A study of image compression techniques, with specific focus on weighted finite automata

Rikus Muller

Thesis presented in partial fulfilment of the requirements for the Degree of Master of Science at the University of Stellenbosch.

Promoters: Prof. B.M. Herbst and K.M. Hunter

December 2005

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Rikus Muller

Date

Summary

Image compression using weighted finite automata (WFA) is studied and implemented in Matlab. Other more prominent image compression techniques, namely JPEG, vector quantization, EZW wavelet image compression and fractal image compression are also presented. The performance of WFA image compression is then compared to those of some of the abovementioned techniques.

Opsomming

Beeldkompaktering deur middel van geweegde eindige outomate (WFA) is bestudeer en geïmplementeer in Matlab. Ander meer prominente beeldkompakteringstegnieke, naamlik JPEG, vektorkwantifisering, EZW golfie beeldkompaktering en fraktaal beeldkompaktering word ook aangebied. Die prestasie van WFA beeldkompaktering word daarna vergelyk met dié van sommige van die bogenoemde tegnieke.

Contents

Chapter 1

Introduction

Over the past decade or so digital images have become an everyday aspect of our lives, with people storing their photographs on their computers, transmitting them over the internet, and pictures decorating websites. Digital images are also used for the archival of medical images and legal documents. However, in raw form, such images require large amounts of computer storage space. Even with ever increasing capacities in harddisk space, such requirements are daunting and impractical. Therefore digital images are almost always stored in compressed form.

In image compression and signal compression we distinguish between lossless and lossy compression. In the case of lossless compression, the decoded image is identical to the original image. Lossless compression is of particular importance in the compression of medical images and legal documents. However, typical compression ratios achieved with such techniques range between 2:1 and 3:1.

On the other hand, lossy compression entails the approximation of the original image, resulting in much higher compression ratios. With lossy compression a trade-off is made between the quality of the reconstructed image (relative to the original) and the compression ratio obtained. Thus, the “closer” the approximant is to the original image, the more storage space is required for the output (compressed) file and the lower the compression ratio, while the higher the compression ratio, the lower the quality of the reconstructed image. Compression ratios achieved where the resulting error is almost, if not totally, imperceptible to the human eye are at least 10:1.

1.1 Digital Representation of Images

A grayscale image is represented digitally by a matrix of m-by-n pixels. Pixels are typically unsigned 8-bit integer values, ranging from 0 to 255, and therefore allowing for 256 shades of gray. Colour images can be simulated, according to trichromatic theory, by linear combinations of the three primary colours red, green and blue, resulting in what is known as a *colourspace* (in this case referred to as the RGB colourspace). Colour images therefore consist of three matrices representing each pixel's coordinates in some or other colourspace. Since each of these matrices consist of 8-bit values, each pixel has 24-bit precision. This is known as 24-bit colour. Most of this thesis will be restricted to grayscale images, since image compression techniques can be generalized to colour images relatively easily.

Other colourspaces can be constructed by linear transformations of the RGB colourspace, most notably the *chrominance-luminance* colourspaces, namely YUV, YIQ and $\mathrm{YC_bC_r}$. The Y component is always the luminance component and provides a grayscale version of the image, whereas the other two components constitute the chrominance components. The Y component is defined by the weighted sum

$$\mathrm{Y} = 0.3\mathrm{R} + 0.6\mathrm{G} + 0.1\mathrm{B}.$$

Chrominance is defined as the difference between a color and a reference white at the same luminance. As a result, the components U and V are defined by

$$\mathrm{U} = \mathrm{B} - \mathrm{Y},$$

$$\mathrm{V} = \mathrm{R} - \mathrm{Y}.$$

These three coordinates (Y, U and V) make up the YUV colorspace, which is used in the European (PAL) television broadcasting system. On the other hand, the YIQ colourspace (also known as brightness, hue and saturation) is used in the North American (NTSC) television standard. The Y component is the same as in the YUV colourspace, while I and Q are defined by the transformation

$$\mathrm{I} = 0.74\mathrm{V} - 0.27\mathrm{U},$$

$$\mathrm{Q} = 0.48\mathrm{V} + 0.41\mathrm{U}.$$

Finally, the JPEG standard uses the colour coordinate system $\mathrm{YC_bC_r}$. Once again, Y remains the same as before, with the components $\mathrm{C_b}$ and $\mathrm{C_r}$ related to U and V, respectively, as follows:

$$\mathrm{C_b} = 0.5\mathrm{U} + 0.5,$$

$$\mathrm{C_r} = 0.625\mathrm{V} + 0.5.$$

1.2 Overview

This thesis is an overview and comparison of the most well-known lossy compression techniques as well as an implementation and investigation of a lesser known technique. In Chapter 2, data compression (also known as entropy coding) concepts are introduced and the two most prominent data compression algorithms, namely Huffman coding and Arithmetic coding, are explained. Data compression is often used in lossy compression techniques and always in lossless compression.

Chapter 3 covers two older lossy compression algorithms, namely the Discrete Cosine Transform (DCT) as utilized by JPEG, and vector quantization (VQ). Two more recent techniques, intrinsically linked with the concept of resolution, are discussed in Chapters 4 and 5: Image compression using wavelet transforms makes use of multiresolution analysis

(MRA) and is covered in Chapter 4, while fractal image compression yields encodings that are resolution independent and is based on Iterated Function Systems (IFS), as discussed in Chapter 5.

Another more recent algorithm, and the one implemented and investigated for this thesis makes use of weighted finite automata (WFA). Like fractal image compression, this leads to resolution independent encodings. As such, these two techniques are related and can be considered as "fractal like" techniques. Image compression using WFA is discussed in Chapter 6, as well as the implementation of such a scheme. Finally, in Chapter 7 we present comparisons of compression results obtained for some of these techniques.

Chapter 2

Data Compression Concepts

Data compression entails the exploitation of correlation in a string of symbols. Consider, for instance, the string of text characters **abacdaab**, to be stored in a text file. One could typically encode this string as a sequence of 8-bit ASCII characters, also referred to as using *fixed length* codewords. However, since some characters occur more frequently than others, we could also use fewer bits (assign shorter codewords) to encode the more probable characters and more bits (assign longer codewords) to encode the less probable characters. This is known as using *variable length* codewords.

