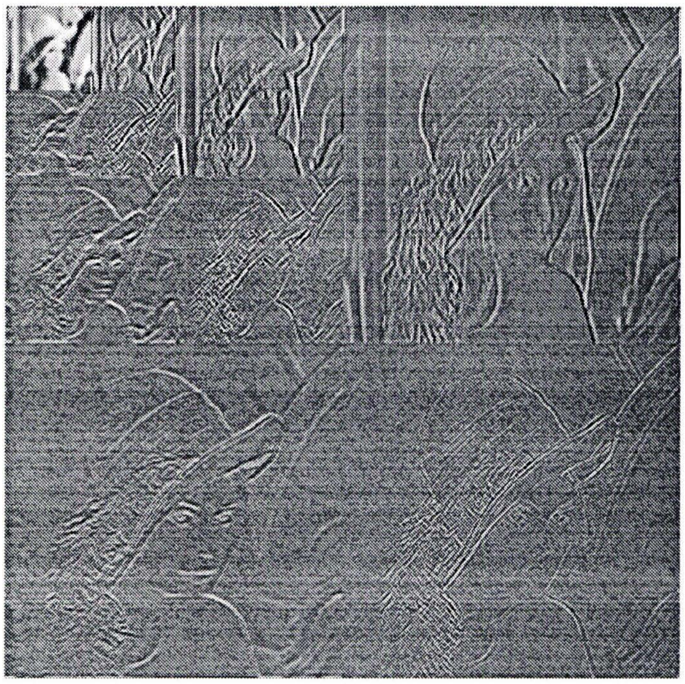


Figure 2.2: A 3^{th} order hierarchical decomposition of the 'Lena' original.

It is obvious that the lowest frequency subband contains wavelet coefficients that represent information on the coarsest scale. The design of filters to obtain decompositions as mentioned before, has been discussed by many authors and will therefore not be treated here.

It is assumed that the transform is lossless as opposed to the quantization, which is lossy. Why this is the fact and how, can be read in the next section.

2.1.2 The quantization

When the quantization is started, the zerotree algorithm comes into account. Essentially, the zerotree is a data structure that is defined to represent certain amounts of data. The idea to use zerotrees is based on the hypothesis that if a coefficient at a coarse scale is insignificant with respect to a given threshold T, then all coefficients of the same orientation in the same spatial location at finer scales are likely to be insignificant, too. So what we have now, is that coefficients at a finer scale are related to coefficients at a coarser scale. The coefficient at the coarse scale will be called a *parent* and the ones at finer scales are called *children*. When looking at a certain parent, the set of coefficients at all finer scales of similar orientation can be called *descendants*. The above statements can be used to define so-called *Parent-Child dependencies*. These dependencies describe in what way the subbands are related to each other (see Section 3.2). To make it more clear, the Parent-Child dependencies for a hierarchical decomposition are plotted in figure 2.3.

To detect the zerotrees in a decomposition, we need an algorithm or a scheme according to which can be searched through all the wavelet coefficients. To gain more insight in the algorithm a flowchart is presented in figure 2.4 according to which the search is done.

First you start with the top-left subband, which is the one with the lowest frequency components. For every coefficient you have to decide whether it is significant (POS/NEG) (the absolute value should be equal or greater than the current threshold) and if not, what kind of zero it is: a zerotree root (ZTR), a descendant of a zerotree root (dZTR) or an isolated zero (IZ). The way you have to code these symbols to get a bytestream is described in the next section.

The flowchart shows us how the algorithm finds the location of important coefficients and

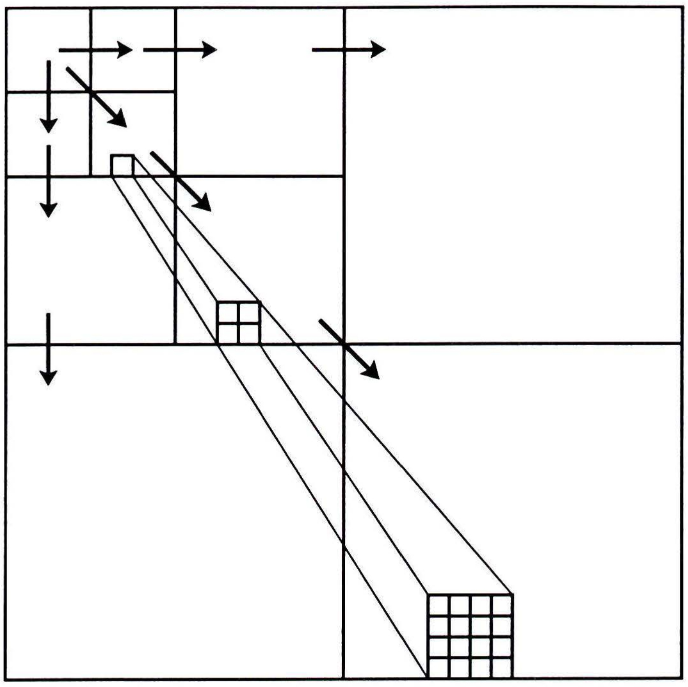


Figure 2.3: The hierarchical Parent-Child relations.

retains them in a space-saving way. Doing this for one specific threshold is called the *Dominant Pass*. After a Dominant pass, a *Subordinate Pass* is performed. This Subordinate Pass takes care of the accuracy of the representation of the coefficients. Every time a Subordinate Pass is started, it will write the next less significant bit of the coefficients that were already found during the Dominant Pass This means that, starting with a Dominant Pass, the passes are alternated until a certain condition is met. This condition can be anything: a bit-rate, a quality factor or a signal-to-noise ratio, for instance.

2.1.3 Lossless data compression

In the introduction it is described why one would need compression. In this case lossless compression is necessary, because the lossy compression occurs in the quantizer part. By getting rid of less important information, an amount of information remains that is much smaller in size. This remainder has to be stored correctly, otherwise strange artifacts might occur. The correct storage of the remaining information in as small a space as possible, can be done with lossless data compression. In the preceding section there was a first step towards this compression. The symbols that are necessary to code the so-called significance map are already mentioned: POS, NEG, ZTR and IZ. The symbol dZTR is not coded, but is used to determine all descendants, as will be explained in Chapter 3. Now four symbols remain. If they occur in equivalent amounts, it is obvious to choose for 2-bit codes. According to empirical evidence, this is almost the case, so the choice is easy. It appears that the zerotree root symbol occurs more often, but when you choose a 1-bit code for ZTR, it means that at least two of the remaining three symbols will have 3-bit codes. This increase in bits is so high that it is unwise to do so. The final result can be found in Table 2.1.

When it is necessary to code a number of coefficients in the highest frequency bands, there is obviously no way that a zerotree root or an isolated zero can be detected, because there are no descendants. So, further optimisation can be achieved by choosing a zero (Z) symbol, to be coded with an '0' for the sole use of coding insignificant coefficients in the highest frequency bands.

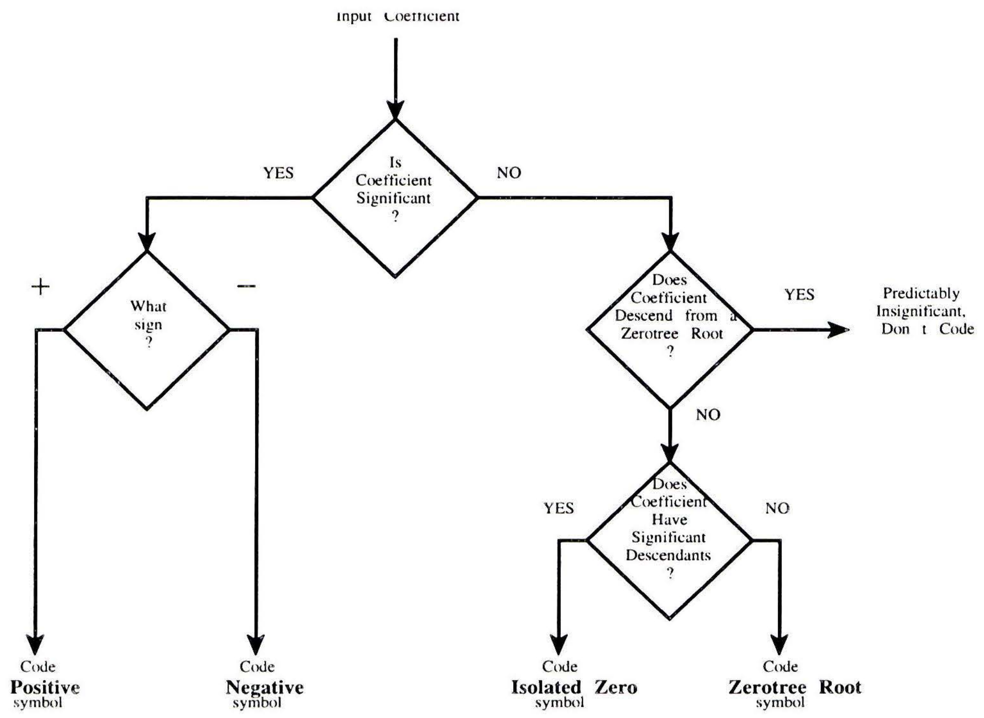

Input Coefficient
Is Coefficient Significant ?
YES
NO
What sign ?
+
−
Does Coefficient Descend from a Zerotree Root ?
YES
Predictably Insignificant, Don t Code
NO
Does Coefficient Have Significant Descendants ?
YES
NO
Code **Positive** symbol
Code **Negative** symbol
Code **Isolated Zero** symbol
Code **Zerotree Root** symbol

Figure 2.4: A flowchart representing the 'Dominant Pass' of the algorithm.

Symbol	POS	NEG	ZTR	IZ
Code	11	10	00	01

Table 2.1: The translation of coded symbols to bits.

To create an even smaller output bytestream, it is possible to code the symbolstream arithmetically. About this subject many articles are written, for instance [IW87].

2.2 The implemented (de)coder

As already mentioned, at IPO Khoros is being used to do image processing. Some of the operations that are necessary in the embedded coder, are also available from the Khoros library. A number of these programs are developed at IPO, so they might not be available to everyone. In the following three sections I will describe what can and cannot be done with the currently available programs and how I solved the discrepancies that occurred.

2.2.1 The transformation

The transform as mentioned in section 2.1.1 cannot be performed by the current software at IPO. There are quite a number of transforms available, but none of a hierarchical form. Time would run too short to fully implement all the necessary software myself, so it was decided to implement the zerotree algorithm for standard cartesian and polar decompositions with cartesian ordering of the coefficients. To take a look at an example of those kinds of decompositions, see figure 2.5.