The two most popular data compression algorithms are Huffman coding and arithmetic coding. To illustrate how coding with variable length codewords can be done, we apply Huffman coding to the string above, in Section 2.1. In Section 2.2, data compression terminology is introduced, and in Section 2.3, arithmetic coding is applied to a different string to illustrate its superiority over Huffman coding.

2.1 Huffman Coding

Huffman coding is based on building a *Huffman tree.* Firstly, the distinct symbols are sorted in a list according to increasing probability, resulting in the list illustrated in

Table 2.1.

Symbol	c	d	b	a
Probability	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$

Table 2.1: Sorted list, during first iteration of Huffman coding

Next, the first two symbols in the list (the two least probable symbols) are extracted and merged to form a node in the tree, resulting in a new symbol with its probability given by the sum of the probabilities of these two symbols. This results in the tree shown in Figure 2.1.

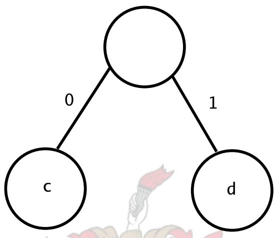


Figure 2.1: Huffman tree, after first iteration of Huffman coding

We assign a **0** and a **1** to the left and right branch, respectively, of this new node, and the two symbols that were merged form the leaves of the node. The newly created symbol is then inserted into the list (in such a way that the list remains sorted), yielding the list shown in Table 2.2.

Symbol	b	c + d	a
Probability	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$

Table 2.2: Sorted list, during second iteration of Huffman coding

The first two symbols are extracted and merged into a new node, resulting in the tree shown in Figure 2.2.

After inserting this new symbol in the list, we have the list shown in Table 2.3.

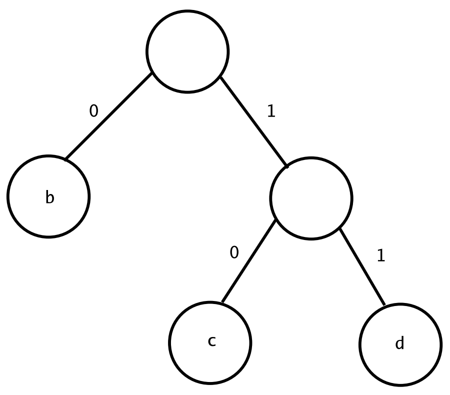


Figure 2.2: Huffman tree, after second iteration of Huffman coding

Symbol	a	b + c + d
Probability	$\frac{1}{2}$	$\frac{1}{2}$

Table 2.3: Sorted list, during third iteration of Huffman coding

Once again, we extract the first two entries from the list and perform the merging operation, upon which our (completed) Huffman tree looks as illustrated in Figure 2.3.

After extracting these last two entries from the list, the algorithm terminates. We see that the node created at the final step forms the root of the tree and the initial symbols **a**, **b**, **c** and **d** result in the leaves (nodes with no children) of the tree. The codeword for each symbol is determined by traversing the tree, starting at the root, until the leaf which corresponds to that symbol is reached. Along the way we concatenate the **0**s and **1**s labelling the branches on the specific path taken.

The resulting Huffman code for our example is given in Table 2.4, from which we see that the codeword lengths increase as the probability of the corresponding symbols decrease. Such a table is also called a Huffman table.

Our example character string can now be encoded as shown in Table 2.5, in other words, as **01001101110010**. The resulting bitrate is $\frac{14}{8} = 1.75$ bits/symbol, whereas, if the symbols were stored as ASCII characters, we would have needed 8 bits/symbol. Indeed,

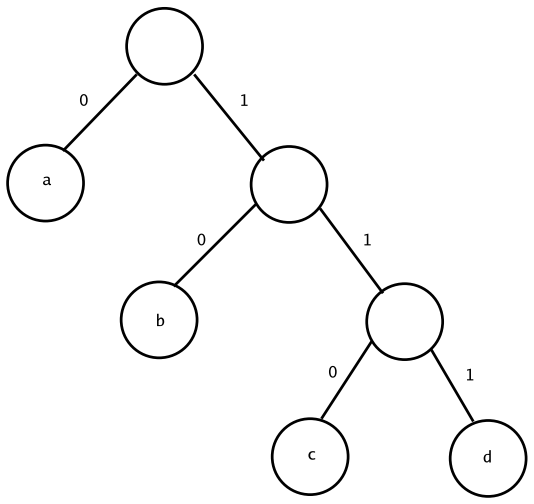


Figure 2.3: Completed Huffman tree, after third iteration of Huffman coding

Symbol	Probability	Huffman code
a	$\frac{1}{2}$	0
b	$\frac{1}{4}$	10
c	$\frac{1}{8}$	110
d	$\frac{1}{8}$	111

Table 2.4: Resulting Huffman code

since our alphabet in this example consists of only four symbols, a 2-bit fixed length code, for example, as in Table 2.6, could have been adopted. This would have yielded a bitrate of 2 bits/symbol, which would, however, still be less efficient than that achieved by the variable length code above.

Decoding is guaranteed to be unique since Huffman codes are part of a family of *prefix codes* — no codeword is a prefix of another codeword. The bits in the encoded file are repeatedly used to traverse the Huffman tree, starting at the root until a leaf is reached, with a **0** indicating that the left branch should be taken, and a **1** indicating that the right branch should be taken. Upon reaching a leaf, its corresponding symbol is placed in the decoded file. We repeat this process, starting at the root and continuing at the position

a	b	a	c	d	a	a	b
0	10	0	110	111	0	0	10

Table 2.5: The encoded character string

symbol	codeword
a	00
b	01
c	10
d	11

Table 2.6: A 2-bit fixed length code

immediately following in the encoded file, until the end of the encoded file is reached. Our compressed file thus decodes uniquely to **abacdaab**.

2.2 Terminology

The above example illustrates some of the basic concepts of data compression. Notice that the lengths of the (Huffman) codewords are $1 = -\log_2(\frac{1}{2})$ for **a**, $2 = -\log_2(\frac{1}{4})$ for **b**, and $3 = -\log_2(\frac{1}{8})$ for **c** and **d**. In data compression, the quantity

$$I = -\log_2(p) = \log_2\left(\frac{1}{p}\right)$$

is defined as the *information* (in bits) associated with a symbol occuring with probability p. This is the ideal codeword length of the symbol, the optimal number of bits required to encode the symbol. We see that this measure of conveyed information is well-defined: Symbols with small probabilities (conveying a large amount of information) result in large values of I, and symbols with large probabilities (conveying a small amount of information) have small values of I associated with them. In particular, for a symbol with a probability of 1, I is 0.