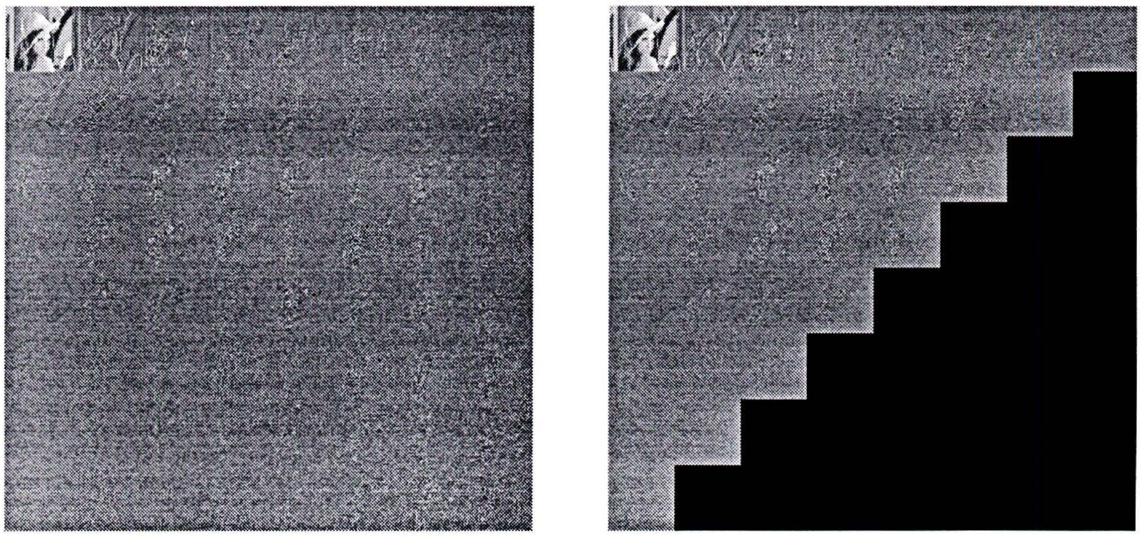


Figure 2.5: Two typical decompositions. (left: cartesian and right: polar)

The pictures in figure 2.5 are produced with the standard software available at IPO. This also means that no further research has to be done in this direction, in this case. The contents of figure 2.2 were created with Cantata, which is a GUI that can be used to easily interconnect and use the different kinds of programs available from the Khoros software library. As can be seen in figure 2.6 it is quite an elaborate job to produce this kind of decompositions with the currently available software.

Synthesizing the original image from a hierarchical decomposition is even more elaborate with this software, so I will not discuss it here.

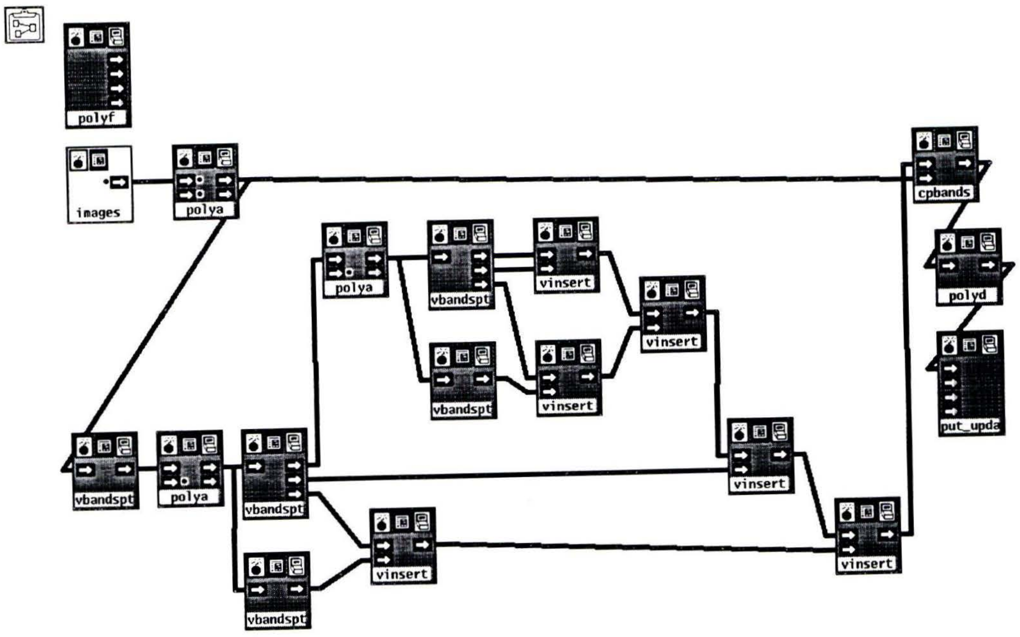


Figure 2.6: A Cantata workspace that produces 3^{th} order hierarchical decompositions.

2.2.2 The quantization

As for the quantization, the other kinds of decompositions mean that there has to be a different way of defining the Parent-Child dependencies, because the hierarchical relations do not fit here. A possible definition of the relations is shown in figure 2.7.

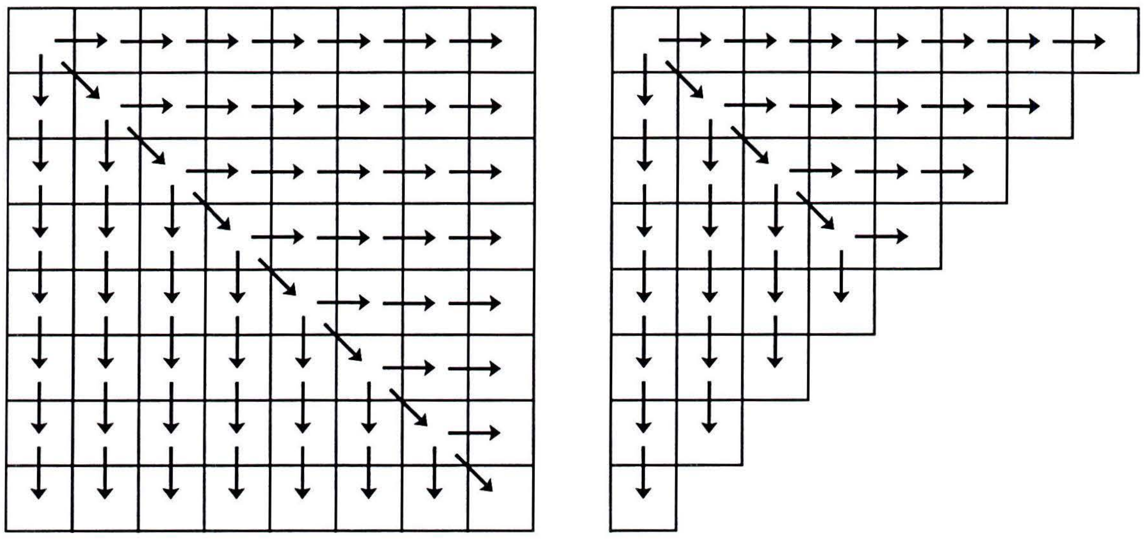


Figure 2.7: Possible 'Parent-Child' relations for cartesian and polar decompositions.

Another goal in this project was to develop an easy method of defining the Parent-Child dependencies. For certain kinds of research here at IPO it is necessary to be able to omit some subbands from coding. So a flexible way of defining the dependencies in an input file was developed as described in section 3.2.

The way the zerotree algorithm behaves, does not change for these kinds of decompositions, but one disadvantage is very clear. The number of descendants of a certain parent will only grow linearly instead of exponentially when looking down along the dependencies at higher frequency components. This might be quite inefficient compared to quantizing hierarchical decompositions.

2.2.3 Lossless data compression

The representation of the symbols which was chosen in section 2.1.3, does not need any changes and is used as is in the developed software that can be found in Appendix B.

In Khoros, a number of arithmetic coders is available, so it is not necessary to do any research on this. The only problem is, that in the paper the arithmetic coder is integrated into the bytestream conversion, which is not possible here, because it is a separate program in Khoros. This puts some constraints to the options that are to be passed to the program. The resulting constraints can be found in section 3.1.

2.3 An example

To gain more insight in the way the algorithm behaves, an example will be shown. Since the transformation and the arithmetic coding are not within the boundaries of this project, only the quantization will be demonstrated. We start with a part of a hierarchical transform (as shown in figure 2.8) of the 'Lena' original (see Appendix A).

63	-34	49	10	7	13	-12	7
-31	23	14	-13	3	4	6	-1
15	14	3	-12	5	-7	3	9
-9	-7	-14	8	4	-2	3	2
-5	9	-1	47	4	6	-2	2
3	0	-3	2	3	-2	0	4
2	-3	6	-4	3	6	3	6
5	11	5	6	0	3	-4	4

Figure 2.8: A part of a hierarchical decomposition.

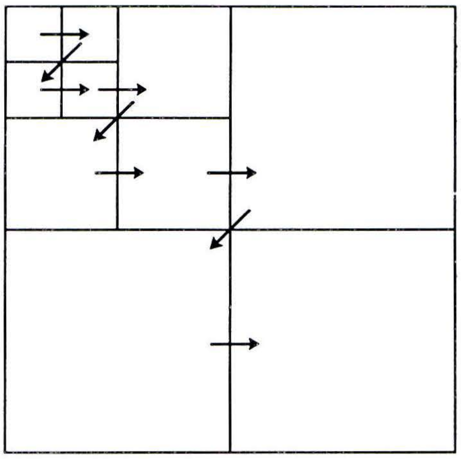


Figure 2.9: The order of importance of the subbands.

A little intermezzo as for the initial threshold: this threshold is the largest power of two that is still smaller than the largest magnitude in the set of coefficients. For example, in the above set of coefficients 63 is the largest magnitude available. The initial threshold will be $T_{ini} = 32$.

2.3.1 The first Dominant Pass

Now, we can start scanning through the decomposition in order of importance. This order is expressed in figure 2.9. We start with 63, which is significant and positive, so a POS symbol is coded. Next, -34 will be coded as a NEG symbol. The coefficient -31 is not significant, but one of its descendants is significant (47) so in this case a IZ symbol will be coded. Now we arrive at coefficient 23. As can be seen, none of the descendants is significant in respect to our current threshold. A ZTR symbol is generated, so that none of the descendants will require further coding in this pass. This coding can be continued, until all coefficients can be reconstructed when using a decoder. When reconstructing the coefficients, it is only sure that when a POS or NEG symbol occurs, the original coefficient had a magnitude in the range of [32,63]. It was chosen to reconstruct with a magnitude that represents the middle of the interval, in this case 48. For the first Dominant Pass this leads to: see Table 2.2.

Coefficient Value	Symbol	Coded Bit(s)	Reconstruction Value	Coefficient Value	Symbol	Coded Bit(s)	Reconstruction Value
63	POS	11	48	-9	ZTR	00	0
-34	NEG	10	-48	-7	ZTR	00	0
-31	IZ	01	0	7	Z	0	0
23	ZTR	00	0	13	Z	0	0
49	POS	11	48	3	Z	0	0
10	ZTR	00	0	4	Z	0	0
14	ZTR	00	0	-1	Z	0	0
-13	ZTR	00	0	47	POS	11	48
15	ZTR	00	0	-3	Z	0	0
14	IZ	01	0	2	Z	0	0

Table 2.2: Code generated by the first Dominant Pass.

2.3.2 The first Subordinate Pass

Next a Subordinate Pass is performed. For all significant coefficients it has to be decided whether their magnitude is in the interval [32,48) or in the interval [48,64). The upper interval will be coded with a '1' and the lower interval with a '0'. This results in: see Table 2.3 where 32 is subtracted, so that we only have to detect whether the coefficient magnitude is greater than or equal to 16 or smaller. Why this is more convenient will be made obvious in the next Subordinate Pass(es).

Coefficient Magnitude	Coded Bit	Sign	Reconstruction Value
63-32=31	1	POS	56
34-32=2	0	NEG	-40
49-32=17	1	POS	56
47-32=15	0	POS	40

Table 2.3: Code generated by the first Subordinate Pass.

As can be seen in Table 2.3 the reconstructed values are subdivided in two different magnitude levels, 40 and 56. These values represent the middle of the two intervals [32,48) and [48,64). The reconstruction of the coefficients 47 and 49 gets a lot worse, but in large decompositions these differences will be equally distributed.

2.3.3 Remaining Dominant Passes

All the remaining Dominant Passes are performed in a similar way as the first one. There is only one important difference. The coefficients that already have been found are treated as zero in the next Dominant Passes. This can be seen in figure 2.10 where all found coefficients are represented with 0^*.

0^*	0^*	0^*	10	7	13	-12	7
-31	23	14	-13	3	4	6	-1
15	14	3	-12	5	-7	3	9
-9	-7	-14	8	4	-2	3	2
-5	9	-1	0^*	4	6	-2	2
3	0	-3	2	3	-2	0	4
2	-3	6	-4	3	6	3	6
5	11	5	6	0	3	-4	4

Figure 2.10: The decomposition after the first Dominant Pass.

When the next Dominant Pass starts, the threshold has to be altered to the next level of precision. In this case that would mean that $T = 16$. The results are shown in Table 2.4.

Coefficient Value	Symbol	Coded Bit(s)	Reconstruction Value	Coefficient Value	Symbol	Coded Bit(s)	Reconstruction Value
0*	IZ	01	0	-9	ZTR	00	0
0*	ZTR	00	0	-7	ZTR	00	0
-31	NEG	10	-24	3	ZTR	00	0
23	POS	11	24	-12	ZTR	00	0
15	ZTR	00	0	-14	ZTR	00	0
14	ZTR	00	0	8	ZTR	00	0

Table 2.4: Code generated by the second Dominant Pass.

2.3.4 Remaining Subordinate Passes

The subordinate passes are always performed the same. All previously found coefficients receive an extra bit for higher precision. This means that they will be subdivided in the intervals [16,24), [24,32), [32,40), [40,48), [48,56) and [56,64). This can be achieved, by subtracting the current threshold value from the coefficients that are already known, when their magnitude is greater than that of the current threshold. The resulting magnitudes only have to be compared to the new threshold, which would be $T = 8$. This results in: see Table 2.5.

Coefficient Magnitude	Coded Bit	Sign	Reconstruction Value
63-32-16=15	1	POS	60
34-32=2	0	NEG	-36
49-32-16=1	0	POS	52
47-32=15	1	POS	44
31-16=15	1	POS	28
23-16=7	0	POS	20

Table 2.5: Code generated by the second Subordinate Pass.

This is for as far as the example will be discussed. In the next chapter the translation of the algorithm to data structures and code will be described.

Chapter 3

Implementation in the Khoros system

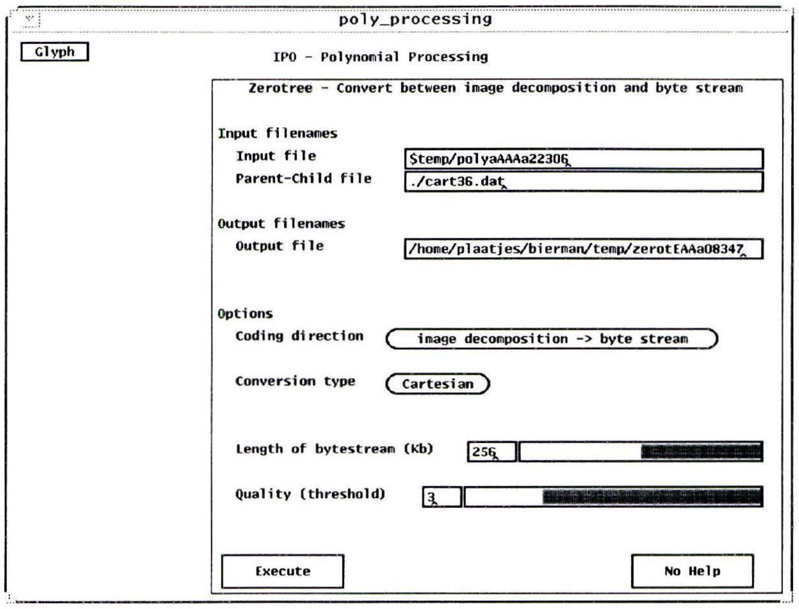
As already mentioned, the most important goal of this project is to implement the zerotree compression algorithm in Khoros. This means that programs need to be written in C and that certain extra information needs to be added. The input and output options are defined in a so-called . pane-file (see Appendix C) which describes the interface that can be used in Cantata. Cantata is a GUI (Graphic User Interface) which can be used to interconnect programs from the Khoros library and execute them in an easy way. Because the implementation is not of an embedded type like the one developed by J.M. Shapiro, some differences are inherent and will also be described in this chapter.

3.1 Global properties

When talking about global properties, one could think of the different attributes that can be used to start the program and what these attributes mean. Looking at the . pane-file, where all the possible attributes are shown, might not be to clarifying. The interface that is prescribed by this . pane-file, looks like what can be seen in figure 3.1.

Some of the attributes, as shown above, might be clear to everyone at first sight, but others might need some clarification. On the next list, all the items will be mentioned and described.

- *The 'Input file'*: The filename of a nonquantized image decomposition when coding and the filename of a coded bytestream when decoding.
- *The 'Parent-Child file'*: The filename of a description of the Parent-Child dependencies, which will be described more extensively in the next section.
- *The 'Output file'*: Almost the opposite of the 'Input file', which is the filename of a coded bytestream when coding and the filename of a (non)quantized image decomposition.
- *The 'Coding direction'*: The choice between coding (decomposition $\rightarrow$ bytestream) and decoding (bytestream $\rightarrow$ decomposition).
- *The 'Conversion type'*: The type of input (or output) decomposition, where you can choose between 'Polar', 'Cartesian' and 'Hierarchical'.
- *The 'Length of bytestream (Kb)'*: The length of the uncompressed bytestream in kilobytes, to be used when coding/decoding when lossy compression should be obtained.
- *The 'Quality (threshold)'*: A number that prescribes the number of alternating Dominant and Subordinate Passes. '1' stands for *maximum* quality (code till threshold is two). The highest number possible depends on the maximum size of the coefficients in the decomposition.



IPO - Polynomial Processing

Zerotree - Convert between image decomposition and byte stream

Input filenames
- Input file: $temp/polyaAAAa22306
- Parent-Child file: ./cart36.dat

Output filenames
- Output file: /home/plaatjes/bierman/temp/zerotEAAa08347

Options
- Coding direction: image decomposition -> byte stream
- Conversion type: Cartesian
- Length of bytestream (Kb): 256
- Quality (threshold): 3

Figure 3.1: The interface as can be seen when using Cantata.

The items that still are not clear enough, will be described even more extensively in the rest of this chapter.

3.2 Parent-Child relations

As already mentioned, the description of the Parent-Child relations has to be as flexible as possible. For certain research at IPO it is necessary to be able to exclude some subbands from coding. To fully embed these possibilities in the code would be to elaborate, so it was decided that loose files should be used. In such a file the relations between the subbands are represented in a very straight forward way.

Since coding hierarchical decompositions is not possible with the current version of the coder and since hierarchical decompositions are hard to process with the current Khoros tools, the only Parent-Child relations that are necessary are for polar and cartesian decompositions. Within these decompositions the subbands are all of the same size, so a zerotree will only grow linearly instead of exponentially like in hierarchical decompositions. Most possibly this will yield lower performance in compression ratios. This is also mentioned in [Sha93], but it is also mentioned that it could still outperform JPEG-coding (for JPEG-coding see: [RJ91] and [ea93]). When coding, we think of the subbands arranged as can be seen in figure 3.2

When we have a cartesian decomposition with 9 subbands and the kind of Parent-Child relations as can be seen in figure 2.7, a way of describing the relations easily is memorizing all the first-degree descendants of a subband. Since the first subband (zero) always has to be coded, one could use zero as a reference for 'no (first-degree) descendants'. When using one line for each subband in the file, it is not necessary to add the numbers of the subbands to the file, because these are inherently represented by the line numbers. For the 9 subband example this would result in:

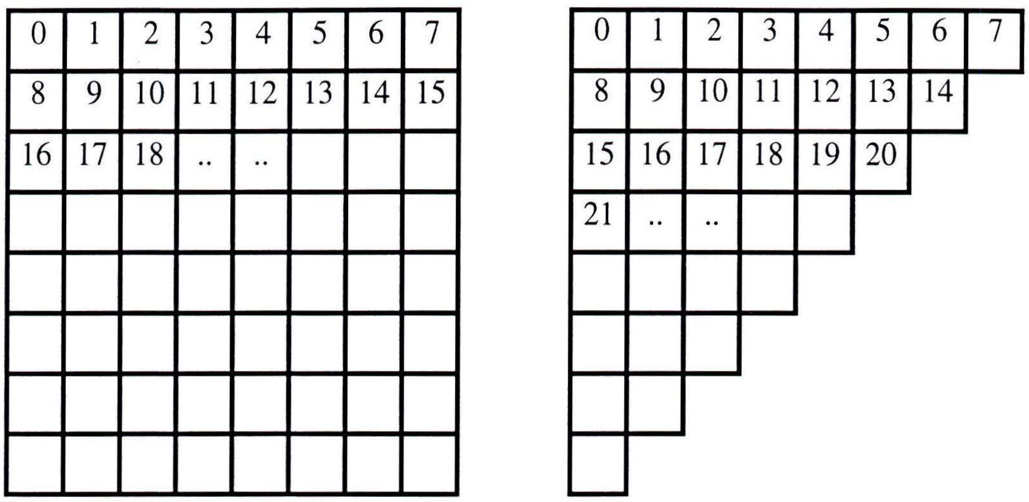
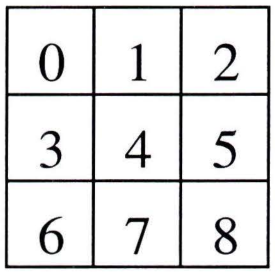