This concept of information corresponds intuitively to the degree of surprise one might feel in encountering symbols with various probabilities (the degree to which things are unpredictable and unexpected). In classical thermodynamics, *entropy* is used as a similar measure, measuring the degree of disorder of a particle system, and, as a result, data compression is also referred to as *entropy coding.*

In the case of data compression, the entropy H is defined as the average information per symbol. If there are n distinct symbols in the input file and p_i represents the probability of the ith symbol, then

$$H = \sum_{i=1}^{n} p_i \log_2 \left(\frac{1}{p_i}\right).$$

The entropy H is the ideal (optimal) bitrate achievable for a given input file; as stated in [1], "it provides a fundamental lower bound for the compression that can be achieved with a given alphabet of symbols. The entropy is therefore a very convenient measure of the performance of a coding system."

This brings us to a weakness of Huffman coding. Notice from our example that Huffman coding is always constrained to integer length codewords. The information of a symbol is an integer value only when the symbol's probability is an integer power of 2, and this is not the case in general. Had our character string been, for instance, **abacdaaaaaaaaab**, Huffman coding would once again have resulted in the Huffman tree in Figure 2.3. However, as we see from Table 2.7, the integer and ideal code lengths are not the same.

Symbol	Probability	Information (in bits)	Huffman code
a	$\frac{3}{4}$	0.415	0
b	$\frac{1}{8}$	3	10
c	$\frac{1}{16}$	4	110
d	$\frac{1}{16}$	4	111

Table 2.7: Comparison of integer and ideal code lengths

The resulting coding rate (for the Huffman code) is

$$R = (\tfrac{3}{4})(1) + (\tfrac{1}{8})(2) + (\tfrac{1}{16})(3) + (\tfrac{1}{16})(3) = 1.375\,\text{bits/symbol},$$

whereas the entropy is

$$H = (\tfrac{3}{4})(0.415) + (\tfrac{1}{8})(3) + (\tfrac{1}{16})(4) + (\tfrac{1}{16})(4) = 1.186\,\text{bits/symbol}.$$

Thus Huffman coding is prone to inefficiencies, especially when symbols occur with probabilities greater than 0.5, since such symbols would still require codewords of length 1, even though their ideal codeword length might be much less than 1. Huffman coding does, however, provide optimal <u>integer</u> length codes. For a proof of this, see [2].

2.3 Arithmetic Coding

In arithmetic coding the compressed data stream, or *code string*, as it is sometimes referred to, is interpreted as a binary fraction lying in the unit interval [0, 1]. The (distinct) symbols of the string to be compressed divide the interval proportional to their probabilities — each symbol is assigned a subinterval with width equal to its probability. In the case of our example string, **abacdaaaaaaaaaab**, the interval could be divided as shown in Figure 2.4.

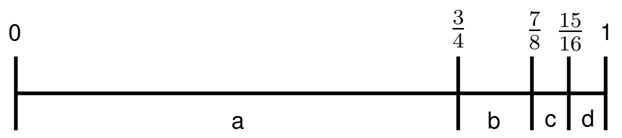


Figure 2.4: Possible partitioning of the unit interval

The lower limit of each subinterval is assigned to the corresponding symbol. Thus the subinterval $[0, \frac{3}{4})$ belongs to **a**, the subinterval $[\frac{3}{4}, \frac{7}{8})$ belongs to **b**, the subinterval $[\frac{7}{8}, \frac{15}{16})$ belongs to **c**, and the subinterval $[\frac{15}{16}, 1)$ belongs to **d**. This partitioning is reflected in

Symbol	Probability	Cumulative Probability
a	$\frac{3}{4}$	0
b	$\frac{1}{8}$	$\frac{3}{4}$
c	$\frac{1}{16}$	$\frac{7}{8}$
d	$\frac{1}{16}$	$\frac{15}{16}$

Table 2.8: Cumulative probabilities corresponding to a possible partitioning

Table 2.8; notice that the lower limit of each subinterval corresponds to the cumulative probability for the corresponding symbol.

Other orderings for the partitioning are equally valid — we could also have ordered our symbols as shown in Table 2.9, which leads to the partitioning shown in Figure 2.5. Here we will stick to the first ordering.

Symbol	Probability	Cumulative Probability
d	$\frac{1}{16}$	0
b	$\frac{1}{8}$	$\frac{1}{16}$
a	$\frac{3}{4}$	$\frac{3}{16}$
c	$\frac{1}{16}$	$\frac{15}{16}$

Table 2.9: Cumulative probabilities corresponding to another possible partitioning

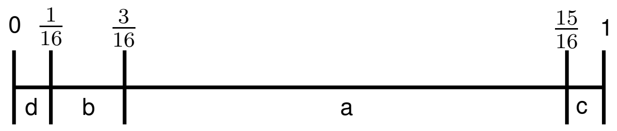


Figure 2.5: Another possible partitioning of the unit interval

Each time a symbol is encoded, its subinterval is chosen as the new "current interval" and is subdivided into the same proportions as the original interval. Thus after the occurence of the symbol **a** in our example string, our new current interval is [0, $\frac{3}{4}$), then, after

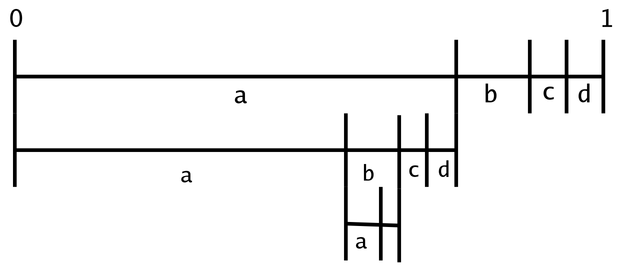


Figure 2.6: Successive subdivisions of the unit interval

encoding **b**, our current interval is $[\frac{9}{16}, \frac{21}{32})$, etc. This is illustrated in Figure 2.6. Each encoding of a symbol constitutes an iteration of the algorithm. During each iteration we therefore have to keep track of, and update, our current interval. We do so by defining two variables: C, the *code point*, is the lower limit of the current interval, and A is the width of the current interval. Let p_i denote the probability and P_i the cumulative probability of the current (ith) symbol to be encoded. Then each iteration of the algorithm is defined by the formulas

$$C := C + (A \times P_i),$$

$$A := A \times p_i.$$

Initially, we always have $C = 0$ and $A = 1$, denoting the entire unit interval. Encoding our example string **abacdaaaaaaaaaab** is then as in Table 2.10.