Figure 3.2: Numerical arrangement of the subbands in cartesian and polar decompositions.

```
1 3 4
2
0
6
5 7 8
0
0
0
0
```

↔



Each line is separated by a hard return, and the numbers on a line are separated by a space. In the code there is a procedure that is used to read this file and puts it in a two-dimensional array. To leave some room for different experiments, it was decided that the array should be 100 by 8, so this provides room for a maximum of eight (first-degree) descendants per subband and a maximum of hundred subbands. This is equivalent to a 9^{th} order cartesian decomposition (100 subbands), a 12^{th} order polar decomposition (91 subbands) or a 33^{rd} order hierarchical decomposition (100 subbands). A few 'Parent-Child files' are already made. They are called, for instance 'CART64.DAT' or 'POLAR36.DAT' with the first letters signifying the kind of decomposition and the number signifying the total number of subbands. As yet, there is no check implemented to verify whether the Parent-Child file fits the input(or output) decomposition. This may cause the program to 'hang' when it is not the case.

Furthermore it is possible to exclude subbands from being coded. When a subband is denoted with a zero and there are no references to it, it automatically will not be coded.

3.3 Translation of the algorithm

The most important part of the algorithm is coding and finding zerotrees, as shown in the flowchart in figure 2.4, which is the Dominant Pass. The Subordinate Pass only is about coding the next less significant bit of the magnitude of every coefficient that was found significant.

When searching for significant descendants, it is necessary to have the complete decomposition in a memory buffer. For a standard 512x512 grayscale image which yields an integer decomposition, this already means a memory consumption of 0.5 megabyte. Since it is necessary to change the coefficients that are found to zero in the original decomposition, while they still have to be remembered, another buffer of equivalent size should be used. In addition to this another buffer should be used to store intermediate results while searching for zerotrees. This means that, together with the other variables and the code itself, there has to be about 2 megabyte of free RAM to run this program. This is no problem with the computers currently in use at IPO.

Using three buffers of equivalent size has got one major advantage; the coordinates are inherently represented by the entry numbers of the arrays/buffers. The buffer names are as follows:

- *im_buffer*: The buffer that contains the original image decomposition.
- *el_buffer*: The buffer that contains the elements/coefficients that are already found to be significant.
- *coding_buffer*: The buffer that contains intermediate information about the coefficients, found while (de)coding.

The first two buffers speak for themselves, but the last one needs some additional explanation.

While coding, it is necessary to know which elements were found in previous Dominant Passes and which are found in the current Dominant Pass. To do this, the second bit of the corresponding element in the coding_buffer is set when a significant element is found. So, a zerotree structure can be created for the current significant elements. After that, all the elements with the second bit set, receive a reset of the second bit to zero and a set of the first bit (the least significant bit in the coding_buffer in this case). Now all that remains is representing zeros and zerotree roots. This can easily be done with one of the remaining bits. Here, bit number four is used for that purpose. Of course, it would have been possible to use other bits for registering all the mentioned information.

About the 'Conversion type' (see Section 3.1), there is no real difference between polar and cartesian decompositions to the program. They both consist of subbands that are all of the same resolution and are stored linearly after each other in a file. Since the ordering of the subbands and their relations are fully described in the Parent-Child file, there is nothing else that the program needs to know.

There is one thing that has to be mentioned, though. The 'Length of bytestream (Kb)' and the 'Quality (threshold)' can interfere with each other. It is possible that one of these conditions is met, while the other still is not. Most of the time, it will be so that experiments will be done with one option, while the other one remains at maximum position. This means 'Length of bytestream (Kb)' at 512 or 'Quality (threshold)' at 1. This also makes it possible to acquire a high quality coded bytestream in one run, so that it does not have to be reacquired, when doing different kinds of decoding experiments.

3.4 Differences

Differences between the implemented coder and the embedded type published in [Sha93] are inherent, as mentioned before. The embedded coder has switches to stop coding when a certain Signal-to-Noise Ratio is obtained or when a certain bit rate is achieved. The last switch is almost impossible to implement in the coder at IPO, because it is hardly predictable what rate of compression will be achieved with the arithmetic coder. As for the first switch, this one is implementable but has not been taken care of, because of lack of time.

Furthermore, in the paper ([Sha93]) nothing is said about the DC-coefficients (the elements in subband zero). Normally, these are in the range of [0,255]. Because the algorithm works for positive as well as negative coefficients, it was obviously better to translate all the DC-coefficients down with an amount of 128, obtaining elements in the range of [-128,127]. Experiments have shown that image quality is much better for bytestreams with the same bit rate. DPCM-coding (see [RJ91]) has also been tried, but this requires that all the DC-coefficients are decoded lossless, which is not possible with this lossy compression scheme.
When looking at the results in the paper it is not possible that nothing special is done with the coefficients of subband zero. Something is obviously missing there.

Chapter 4

Results

As said before, a lack of time has caused that it was not possible to do some thorough testing of the developed software. There are some minor problems that will be pointed out in this chapter. And because there were a few things missing in the paper, some obvious differences can be described.

4.1 Khoros implementation

There is a working version of the coder right now. Because of lack of time, almost no comparative tests have been done. This was mainly due to the fact that my C-programming skills were not that good when I started with this project and to the fact that part of the algorithm was misinterpreted at first. There still is a minor problem in the coder. It has got something to do with the representation of the Subordinate Pass.

As mentioned in section 2.3.2, the reconstruction of coefficients depends on the intervals to whom they belong. The most significant bit of a coefficient found to be significant, is represented by the coded symbols found in the preceding Dominant Pass. First it was assumed that Dominant Passes only produce code representing spatial information, but because of the inclusion of the sign of significant elements it also sort of represents the most significant bit. This means that the Subordinate Pass has to code the next bit of a coefficient. Since this was the part that was misinterpreted, the coder stored the most significant bit twice. When decoding, this was obviously visible. The error image produced, while comparing the original with the decoded image, shows strong blocking effects in smooth transitions (see figure 4.1).

This is due to the fact that coefficients with the second most significant bit equal to zero are strongly influenced in magnitude, because here it will be one all the time when decoding.
This problem is not properly solved yet. The procedure *Create_Sub_Code* has to be altered to produce the required results. It can easily be checked by feeding a number to the algorithm and see how it is coded. Decoding, which is done in the procedure *Retrieve_Sub_Code*, should be able to retrieve the correct number, which could be quantized. As can be seen, this is not the case, yet. Again, lack of time stops me from doing more research on this problem.

The procedures that do the zerotree coding have already been tested thoroughly by doing some extensive calculations by hand. These calculations have been compared with output from all the procedures involved, and were found to be correct. Of course, a test pattern was used, which inherited all different kinds of combinations of coefficients. Unfortunately, human errors are still possible and there was no time left to double check my calculations so a 100% guarantee for bug-free code for this part would not be appropriate. The test pattern consisted of 9 subbands, each with 3x3 resolution, representing a 2^{nd} order cartesian decomposition.

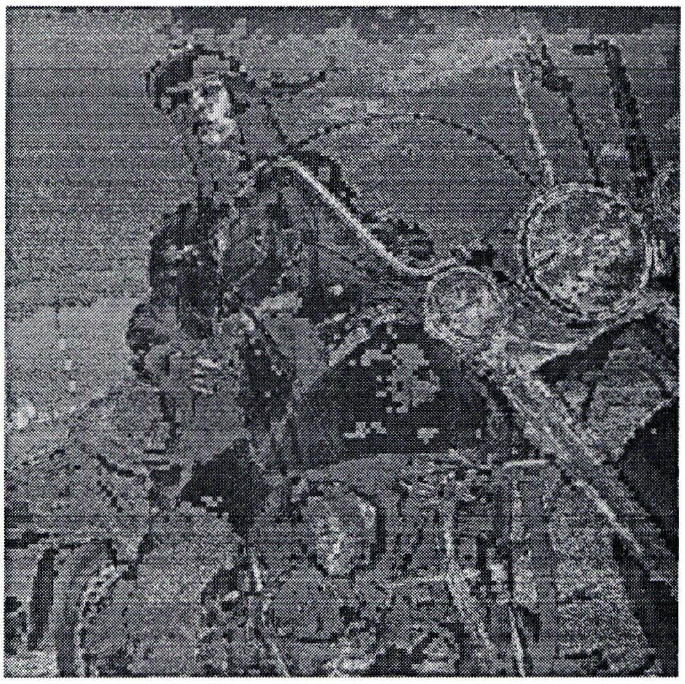


Figure 4.1: The error image produced from comparing the 'Bikerlady' original with the coded version, scaling up the error by a factor 2.

All remaining procedures, of which most are necessary for retrieving the information incorporated in the switches and settings, are also tested. None are producing faulty in- or outputs anymore, although some minor changes had to be made to the initial code to streamline the communication between procedures.

4.2 Comparison with Shapiro's paper

As already told in section 3.4, a big difference in decoded image quality is possibly due to the fact that the coefficients of subband zero are treated in a different way. Possibly, the image quality in the paper is higher, because the way of decomposing an image is different. From some of the experiments done, I learned that the decomposition method has a large influence on compression ratios. When testing the zerotree algorithm, a few different decomposition methods were used. For instance, Hermite transforms (see [vDM97]) and standard DCT (see, for instance [RJ91]) in both polar and cartesian decompositions. Both the order and window length have great influence. Standard DCT seems more promising to me, because child coefficients seem more correlated to the magnitude of their parents, but the number of tests performed is too small to conclude that DCT is better. Maybe other kinds of decomposition methods will prove to be even better than the ones mentioned.