	Symbol to encode	Encoding
first	**a**	$C := 0 + (1)(0) = 0$ $A := (1)(\frac{3}{4}) = \frac{3}{4}$
second	**b**	$C := 0 + (\frac{3}{4})(\frac{3}{4}) = \frac{9}{16}$ $A := (\frac{3}{4})(\frac{1}{8}) = \frac{3}{32}$
third	**a**	$C := \frac{9}{16}$ $A := (\frac{3}{32})(\frac{3}{4}) = \frac{9}{128}$

fourth	**c**	$C := \frac{9}{16} + (\frac{9}{128})(\frac{7}{8}) = \frac{639}{1024}$ $A := (\frac{9}{128})(\frac{1}{16}) = \frac{9}{2048}$
fifth	**d**	$C := \frac{639}{1024} + (\frac{9}{2048})(\frac{15}{16}) = \frac{20583}{32768}$ $A := (\frac{9}{2048})(\frac{1}{16}) = \frac{9}{32768}$
sixth	**a**	$C := \frac{20583}{32768}$ $A := (\frac{9}{32768})(\frac{3}{4})$
$\vdots$	$\vdots$	$\vdots$
fifteenth	**a**	$C := \frac{20583}{32768}$ $A := (\frac{9}{32768})(\frac{3}{4})^{10}$
sixteenth	**b**	$C := \frac{20583}{32768} + (\frac{9}{32768})(\frac{3}{4})^{11} = \frac{86332953555}{137438953472}$ $A := (\frac{9}{32768})(\frac{3}{4})^{10}(\frac{1}{8}) = \frac{531441}{274877906944}$

Table 2.10: Encoding the string **abacdaaaaaaaaaab**

Our resulting interval is $[C,\, C + A) = [0.628154910...,\, 0.628156844...)$. As we will see, decoding is done by magnitude comparison: examining the code string to determine the interval in which it lies. As code string, any value greater than or equal to C and less than $C + A$ can be chosen to identify the final interval $[C,\, C + A)$. Notice that as more and more symbols are coded, our interval becomes smaller and smaller, and thus more and more bits of precision are required to represent a number in that interval — this is how our output file grows as more and more symbols are coded. The value requiring the smallest number of bits to represent the interval $[C,\, C + A)$ is chosen; in this case, that value is

$$0.1010000011001110111_2$$

$$= 2^{-1} + 2^{-3} + 2^{-9} + 2^{-10} + 2^{-13} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-18} + 2^{-19}$$

$$= \tfrac{329335}{524288} = 0.6281566619873046875$$

The code string is conventionally written as

$$.1010000011001110111$$

and the output file is **1010000011001110111**, which consists of 19 bits.

Thus arithmetic coding outperforms Huffman coding, achieving a coding rate of $\frac{19}{16} =$ 1.1875 bits/symbol, whereas Huffman coding would have given $\frac{22}{16} = 1.375$ bits/symbol. Notice also that arithmetic coding has come very close to the entropy $H = 1.186$ bits/symbol for our character string. This is in general the case: as the length of a file to be compressed increases, the coding rate achieved by arithmetic coding converges to the entropy for that file.

Decoding is done by iterating the following 3 steps [3]:

1. Examine the code string and determine the interval in which it lies. Decode the symbol corresponding to that interval.
2. Subtract the cumulative probability of the decoded symbol from the code string.
3. Rescale the code string by undoing the multiplication for the value A: Divide the code string by the probability of the decoded symbol.

Thus, examining the code string $.1010000011001110111_2 = \frac{329335}{524288}$, we see that it lies in the interval for symbol **a**: $[0, \frac{3}{4})$. We therefore decode symbol **a** and subtract 0 from the code string. The encoder multiplied A with $\frac{3}{4}$, therefore we now multiply the code string with $\frac{4}{3}$, to get

$$\text{code string} := (\tfrac{329335}{524288})(\tfrac{4}{3}) = \tfrac{329335}{393216}.$$

Subsequent iterations of the decoding are shown in Table 2.11.

	Symbol	Code string after step 2	Code string after step 3
second	**b**	$\frac{329335}{393216} - \frac{3}{4} = \frac{34423}{393216}$	$(\frac{34423}{393216})(8) = \frac{34423}{49152}$
third	**a**	$\frac{34423}{49152} - 0 = \frac{34423}{49152}$	$(\frac{34423}{49152})(\frac{4}{3}) = \frac{34423}{36864}$
fourth	**c**	$\frac{34423}{36864} - \frac{7}{8} = \frac{2167}{36864}$	$(\frac{2167}{36864})(16) = \frac{2167}{2304}$
fifth	**d**	$\frac{2167}{2304} - \frac{15}{16} = \frac{7}{2304}$	$(\frac{7}{2304})(16) = \frac{7}{144}$
sixth	**a**	$\frac{7}{144} - 0 = \frac{7}{144}$	$(\frac{7}{144})(\frac{4}{3})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
fifteenth	**a**	$(\frac{7}{144})(\frac{4}{3})^9 - 0 = (\frac{7}{144})(\frac{4}{3})^9$	$(\frac{7}{144})(\frac{4}{3})^9 = (\frac{7}{144})(\frac{4}{3})^{10}$
sixteenth	**b**		

Table 2.11: Decoding the string **abacdaaaaaaaaaab**

The decoded string is now **abacdaaaaaaaaaab**. We would like decoding to terminate at this point, so to make sure the decoder stops at the desired time, we can do one of the following things:

1. Specify at the beginning of the output file the number of symbols the uncompressed file consists of. After decoding that number of symbols, the decoder terminates.