Another important difference is, that I do not use hierarchical decompositions. Therefore, the amount of important parents is much larger. For instance, a 7^{th} order DCT decomposition has a subband zero composed of 64x64 bytes = 4096 bytes = 4 Kb. This means that every Dominant Pass requires 4096/4 = 1024 bytes, just for representing subband zero. Obviously, this requires that decompositions should only produce low magnitude coefficients. When you have coefficients up to 255 in magnitude and you want to compress the image lossless, this requires seven runs of alternating Dominant and Subordinate Passes, so this already requires 7 Kb of space, to represent subband zero, which contains the most important parents. For a 7th order hierarchical decomposition this equals to a subband zero of resolution 2x2 = 4 bytes. In a Dominant Pass these are represented by just 1 byte. This means less space is used to represent important parent coefficients.

When looking at (almost) lossless compression, the algorithm does not perform very well for quality 1, which should be lossless. When the Dominant Pass has finished, there is enough information present to reconstruct all coefficients with a maximum error of 1 in magnitude. This means that performing a standard Subordinate Pass will not provide the necessary information to be able to produce a lossless representation of the original decomposition. After that, a Dominant Pass should be performed to find the coefficients that have magnitude 1. This means that all remaining coefficients are really zero. A lot of redundant information is most likely to be produced. To solve this, an alternative procedure has to be developed to produce lossless compression at quality 1. This has not been done yet. Implementing this will also acquire a new method to check whether reproduction is correct. If any differences larger than 1 will arise in this last step, it is obvious that there is a problem in the algorithm, as implemented.

In the paper, the coder is designed to stop at a certain bit rate or a certain signal-to-noise ratio. It is possible to implement this in the version made at IPO, although some problems will arise, especially with the bit rate part. Signal-to-noise ratio is 'fully' detached from the bytestream length. This means, that additional lossless compression does not influence the signal-to-noise ratio. Implementing this feature will not be the problem, but it will take a lot more of processing time to compute a bytestream.

The separation of the coder into functional blocks, causes the lossless arithmetic compression to be fully detached from the bytestream production, so a feature that makes sure coding stops at a certain bit rate will only be possible for the uncompressed bytestream. This feature is already implemented. Unfortunately, the result of further lossless compression is almost unpredictable, so adding such a feature requires additional code in the zerotree coder, to obtain runlength lossless arithmetic compression. The additional arithmetic coder present in Khoros, in this case, will not be necessary anymore. In contrast to the objections, these features will turn the coder into a very useful reference coder.

Chapter 5

Conclusions and Recommendations

Although the research that can be done in the direction of this kind of compression is not finished, we can arrive at some conclusions and recommendations regarding the work already done.

5.1 Conclusions

An important difference between the implemented coder and the one published in [Sha93] is, that it uses different image decomposition methods. This is the main reason for the differences in performance between the coder at IPO and the embedded coder, although some things, like subband zero treatment, are not properly described in the paper and will have a major influence, too.

The zerotree algorithm seems very promising. Since very few researchers have been experimenting with it, until recently, not too many results are achieved that can be compared. But according to the latest news at conferences and the like, a lot more will be heard about it in the near future.

All the major implementation problems have been taken care of. There still is a minor bug, but it can be solved quite easily, as described in section 4.1.

The amount of tests performed, to prove whether specific parts of the coder work like they should, is too small. Additional research and tests will be necessary to make sure that everything works properly.

Without the hierarchical decomposition implemented, experiments on extremely low bit rates will be useless. As shown in section 4.2 this is very easy to prove. The exponentially growing parent-child trees prove to be very compressible.

5.2 Recommendations

Very recent developments are showing a growing interest in zerotree coding and derivates of it. Therefore it might be wise to continue to do research in this direction. It might even lead to better compression results, when combining it with the knowledge, already present at IPO. Keeping track of the new developments will be necessary, because some of the problems and differences stated in this report might already be solved by other researchers, or may be solved in the near future.

Implementing the hierarchical decomposition in Khoros and in the coder will certainly improve performance. According to the paper ([Sha93]) this should increase performance about 1.5 times. (Where 1.5 stands for the ratio to which the length of a bytestream decreases, while maintaining the same image quality.)

Other kinds of filtering to obtain a decomposition should be tested. There is an obvious difference between DCT transforms and Hermite transforms, and even changing parameters while using the same transform results in large differences in obtainable compression ratios. It might be quite possible that it depends a lot on the original, so some major research has to be done in compression ratios related to the transform method.

To make sure that the coder performs correctly, additional tests have to be done. This involves using different kinds of originals, using different kinds of decomposition methods and checking the exact reproduction of originals on bit level.

Chapter 6

Evaluation

As already mentioned in the introduction, I could not help but notice the pleasant working atmosphere at IPO. If you would need any help or advice, there was always someone who would help you out. The fact that I chose to do a project at IPO, vision research department, fits well with my personal interest in image processing. I also followed some courses in image processing and technology.

The project itself was a little to comprehensive for the time that stood for it. Since I managed to complete almost every goal in the project, it is unnecessary to say that it took a lot more time than planned. I could have quitted the project as soon as the amount of time invested was sufficient, but what is five weeks? Too less for a real project, that is. When you read this, I will have spend almost three times the required amount of time. Maybe there are people out there who think this is just a good example of bad scheduling, but I'm afraid it is more an example of typical TUE practice. There is always to less time scheduled for practical training and projects. So be it.

Apart from all this, I would like to thank my supervisor for his pleasant and willingly support. I learned a great deal of him. Now I know a lot more about compression and image processing techniques. My C programming skills have improved a lot, also, and my knowledge about unix and Xwindows has greatly advanced.

My first official lecture, was quite an experience, too. But thanks to the support of many people, this worked out quite well. I would like to conclude with saying that working at IPO when you are a student is very recommendable, because you encounter so many different people from whom you can always learn something new.

Appendix A

Originals

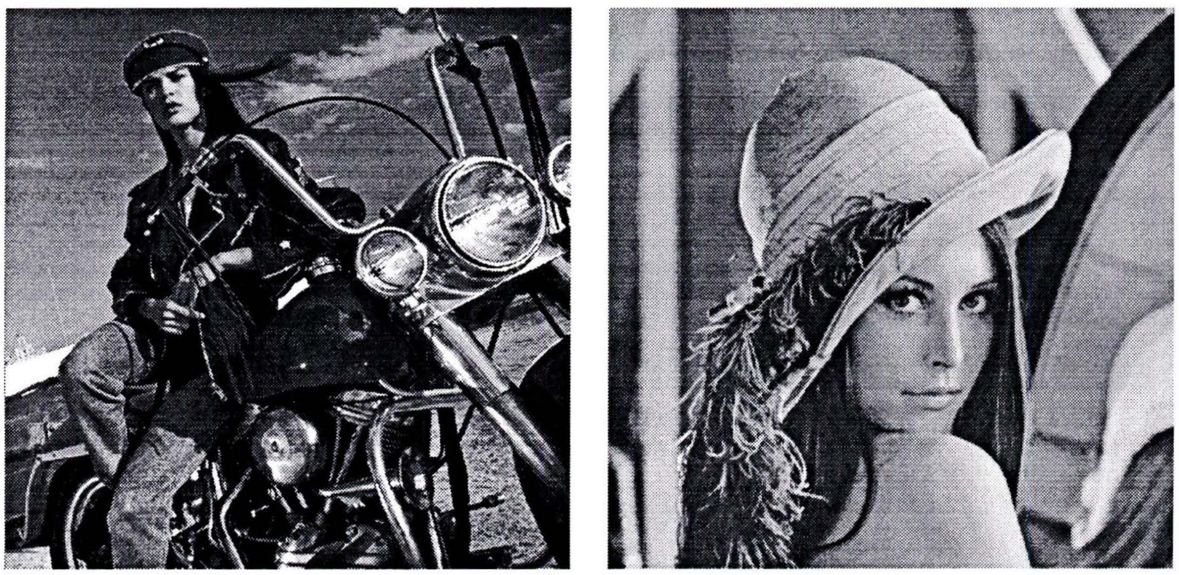


Figure A.1: The two originals that were used. Left the 'Bikerlady' and right 'Lena'.