2. Introduce a special End-Of-File (EOF) symbol into our alphabet of symbols and append it to the string before encoding, updating the table of probabilities accordingly. Upon decoding the EOF symbol, the decoder stops.

Both of these approaches lead to a slight degradation in the coding performance, but in either case the inefficiency tends to zero as the length of the string to be compressed increases.

As a result of arithmetic coding attaining coding rates that are arbitrarily close to the ideal coding rates, some image compression techniques exclusively use arithmetic coding. Examples of such techniques are the EZW algorithm, presented in Chapter 4, used in wavelet image compression, and image compression with WFA (see Chapter 6); both of these rely on the entropy bound as an accurate estimate of the compression that will be achieved, and make decisions based thereon during encoding.

2.4 A Final Note On Huffman Coding

The impression should not be that Huffman coding is obsolete. To remedy the cases where Huffman coding performs poorly, namely when a single symbol appears with probability far greater than 0.5, it is combined with *runlength coding*. Since long "runs" of the symbol in question will occur in such cases, runlength coding introduces special symbols that are used to designate such runs. For instance, a run of 10 zeros will be encoded with some or other symbol designating a run of 10 zeros, rather than encoding each zero separately with its own codeword. This results in significant improvements in compression performance, and according to experiments the improved performance comes very close to the performance achieved by arithmetic coding. See, for instance, the comparison of results in [1, Chapter 15].

Chapter 3

JPEG and Vector Quantization

3.1 JPEG

The JPEG standard defines various modes of operations for the encoding of an image, namely lossless mode, and various lossy modes, namely sequential, progressive and hierarchical mode. Hierarchical mode will not be discussed here; for details on this mode of operation, see [1]. The lossy modes, described in Section 3.1.2, make use of the discrete cosine transform (DCT), whereas the lossless mode, described in the next section, utilizes what is known as predictive coding.

3.1.1 Lossless Mode

The simplest, and also poorest, form of lossless compression is to simply feed the pixel values of an image to an entropy coder. This is known as *pulse code modulation* (PCM). However, neighbouring pixels in a continuous tone image tend to be very correlated and are often equal. The differences between such values are therefore usually very small, and in many cases zero. Figure 3.1 shows a typical histogram of the differences between each pixel and its left neighbour in an image, from which we see that the differences cluster around zero. The image used was the 512×512 image of Lenna (A 256×256 version is shown in Appendix B).

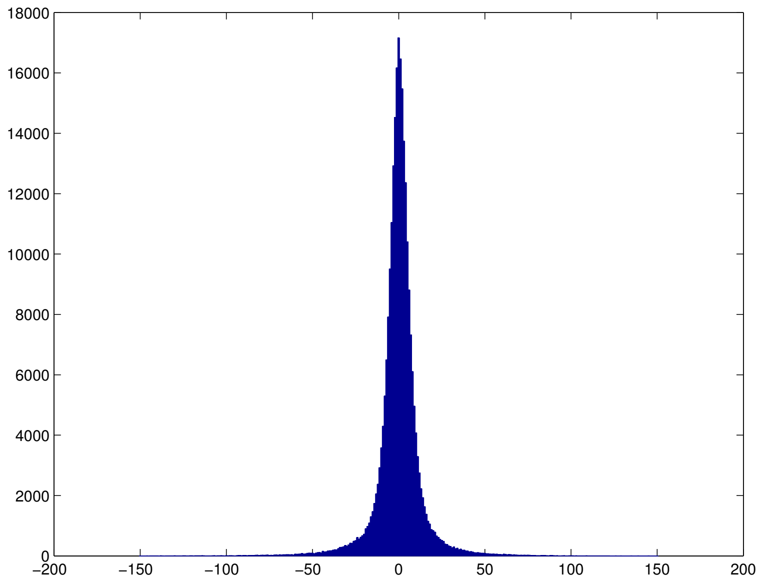


Figure 3.1: Histogram of differences from a sample image

Thus, instead of encoding samples directly, we could encode the differences between sample values, which leads to much better results. This form of coding is known as *differential pulse code modulation* (DPCM). A schematic view of the DPCM encoder and decoder models are shown in Figures 3.2 and 3.3, respectively.

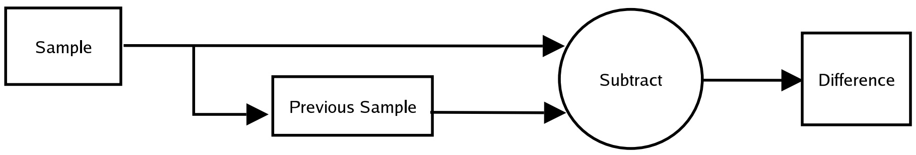


Figure 3.2: DPCM encoder

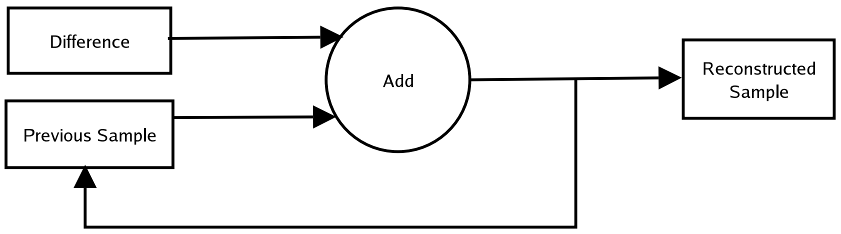


Figure 3.3: DPCM decoder

DPCM is a form of *predictive coding*: The previous sample is used as the prediction for the current sample. In predictive coding, in general, a combination of samples already coded can be used as a prediction for the sample to be coded. Typically, an average of the samples immediately above and to the left is used as predictor. Consider the neighbouring samples shown in Figure 3.4, where sample x is the current sample (sample to be coded), and the neighbouring samples that have already been coded are a, b and c.

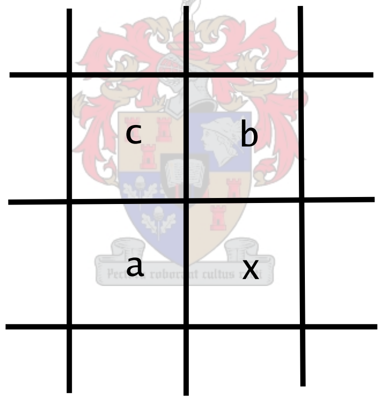


Figure 3.4: Neighbouring samples

JPEG lossless mode defines the list of predictors shown in Table 3.1 below.

For the first line of an image the neighbouring sample a is always used as predictor (selection value 1), and at the start of each subsequent line sample b (selection value 2) is always used. The predictor giving the best performance differs from one image to another, and it is up to the user to instruct the JPEG encoder which predictor to use.

Selection Value	Prediction
0	Reserved for Hierarchical Mode
1	a
2	b
3	c
4	a + b - c
5	a - (b - c)/2
6	b - (a - c)/2
7	(a + b)/2

Table 3.1: JPEG lossless predictors

3.1.2 Lossy Modes

The JPEG sequential and progressive modes of operation make use of the model of transform coding illustrated in Figure 3.5.

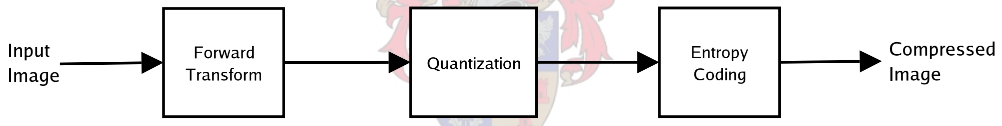


Figure 3.5: Model of transform coding

Forward Transform

Here the forward transform is a 2D discrete cosine transform (DCT). The image is divided into non-overlapping 8×8 blocks, with the 2D DCT applied to each block. If the rows and/or columns are not divisible by 8, padding is done by replicating the last row and/or column the required number of times. At worst, 7 rows and 7 columns would be added.

Given an 8×8 matrix A, the 2D forward and inverse DCTs are defined by

$$\widehat{A}_{x,y} = \frac{C(x)}{2}\frac{C(y)}{2}\sum_{i=1}^{8}\sum_{j=1}^{8} A_{i,j} \cos\left[\frac{\pi(x-1)(2i-1)}{16}\right] \cos\left[\frac{\pi(y-1)(2j-1)}{16}\right],$$

$$A_{i,j} = \sum_{x=1}^{8}\frac{C(x)}{2}\sum_{y=1}^{8}\frac{C(y)}{2}\widehat{A}_{x,y} \cos\left[\frac{\pi(x-1)(2i-1)}{16}\right] \cos\left[\frac{\pi(y-1)(2j-1)}{16}\right],$$

where

$$C(s) = \begin{cases} \frac{1}{\sqrt{2}}, & s = 1, \\ 1, & s > 1. \end{cases}$$

Just like discrete Fourier transforms (DFTs), DCTs are computationally expensive, as can be seen from the equations above, and just like DFTs, fast algorithms have been developed to implement DCTs. Numerous algorithms, in fact, exist for the implementation of 8×8 2D DCTs, resulting in significant improvements in execution time.

The output values of the DCT are known as DCT coefficients — they are the coefficients of a linear combination of discretely sampled 2D cosine functions of increasing frequency, referred to as DCT basis functions. These functions are shown in Figure 3.6. The DCT coefficient corresponding to the basis function in the upper left corner (which is the only constant basis function) is called the DC coefficient; the other coefficients are called AC coefficients.

Quantization

Figure 3.7 (taken from [1]) shows how the sensitivity of the human eye to chrominance and luminance intensities varies with spatial frequency. According to this, we see that the DCT coefficients corresponding to higher frequencies are "less important" than those corresponding to lower frequencies (for both chrominance and luminance components). Colour images are therefore first converted from the RGB colour system to the $\mathrm{YC_bC_r}$ chrominance-luminance colour system before each component is divided into 8×8 blocks for the DCT, and the DCT coefficients are then *quantized*: rescaled and rounded off to integer values. This provides the second step in transform coding.

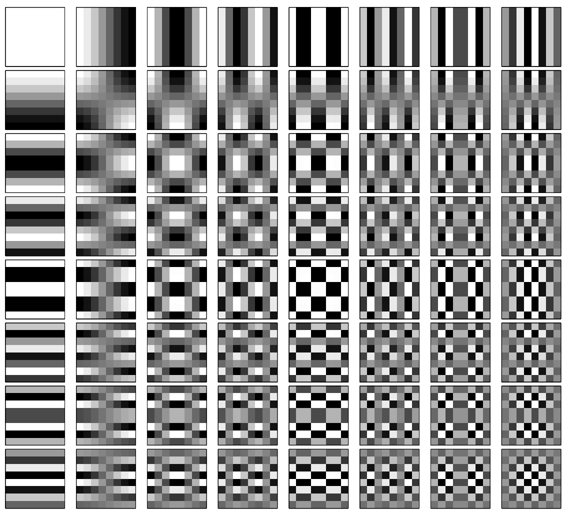


Figure 3.6: DCT basis functions

Tables 3.2 and 3.3 show the quantization tables defined by JPEG for luminance and chrominance, respectively. These values were determined by measuring the thresholds for visibility of the DCT basis functions, in other words, measuring for each basis function the amplitude (coefficient value) that is just detectable by the human eye. Each DCT coefficient is then divided by its corresponding quantization value and rounded off to the nearest integer value. Dequantization is done by multiplying the quantized coefficients with their corresponding quantization table entries.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table 3.2: Luminance quantization table