Appendix B

Program code

This part of the appendix contains the C-code that was generated, to perform zerotree coding from within the Khoros software. The interface which can be used to start this program, is described in Appendix C All the different kinds of input options can be set in this interface.

```
-authors
M.L.A.Bierman
-authors_end

-short_prog_description
ZEROTREE performs quantization and bytestream conversion of an image description
according to the zerotree-algorithm.
-short_prog_description_end

-short_lib_description
ZEROTREE performs quantization and bytestream conversion of an image description
according to the zerotree-algorithm.
-short_lib_description_end


/*****************************************************************/
/* Programma-idee voor ZEROTREE */


/* Initialisatie */
/* Lees de start-opties  { standaard;zie oude sources } */

/* if toggle == 0 then Encode */

/* if toggle == 1 then Decode */

/*
Encode:

-Lees de decompositie van een plaatje in.
-Quantiseer en codeer het plaatje volgens het zerotree-algoritme.
 (Houdt hierbij rekening met de aangegeven boomstructuur.
  Deze beschrijft de parent-child-relaties.)
-Maak van de informatie een bytestream.
-Geef de boomstructuur en andere informatie door.
*/

/*
Decode:

-Lees de bytestream in.
```

```
-Decodeer het plaatje volgens het zerotree-algoritme.
 (Houdt hierbij rekening met de aangegeven boomstructuur.)
-Maak van de informatie een decompositie.
 (Gaat vermoedelijk samen met de vorige stap.)
-Geef de initiele informatie door, zoals eerst in het commentaarveld stond.
 (Dit gebeurt vermoedelijk al bij het initialiseren van de decompositie.)

*/

/*****************************************************************/


-man1_long_description
/***************/
-man1_long_description_end

 -man1_examples
-man1_examples_end

-man1_restrictions
/***************/
-man1_restrictions_end

-man1_see_also
-man1_see_also_end

-man3_long_description
-man3_long_description_end

-man3_restrictions
-man3_restrictions_end

-man3_see_also
-man3_see_also_end

-usage_additions
-usage_additions_end

-include_includes
-include_includes_end

-include_additions
-include_additions_end

-include_macros
-include_macros_end

-main_variable_list
        struct  xvimage *image, *readimage();
        char    *cpointer, *spointer;
        FILE    *cfile;

        int     poly;                   /* type of image decomposition */
        int     xy;                     /* processing option */
        int     ts;                     /* sampling distance */
        int     dnm;                    /* maximum order (signed) */
        int     dim;                    /* 1D or 2D coefficient order */
        int     ftype;                  /* filter type */
        int     frame;                  /* frame parameter */

        char    comment[512];           /* comment */
        char    prog[30];               /* program name */
        int     status;                 /* status flag */
        int     nrows, ncols, nbands, ntree, t_ini;
   long int     leng, size;
        int     i, j;
        float   one = 1.0;
        int      t = 1, f = 0;
-main_variable_list_end

-main_before_lib_call
        if (check_args()) exit(1);


/****** encode image decomposition ******/

        if ( zerotree->c_toggle == 0 )
        { image = readimage(zerotree->if_file);
          if(image == NULL)
          { fprintf( stderr, "Error opening file %s\n", zerotree->if_file );
            exit(1);
          }
          if ( image->data_storage_type != VFF_TYP_FLOAT )
            (void) lvconvert(image,VFF_TYP_FLOAT,f,t,one,one,f);

          status = sscanf( image->comment, "poly = %2d, xy = %1d, ts = %2d, \
dnm = %2d, dim = %1d, ftype = %2d, frame = %1d -- generated by %s",
          &poly, &xy, &ts, &dnm, &dim, &ftype, &frame, prog );
          if ( status < 0 )
          { fprintf( stderr, "cannot read parameters from file %s\n",
                       zerotree->if_file);
            exit(1);
          }
          if ( (poly >= 3) && (poly <= 0) && (poly != -4) )
          /* When poly = 1, 2 or -4 we continue coding */
          { fprintf( stderr, "cannot perform coding for poly = %d\n",poly);
            exit(1);
          }
          nrows = image->col_size;
          ncols = image->row_size;
          nbands = image->num_data_bands;
          t_ini = 0;
          if (poly == -4) { ntree = nbands / 3; } else { ntree = 0; }

          cfile = fopen( zerotree->of_file, "w" );
          fputc( (char)poly, cfile );
          fputc( (unsigned char)xy, cfile );
          fputc( (unsigned char)ts, cfile );
          fputc( (unsigned char)dnm, cfile );
          fputc( (unsigned char)dim, cfile );
          fputc( (char)ftype, cfile );
          fputc( (unsigned char)frame, cfile );
          i = nrows / 256;
          j = nrows % 256;
          fputc( (unsigned char)i, cfile );
          fputc( (unsigned char)j, cfile );
          i = ncols / 256;
          j = ncols % 256;
          fputc( (unsigned char)i, cfile);
          fputc( (unsigned char)j, cfile );
          fputc( (unsigned char)nbands, cfile );
          fputc( (unsigned char)ntree, cfile );
          size = nrows * ncols * nbands;
          cpointer = malloc( size );
          if ( cpointer == NULL )
          { fprintf( stderr, "Unable to allocate sufficient memory");
            exit(1);
          }

        }


/****** decode image decomposition ******/
```

```
        if ( zerotree->c_toggle == 1 )
        { cfile = fopen( zerotree->if_file, "r" );
          if(cfile == NULL)
          { fprintf( stderr, "Error opening file %s\n", zerotree->if_file );
            exit(1);
          }
          poly = (char)fgetc( cfile );
          xy = fgetc( cfile );
          ts = fgetc( cfile );
          dnm = fgetc( cfile );
          dim = fgetc( cfile );
          ftype = (char) fgetc( cfile );
          frame = fgetc( cfile );
          i = fgetc( cfile );
          j = fgetc( cfile );
          nrows = i * 256 + j;
          i = fgetc( cfile );
          j = fgetc( cfile );
          ncols = i * 256 + j;
          nbands = fgetc( cfile );
          ntree = fgetc( cfile );
          size = ncols * nrows * nbands;


          cpointer = malloc( size );
          if ( cpointer == NULL )
          { fprintf( stderr, "Unable to allocate sufficient memory");
            exit(1);
          }
          spointer = cpointer;
          for (i=0; i < size; i++) *spointer++ = fgetc( cfile );
          fclose( cfile );

          sprintf( comment, "poly = %2d, xy = %1d, ts = %2d, dnm = %2d, \
dim = %1d, ftype = %2d, frame = %1d, ntree = %1d \
 -- generated by zerotree",
          poly, xy, ts, dnm, dim, ftype, frame, ntree, t_ini );
          image = createimage(
                (unsigned long) nrows,                  /* number of rows */
                (unsigned long) ncols,                  /* number of columns */
                (unsigned long) VFF_TYP_FLOAT,          /* data storage type */
                (unsigned long) 1,                      /* num_of_images */
                (unsigned long) nbands,                 /* num_data_bands */
                comment,                                /* comment */
                (unsigned long) 0,                      /* map_row_size */
                (unsigned long) 0,                      /* map_col_size */
                (unsigned long) VFF_MS_NONE,            /* map_scheme */
                (unsigned long) VFF_MAPTYP_NONE,        /* map_storage_type */
                (unsigned long) VFF_LOC_IMPLICIT,       /* location type */
                (unsigned long) 0 );                    /* location_dim */
          if (image == NULL)
          { fprintf( stderr, "Unable to create output image\n");
            exit(0);
          }
        }
-main_before_lib_call_end

-main_library_call
        t_ini = lzerotree( image, cpointer, &size, zerotree->c_toggle, ntree,
                        zerotree->ipcf_file, zerotree->length_int,
                        zerotree->t_out_int );
         if (!t_ini) { (void) fprintf(stderr, "lzerotree Failed\n");
                       exit(1); }
-main_library_call_end

-main_after_lib_call
        if ( zerotree->c_toggle == 0 )
        {
          i = (size / (256*256*256)); j = (size / (256*256));
          fputc( (unsigned char) i, cfile);
          fputc( (unsigned char) j, cfile);
          i = (size / 256); j = (size % 256);
          fputc( (unsigned char) i, cfile);
          fputc( (unsigned char) j, cfile);
          for ( i=0; i < size; i++ ) fputc( *cpointer++, cfile );
          fclose( cfile );
          printf( "encoded information, no_of_bytes = %d\n", size );
          one = (float)(8*size) / (nrows*ts*ncols*ts);
          printf( "bit rate is %f bits/pixel\n", one );
        }
        if ( zerotree->c_toggle == 1 )
        {
          writeimage(zerotree->of_file,image);
          if ( poly == -4 ) printf( "Decoded image is a hierarchical decomposition of level %d\n",ntree );
        }
-main_after_lib_call_end

-library_includes
#include "/home/khoros/newtools/include/poly.h"
-library_includes_end

-library_input
.IP "image" 15
multiband xvimage structure containing the input image decomposition
if cmode = 0
.IP "cpointer" 15
pointer to character array that holds coded bitstream
.IP "csize" 15
size of character array
.IP "cmode" 15
either encode (cmode=0) or decode (cmode=1) image decomposition
-library_input_end

-library_output
.IP "cpointer" 15
character array holding the encoded bitstream if cmode=0
.IP "csize" 15
length of encoded bitstream (in #bytes) if cmode=0
.IP "image" 15
multiband xvimage structure containing the output image decomposition
if cmode=1
.IP "lzerotree" 15
return zero on failure and t_ini upon success (t_ini if cmode=0 else TRUE)
-library_output_end

-library_def

/****** global variables ******/

int   nrows, ncols, nbands, nsize, size, ntree, t_ini, conversion, dec;
                                /* parameters of image decomposition */
unsigned char  *spointer;       /* coded bitstream pointer */
int   *im_buffer;               /* integer converted image */
int   *el_buffer;               /* coded elements buffer */
char  *coding_buffer;           /* memory for coordinates of old and new
                                   elements */
char pc_rel[100][8];            /* parent-child relations buffer */
int  count8, temp;              /* bitstream variables */

int maxt_out, quit;
long int maxlength, bsize, leng;
```

```
/***************************************************************************
        start library procedure
 ***************************************************************************/

int lzerotree( image, cpointer, csize, cmode, ntree, pc_file,
               mlength, maxt_out )
struct  xvimage *image;
unsigned char   *cpointer;
long int        *csize;
int             cmode;
char            *pc_file;
int             mlength;

-library_def_end

-library_code
{
        int     poly;                   /* type of image decomposition */
        int     xy;                     /* processing option */
        int     ts;                     /* sampling distance */
        int     dnm;                    /* maximum order (signed) */
        int     dim;                    /* 1D or 2D coefficient order */
        int     ftype;                  /* filter type */
        int     frame;                  /* frame parameter */



        int     status;                 /* status flag */
        char    prog[30];               /* program name */
        int     i, x, y, T, max;
        float   *pointer;               /* image data pointer */
        FILE    *ipcf;


        void reset_loc_buffer();
        void copy_buffer();
        void retrieve_dom();
        void retrieve_sub();
        void detect_elements();
        void create_dom_code();
        void create_sub_code();
        void get_ini_t();
        int get_p_c();
        void code();
        void decode();

        nrows = image->col_size;
        ncols = image->row_size;
        nbands = image->num_data_bands;


        nsize = nrows * ncols;
        size = nrows * ncols * nbands;
        maxlength = 1024 * mlength;
        count8 = 8;
        bsize = 0;
        quit = FALSE;                 /* keep coding as long as quit==FALSE */
        temp = 0;


   spointer = cpointer;                     /* make pointer global */
   pointer = (float *)image->imagedata;     /* the image pointer    */
   im_buffer = (int *)calloc(size,sizeof(int));
   el_buffer = (int *)calloc(size,sizeof(int));
   coding_buffer = (char *)calloc(size,sizeof(char));

   if ( (im_buffer == NULL) || (el_buffer == NULL)  || (coding_buffer == NULL) )
   { fprintf( stderr, "Unable to allocate sufficient memory"); exit(1); }

   max = 1;
   for (i=0; i < maxt_out; i++) max *= 2;

   printf("nbands = %d and maxlength = %d\n", nbands, maxlength);


                                       /* Read the parent-child relations, and */
   for (y = 1; y < nbands + 1; y++) /* put them in a buffer, where 111 means */
   {                                   /* "no relation found"                  */
     for (x = 1; x < 9; x++)
     {
        pc_rel[y-1][x-1] = get_p_c(pc_file,y,x);
     } /* End FOR x */
   } /* End FOR y */


              /**********************************************/
              /**********    code coefficients    **********/
              /**********************************************/

        if ( cmode == 0 )
        {

        /* initialize parameters */

           status = sscanf( image->comment, "poly = %2d, xy = %1d, ts = %2d, \
dnm = %2d, dim = %1d, ftype = %2d, frame = %1d -- generated by %s",
           &poly, &xy, &ts, &dnm, &dim, &ftype, &frame, prog );
           if ( status < 0 )
           { fprintf( stderr, "lzerotree: cannot read parameters\n");
             exit(1);
           }
           conversion = poly;            /* make poly globally known */


           for (i=0; i < size; i++) *im_buffer++ = (int)*pointer++;
           im_buffer -= size;  /* Initializing copy of original image */
/*           for (i=nsize; i>0; i--) *(im_buffer + i) -= *(im_buffer + i-1); */
           for (i=0; i<nsize; i++) *(im_buffer + i) -= 128;
           /* DC-coefficients now are between -128 and 128 so the algorithm
              works more efficiently */

           t_ini = 0;
           get_ini_t();
           T = t_ini;

           printf("code downto : threshold = %d\n", max);

           reset_loc_buffer();

           while( (!quit) && (t_ini >= max) )
           {

        /****** Dominant Pass ******/

              detect_elements();   /* Find elements higher than threshold and
                                      put them in the "local_code_buffer",
                                      copy found elements to "elements_buffer"
                                      and make them zero in the "image_buffer"
                                      */

              create_dom_code();     /* Create dominant part of the bytestream,
                                        according to input parameters */
/*              printf("Dominant code generated\n"); */

        /****** Subordinate Pass ******/

             if (!quit)
             { reset_loc_buffer();
               create_sub_code();   /* Create subordinate part of bytestream */

/*               printf("Subordinate code generated\n"); */
             }
             t_ini /= 2;
          }

          *csize = bsize;


       }  /* End of "code coefficients" */


             /**********************************************/
             /**********   decode coefficients   **********/
             /**********************************************/

       if ( cmode == 1 )
       {

          x = *spointer++;
          y = *spointer++;
          leng = (x * 256*256*256 + y * 256*256);
          x = *spointer++;
          y = *spointer++;
          leng += (x * 256 + y);
          printf("Retrieved length = %d\n", leng);

          x = *spointer++;
          y = *spointer++;
          t_ini = (x * 256 + y);
          printf("Retrieved t_ini = %d  (256 x %d + %d)\n",t_ini, x, y);

          reset_loc_buffer();

          while ( (!quit) && (t_ini >= max) )
          {

       /****** Dominant Pass ******/

             retrieve_dom();  /* Get coordinates of new elements (in buffer) */

       /****** Subordinate Pass ******/

             if (!quit)
             { reset_loc_buffer();
               retrieve_sub();  /* Alter values of elements in buffer */

             }