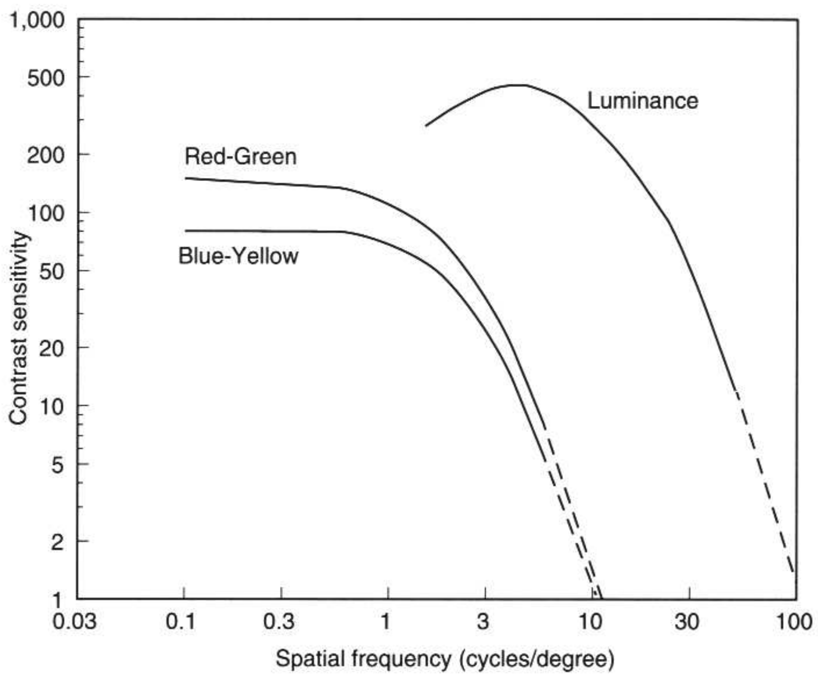


Figure 3.7: Contrast sensitivity functions [1]

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Table 3.3: Chrominance quantization table

Note that not only does quantization map a large number of DCT coefficients to zero, but, as a result of rounding off, those that are not mapped to zero map to values that are more correlated than their unquantized counterparts. For instance, if a quantization table entry is 4, then all applicable DCT coefficients in the interval [2, 6) will be mapped to 1; all applicable DCT coefficients in the interval [6, 10) will be mapped to 2; etc. The larger the quantization values become, the wider these corresponding intervals become, i.e. the more correlated the quantized DCT coefficients become. Thus, by uniformly rescaling the quantization tables, the quality versus compression tradeoff can be manipulated: Multiplying a quantization table by a number greater than 1 (and taking its ceiling since quantization values are always integers), results in larger entries and thus more correlated quantized DCT coefficients. This gives a higher compression ratio, but a poorer approximation to the original image, due to the quantization doing more "damage". Dividing a quantization table by a number greater than 1 (and taking its ceiling), results in smaller entries and less correlated quantized DCT coefficients. This results in a smaller compression ratio, but a better approximation to the original image, due to quantization doing less "damage".

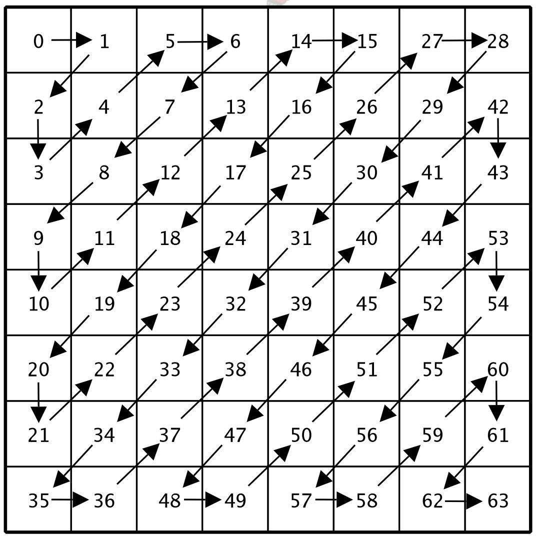


Figure 3.8: Zigzag sequence

The quantized DCT coefficients are fed to the entropy coder, which is either a Huffman or an arithmetic coder, in *zigzag* sequence, shown in Figure 3.8. As can be seen, this results in the lower frequencies occuring earlier and the higher frequencies occuring later in the sequence. The reason for this ordering is the following: Notice, firstly, that DCT coefficients usually rapidly tend to zero as their frequencies increase. Secondly, quantization table entries are larger for the higher frequency coefficients and smaller for the lower frequency coefficients, in other words, quantization table entries increase with vertical and horizontal frequency. Since quantization involves dividing by the values and then rounding off, a large number of quantized DCT coefficients will be zero. As a result, the quantized coefficients in the zigzag ordering form an approximately monotonic decreasing sequence, with the last entries all being zero (forming a run of zeros). This ensures that JPEG encoders using Huffman coding with runlength coding achieve good compression results.

When it comes to feeding the quantized coefficients to the entropy coder, we distinguish between sequential coding (JPEG sequential mode) and progressive coding (JPEG progressive mode). In general, we refer to the output of the quantization component as *image descriptors*.

Sequential Lossy Mode

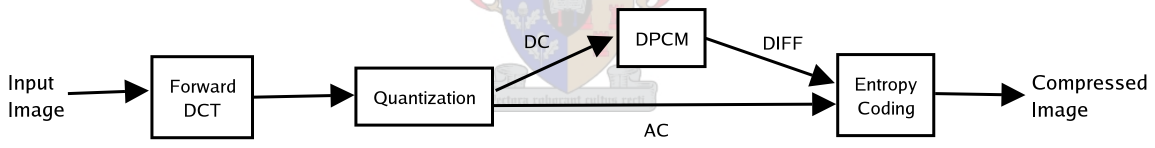


Figure 3.9: JPEG sequential mode encoder

The term sequential coding refers to the compression of image descriptors (in this case, DCT coefficients) in a single scan or pass. For sequential lossy mode we slightly refine our diagram of transform coding to that of Figure 3.9. Instead of coding the DC coefficients directly, we can improve our compression results by DPCM coding them. The DC coefficient from the previous 8×8 block (from the same component) is used as predictor for the current DC coefficient (DC coefficient to be coded). AC coefficients are coded directly. The decoding process is illustrated in Figure 3.10 below.

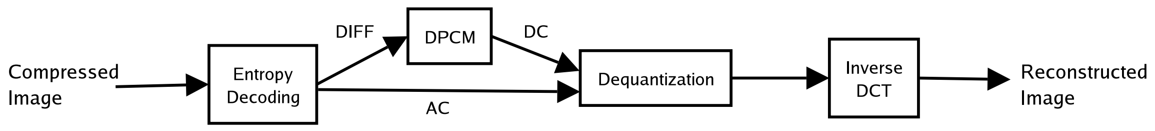


Figure 3.10: JPEG sequential mode decoder

A special restricted form of JPEG sequential lossy mode is the JPEG *baseline* mode. This mode only uses Huffman coding for the entropy coder, and is restricted to fixed Huffman tables — tables that were designed using a wide range of images [1]. Using these tables when encoding an image, rather than designing custom tables, leads to a significant reduction in encoding time. This does, however, lead to slightly worse compression results since these fixed tables are in general not the optimal Huffman tables for any specific image.