             t_ini /= 2;

          }
/*          for (i=0; i<nsize; i++) *(im_buffer + i+1) += *(im_buffer + i); */
          for (i=0; i<nsize; i++) *(im_buffer + i) += 128;

          printf("Copying buffer to image\n");
          for (i=0; i < size; i++)
          { *pointer = (float)*im_buffer; pointer++; im_buffer++; }
                /* Put final version of image in image pointer */

          T = TRUE;
        } /* End of "decode coefficients" */
        printf("Size of decoded part : %d bytes\n",bsize);

        return(T);

} /* End of "lzerotree" */




/***************************************************************************
  code(n) :
    Procedure to add a one or a zero to the bitstream
***************************************************************************/
void code(n)
unsigned char n;
{
   count8--;
   if (count8 < 0) count8 = 7;
   if (n==1) { n = n << count8; temp = temp | n; }
   if (count8 == 0) { *spointer++ = temp; bsize++;
                      temp = 0;}
   if (bsize >= maxlength) { quit = TRUE; if (count8 != 0) *spointer++ = temp;}
}


/***************************************************************************
  decode :
    Procedure to substract the next bit in the bitstream
***************************************************************************/
void decode()
{
   int n;

   n = 1;
   count8--;
   if (count8 < 0) count8 = 7;
   if (count8 >= 7) { temp = *spointer++; bsize++; }

   n = n << count8;
   if (temp & n) {n = TRUE;} else {n = FALSE;}

   if ( (bsize >= maxlength) || (bsize > leng) )
   { quit = TRUE; printf("Quit is TRUE\n"); }

   dec = n;
}


/***************************************************************************
  retrieve_dom :
    Get coordinates of new elements (in buffer)
***************************************************************************/
void retrieve_dom()
{

/* Aan de hand van de resolutie, het aantal banden en de organisatie van de
   banden gaan we de coordinaten bepalen van de coefficienten. Deze leggen we
   vast in de coding_buffer, zodat ze onthouden blijven.
   Dit doen we tot we alle banden gehad hebben.
*/

   int pos, neg, IZ, Z, ZTR, dZTR;
   int curr_band, k, i, check, temp1, number, n, total;
   int kids[100];

   pos = 0; neg = 0; IZ = 0; Z = 0; ZTR = 0; dZTR = 0;
   curr_band = 0;
   check = 0;
   total = 0;

/*   if (conversion == -4) printf("Hierarchical is not yet implemented\n"); */
                                                      /* Hierarchical */


/*   if ( (conversion == 1) || (conversion == 2) )  { */
                                                      /* Polar & Carthesian */


      while (curr_band < nbands)
      {

               i = 0; n = 0;           /* Find all childs for curr_band */

               while ( (pc_rel[curr_band][i] > 0) &&
                       (pc_rel[curr_band][i] < 111) )
                  { kids[i] = pc_rel[curr_band][i++]; }

               for (k = 0; k < i; k++)
               {
                  if (pc_rel[kids[k]][0] != 0)
                  {
                     while ( (pc_rel[kids[k]][n] > 0) &&
                             (pc_rel[kids[k]][n] < 111) )
                     { kids[i+n] = pc_rel[kids[k]][n]; n++; }
                     i +=n;
                  }
                  n = 0;
               }
               number = i;

/* for (i=0; i<number; i++) printf("%d ",kids[i]);
printf("    Current band = %d and bsize = %d\n", curr_band, bsize); */

         if (curr_band == 0)
         {
            for (k = 0; k < nsize; k++)
            {
               decode(); if (dec)
               { total++;
                 decode();
                 if (dec) {*(coding_buffer + k) = 6; pos++;}
                 else {*(coding_buffer + k) = 2; neg++;}
               }
               else
               { *(coding_buffer + k) |= 8; Z++;
                 decode(); if (!dec)
                    { for (i=0; i < number; i++)
                        { *(coding_buffer + kids[i] * nsize + k) |= 8;
                          dZTR++;}
                    }
               }
            if (quit) goto end_dom;
            }


         }
         else   /* curr_band isn't zero */
         {
          /* check if decoding is necessary, if so, continue, else next band */
             check = 0;
             for (k=0; k < nbands; k++)
             {
                for (i=0; i < 8; i++)
                {  if (pc_rel[k][i] == curr_band) check = 1; }
             }

             if (check == 1)
             {
               for (k=0; k < nsize; k++)
               {
                 if ( (bsize == (leng + 1)) || (quit) ) goto end_dom;
                 if (*(coding_buffer + curr_band * nsize + k) & 8)
                 {goto next_one;}
                 decode(); if (quit) goto end_dom;
                      if (dec)
                      { total++;
                        decode(); if (dec)
                        {*(coding_buffer + curr_band * nsize + k) = 6;
                         pos++; }
                        else
                        {*(coding_buffer + curr_band * nsize + k) = 2;
                         neg++; }
                      }
                      else
                      { if (pc_rel[curr_band][0] == 0)
                        {*(coding_buffer + curr_band * nsize + k) |= 8;
                         Z++;}
                        else
                        {  Z++;
                           *(coding_buffer + curr_band * nsize + k) |= 8;
                           decode(); if (!dec)
                              { for (i=0; i < number; i++)
                                 {*(coding_buffer + kids[i] * nsize + k) |= 8;
                                  dZTR++;}
                              }
                        }
                      }
                 next_one: ;
               }  /* End FOR k */
             }  /* End IF check */
          }
/*
printf("Sign.: %d, Pos: %d, Neg: %d, IZ+Z+ZTR: %d, dZTR: %d\n", total, pos, neg, Z, dZTR);
*/
          curr_band++;
       }  /* End WHILE curr_band */


   end_dom: ;

/* printf("bsize = %d\n",bsize); */

       for (k = 0; k < size; k++)
       {
          if ( (*coding_buffer & 2) && (!(*coding_buffer & 1)) )
          {    if (*coding_buffer & 4) { *im_buffer = (int)(t_ini*1.5); }
               else { *im_buffer = (int)(t_ini*-1.5); }
          }

          coding_buffer++; im_buffer++;
       }
       coding_buffer -= size; im_buffer -= size;


/*    }   End of Polar & Carthesian */
```

```
}


/****************************************************************************
  retrieve_sub :
    Alter values of elements in buffer
****************************************************************************
void retrieve_sub()
{

/* Aan de hand van het totale aantal gevonden elementen en de t_ini gaan w
   deze aanpassen in de im_buffer
*/
   int k;
   int pos, neg;

   pos = 0; neg = 0;

   for (k = 0; k < size; k++)
   {
      if (*coding_buffer & 1)
      {  decode(); if (dec) { *im_buffer += (int)(t_ini/4); pos++;}
         else { *im_buffer -= (int)(t_ini/4); neg++;}
      }

      if (quit) goto end_sub;
      coding_buffer++; im_buffer++;
   }
   coding_buffer -= size; im_buffer -= size;

   end_sub: if (k < size) { coding_buffer -= k; im_buffer -= k;};
/* printf("decoded pos : %d, neg : %d\n",pos, neg); */
}


/****************************************************************************
  detect_elements :
    Procedure to find elements higher than the current threshold and put tem
    in the "local_code_buffer" and to copy found elements to "elements_buf:er"
    and make them zero in the "image_buffer"
****************************************************************************
void detect_elements()
{
   int k, total;

   total = 0;
   for(k=0; k < size; k++)
   {
      if (abs(*im_buffer) >= t_ini)
      { *coding_buffer = 2;
        *el_buffer = *im_buffer;
        *im_buffer = 0;
        total++;
      }
      im_buffer++;
      coding_buffer++;
      el_buffer++;
   }
/*   printf("New (detect)elements found : %d\n", total); */
   im_buffer -= size;
   coding_buffer -= size;
   el_buffer -= size;

}
```

```
/****************************************************************************
  create_dom_code :
    Procedure to create dominant part of the bytestream, according to input
    parameters (hard part)
****************************************************************************/
void create_dom_code()
{
        /*
        - kijk of curr_band gecodeerd moet worden (dus met parents,
          of wanneer curr_band == 0 )
        nsize maal:
           - lees *(im_buffer + nsize * curr_band)
           - bepaal met t_ini of POS/NEG -> ja: code en nieuwe, nee: ga verder
           - kijk of er parents zijn -> ja: als code is ZTR of afstammeling
             van ZTR dan nieuwe en anders ga verder, nee: als curr_band == 0
             dan ga verder en anders nieuwe
           - kijk of er kids zijn -> nee: code Z, ja: is er een groter dan
             t_ini, code IZ en anders code ZTR
        - ga naar het eerste punt
        */

   int pos, neg, IZ, Z, ZTR, dZTR;
   int curr_band, k, i, l, check, temp1, number, n, once, par, total;
   int kids[100];

   curr_band = 0;
   once = 0;
   total = 0;
   par = 0;   /* memorize parent */
   pos = 0; neg = 0; IZ = 0; Z = 0; ZTR = 0; dZTR = 0;


/*   printf("create_dom_code start : conversion = %d\n", conversion); */

   if (conversion == -4) printf("Hierarchical is not yet implemented\n");
                                               /* Hierarchical */


   if ( (conversion == 1) || (conversion == 2) )  /* Polar & Carthesian */
   {

         while (curr_band < nbands)
         {

/*         printf("current band : %d  bsize : %d\n", curr_band, bsize);
printf("count8 is %d\n",count8); */

            check = 0;
            k = 0;
            if (curr_band != 0)
            {                                  /* search for parent */
               for (k=0; k < nbands; k++)
               {
                  for (i=0; i < 8; i++)
                  { if (pc_rel[k][i] == curr_band) {check = 1; par = k;} }
               }
            }   /* kijk of curr_band gecodeerd moet worden (dus met parents,
                   of wanneer curr_band == 0 ) */

/*            if (check == 1) printf("Parent found : %d\n", par); */
            if ( (check != 1) && (curr_band != 0) )
            { printf("Band %d is not coded\n", curr_band); goto newband;}

                /* parent found or curr_band == 0 */


               i = 0; n = 0;      /* Find all children of curr_band */
```

```
                while ( (pc_rel[curr_band][i] > 0) &&
                        (pc_rel[curr_band][i] < 111) )
                   { kids[i] = pc_rel[curr_band][i++]; }

                for (k = 0; k < i; k++)
                {

                   if (pc_rel[kids[k]][0] != 0)
                   {
                      while ( (pc_rel[kids[k]][n] > 0) &&
                              (pc_rel[kids[k]][n] < 111) )
                      { kids[i+n] = pc_rel[kids[k]][n]; n++; }
                      i +=n;
                   }
                   n = 0;
                }
                l = i;

           for (number = 0; number < nsize; number++)
           {
             temp1 = *(el_buffer + nsize * curr_band + number);
             if (*(coding_buffer + nsize * curr_band + number) & 1) {temp1 = 0;}

             if (abs(temp1) >= t_ini)    /* Is this element significant ? */
             {
                code(1);
                if (quit) goto end;
                total++;
                if (temp1 > 0) {code(1); pos++;} else {code(0); neg++;}
                goto next;
             }
             if (quit) goto end;

             if ( (*(coding_buffer + nsize * par + number) & 8 )
                  && ( curr_band != 0) )
             {  *(coding_buffer + nsize * curr_band + number) |= 8;
                dZTR++;
                goto next;
             }   /* Is parent ZTR or descendant, than current element too */

             if (pc_rel[curr_band][0] == 0) { code(0);}
             else                 /* Find IZ or ZTR */
             {
                code(0);
                n = 0;
                for (k = 0; k < l; k++)
                { if ( (abs(*(el_buffer + kids[k] * nsize + number)) >= t_ini)
                       && (!(*(coding_buffer + kids[k] * nsize + number) & 1)) )
                  n = 1;
                }
                if (n == 1) {code(1); IZ++;}
                else { code(0); ZTR++;
                       *(coding_buffer + nsize * curr_band + number) |= 8;
                     }

             }
           if (quit) goto end;

           next: ;

           }  /* End FOR number */

           newband: ;

             curr_band++;
/* printf("Sign.: %d, Pos: %d, Neg: %d, IZ: %d, Z: %d, ZTR: %d, dZTR: %d\n", total, pos, neg, IZ, Z, ZTR, dZTR); */
          }  /* End WHILE curr_band */
```

```
   end: ;
   } /* End Polar & Carthesian */


}


/****************************************************************************
  create_sub_code :
    Procedure to create subordinate part of bytestream (easy part)
****************************************************************************/
void create_sub_code()
{
   int k;
   int pos, neg;

   pos = 0; neg = 0;

   for (k = 0; k < size; k++)
   {
      if (*coding_buffer & 1)
      {
         if (abs(*el_buffer) >= t_ini)
         {  if (*el_buffer > 0) {*el_buffer -= t_ini;}
            else {*el_buffer += t_ini;}
         }
         if (abs(*el_buffer) < (t_ini/2)) {code(0);} else {code(1);}
      }
      coding_buffer++; el_buffer++;
   }
   coding_buffer -= size; el_buffer -= size;
/* printf("coded pos : %d, neg : %d\n",pos, neg); */
}


/****************************************************************************
  reset_loc_buffer :
    Procedure to reset all coefficients found in the last Dominant Pass and
    copy them to the global buffer
****************************************************************************/
void reset_loc_buffer()
{
   int k, total;

   total = 0;
   for(k=0; k < size; k++)
   {  if (*coding_buffer & 2) { *coding_buffer = 1; total++;}
      if (*coding_buffer & 1) { *coding_buffer = 1;}
      if ( !(*coding_buffer & 1) ) { *coding_buffer = 0;}

      coding_buffer++;
   }
   coding_buffer -= size;
/* printf("Total resets : %d\n",total); */

}


/****************************************************************************
  get_ini_t :
    Procedure to find the initial threshold
****************************************************************************/
void get_ini_t()
{
   int k, t;
   float s1,s2;


       t = 0;
       for(k=0; k < size; k++)
       {
          if ( abs(*im_buffer) > t ) t = abs(*im_buffer);
          im_buffer++;

       }
       im_buffer -= size;
                 /* Find a power of two which is at least as
                    big as half of the maximum element       */
       k = 1;
       while (k < t) k *= 2;
       t = k / 2;
       k = t;

       *spointer++ = (unsigned char)(k / 256);
       *spointer++ = (unsigned char)(k % 256);
       t_ini = t;
       printf("final t_ini = %d\n",t_ini);

}

/**************************************************************************
  get_p_c :
    Procedure to retrieve the parent-child dependencies
**************************************************************************/
int get_p_c(fn,m,n)
char *fn;
int  m,n;
{
   int c,i;
   FILE *fp;
   int rij,kolom;
   char s[80];
   int result;

   rij =1; kolom=1; i=0; s[79] = '-';
   fp = fopen(fn,"r");


   while (rij != m)
       if ((c = getc(fp)) == '\n') rij++;
    /* no. rows is always m or smaller */

   while ( (c = getc(fp)) == ' ');
   s[i] = c; i++;

   if (kolom != n)
   {
     i = 0;

     while (kolom != n)
     {
       while ( (c = getc(fp)) != ' ') if ((c == '\n') || (c == EOF)) goto out;
       while ( (c = getc(fp)) == ' ') if ((c == '\n') || (c == EOF)) goto out;

       kolom++;
     }
     if ( (c != '\n') && (c != EOF) ) {s[i] = c; s[79] = 'x';} else
     { goto out;}
     i++;
   } else {s[79] = 'x';}

   while ( (c = getc(fp)) != ' ' && (c != EOF) && (c != '\n') )
   {
     s[i] = c;
     i++;
   }

   out:
   s[i] = '\0';
   if (s[79] == '-') { s[0] = '1'; s[1] = '1'; s[2] = '1'; s[3] = '\0'; }

   fclose(fp);

   result = atoi(s);

   return(result); /* Result is 111 if nothing is found. Check for valid m */
}


/*************************************************************************/

-library_code_end

-library_mods

-library_mods_end
```

Appendix C

The interface

In this part of the appendix, a so-called . pane-file is shown. This file contains a number of codes which produce an interface that can be used in Cantata, which is a GUI that can be used to easily interconnect and use the different kinds of programs available from the Khoros software library.

```
-F 4.2 1 0 170x7+10+20 +35+1 'CANTATA Visual Programming Environment for the KHOROS
 System' cantata
-M 1 0 100x40+10+20 +27+1 'IPO - Polynomial Processing' poly_processing
-P 1 0 80x38+22+2 +0+0 'Zerotree - Convert between image decomposition and byte stream'
 zerotree
-b +0+2 'Input filenames'
-I 1 0 0 1 0 1 58x1+2+3 +0+0 ' '  'Input file        '  'nonquantized image decomposition
 or input byte stream' if
-I 1 0 0 1 0 1 58x1+2+4 +0+0 ' '  'Parent-Child file ' 'input parent-child dependencies
 file' ipcf
-b +0+6 'Output filenames'
-O 1 0 0 1 0 1 58x1+2+7 +0+0 ' '  'Output file       ' 'output byte stream or nonquantized
 image decomposition' of
-b +0+10 'Options'
-c 1 0 0 1 0 27x1+0+11 +2+0 2 0 'Coding direction' 'coding direction' c 'image
 decomposition -> byte stream' 'byte stream -> image decomposition'
-c 1 0 0 1 0 27x1+0+13 +2+0 3 0 'Conversion type ' 'conversion type' conversion 'Polar'
 'Cartesian' 'Hierarchical'
-i 1 0 0 1 0 58x1+2+16 +0+0 0 512 512 'Length of bytestream (Kb)' 'Size of uncompressed
 bytestream' length
-i 1 0 0 1 0 58x1+2+18 +0+0 1 10 1 'Quality (threshold) ' 'Quantisation level' t_out
-R 1 0 1 13x2+1+21 'Execute' 'do operation' zerotree
-H 1  13x2+46+21 'No Help' 'man page for lattice' $NEWTOOLS/doc/manpages/zerotree.1
-E
-E
-E
```

Bibliography

[BA83] P.J. Burt and E.H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transitions on Communications*, COM-31:532–540, 1983.

[ea93] R. Aravind et al. Image and video coding standards. *AT&T Technical Journal*, pages 67–77, jan/feb 1993.

[IW87] J.G. Cleary I.H. Witten, R.Neal. Arithmetic coding for data compression. *Communications ACM*, 30:520–540, June 1987.

[Jai81] A.K. Jain. Image data compression: A review. *Proceedings of the IEEE*, 69:353–364, March 1981.

[MKK85] A. Ikonomopoulos M. Kunt and M. Kocher. Second-generation image-coding techniques. *Proceedings of the IEEE*, 73(4):353–364, April 1985.

[Pea95] D.E. Pearson. Developmets in model-based video coding. *Proceedings of the IEEE*, 83(6):892–906, June 1995.

[RJ91] M. Rabbani and P.W. Jones, editors. *Digital Image Compression Techniques*, volume TT 7 of *Tutorial Texts in optical engineering*. SPIE Optical Engineering Press, 1991.

[Sha93] J.M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE transactions on signal processing*, 41(12):3445–3462, December 1993.

[vDM97] A.M. van Dijk and J.B.O.S. Martens. Image representation and compression using steered hermite transforms. In *Signal Processing*. Accepted for publication, 1997.