Progressive Lossy Modes

Progressive coding entails the encoding of image descriptors (in this case, DCT coefficients) in a sequence of scans or passes. Each of these passes are encoded separately, and are thus decoded separately, so that each pass can be displayed during decoding, yielding a progression of images, each one of better quality than the previous one. Not only is this useful to certain applications, but in many cases slightly better compression results can be achieved than with sequential coding. JPEG defines two progressive coding techniques: spectral selection and successive approximation.

Spectral Selection

In spectral selection, the zigzag sequence of DCT coefficients is segmented into contiguous bands, or *spectral* bands, and each band is encoded in a separate scan. The lower frequency bands are usually encoded first. The diagram in Figure 3.9 is once again applicable; in the case of sequential coding the entropy coding component is invoked only once, whereas here it is invoked multiple times.

Successive Approximation

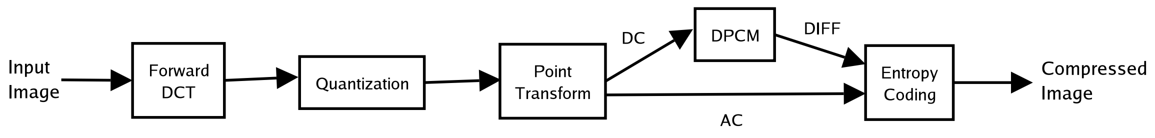


Figure 3.11: First stage of successive approximation

In the first scan of successive approximation, the n most significant bits of the DCT coefficients are coded (for some positive integer n), in other words, the coefficients are coded at reduced precision. This is done by dividing them by 2^n, which is referred to as a *point transform*. This is illustrated by Figure 3.11 above. DC coefficients are once again DPCM coded (after being point transformed). For each subsequent scan the value of n is decremented by 1, with the point transform once again being applied to the AC coefficients, improving their precision with each scan. On the other hand, the remaining lower order bits of the DC coefficients are sent one at a time in these subsequent scans, from the most significant to the least significant bit of the remaining bits. These bits are not DPCM coded, but rather coded directly, because they are almost completely random. Upon n reaching 0 the successive approximation progression is complete: the coefficients are at full precision.

Mixing Spectral Selection and Successive Approximation

Spectral selection and successive approximation may be mixed: spectral selection scans can be applied within each stage of successive approximation, or successive approximation scans can be applied within each stage of spectral selection. Mixing these two techniques results in a very graceful progression.

3.2 Vector Quantization

Vector quantization (VQ) is a very simple lossy compression technique, with very fast encoding and decoding times, applicable to both image and audio compression. See, for example, [4].

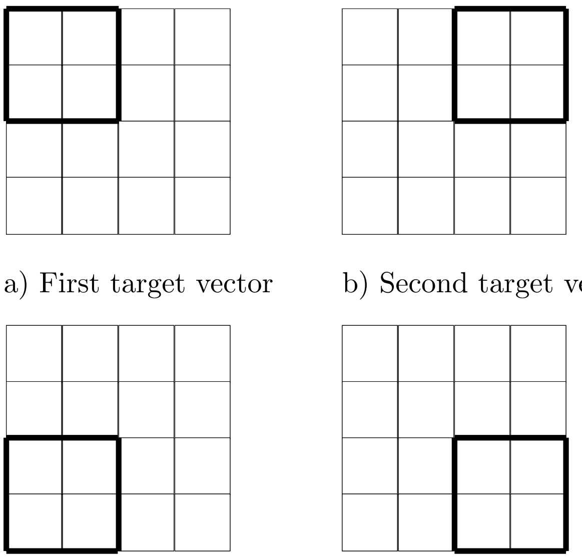
3.2.1 Vector Quantization of Images

Assume we have a set of, say M, $m \times n$ matrices, with m and n small, known as a *codebook*. Each matrix in our codebook is referred to as a *codevector*. A grayscale image to be compressed is partitioned into non-overlapping $m \times n$ blocks, each referred to as a *target vector*. For each target vector a search is performed through the codebook to find the codevector which is "closest" to that target vector. For images the Euclidean metric is used: An $m \times n$ matrix is reshaped to a vector with mn entries. The Euclidean distance between two such vectors $\mathbf{x}$ and $\mathbf{y}$ is then

$$d(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2 = \sqrt{\sum_{k=1}^{mn} (x_k - y_k)^2}.$$

Codevectors are numbered from 1 to M, and once the codevector closest to the target vector in question is found, its number is placed in an array. Thus, the output file is an array of indices indicating which codevector is closest to each target vector. The encoding is typically done from left to right, top to bottom.

Consider, for instance, the 4×4 matrix in Figure 3.12 as the image to be encoded, given a codebook consisting of eight 2×2 codevectors. Our image therefore consists of four target vectors. The encoding is done in the sequence indicated in these figures, each time searching through the codebook for the codevector closest to the particular target vector. Our resulting output array looks as shown in Table 3.4, where the ith codevector is closest to the first target vector, the jth codevector is closest to the second target vector, etc. Since there are eight codevectors, each index can be represented by a 3-bit binary number, and since each vector consists of four elements, we have a resulting bitrate of $\frac{3}{4}$ bits per



a) First target vector b) Second target vector

c) Third target vector d) Fourth target vector

Figure 3.12: Example image

i	j	k	l

Table 3.4: Example output array

pixel. In general, if we have a codebook consisting of M codevectors and each codevector has s entries, then the bitrate is given by the formula

$$r = \frac{\lceil \log_2 M \rceil}{s} \text{ bits/pixel.}$$

Since we take the ceiling of $\log_2 M$, to calculate the number of bits needed to represent a codevector, we could just as well choose M to be a positive integer power of 2, to fully utilize that number of bits. Hence, VQ codebooks always contain 2^n codevectors, for some positive integer n. Our formula for the bitrate is now

$$r = \frac{n}{s} \text{ bits/pixel.}$$

The larger the value of n, and thus the more elements we have in our codebook, the better we can approximate each target vector and the higher the quality of the decoded image will be. This results in a higher bitrate r and thus a smaller compression ratio. Conversely, the smaller the value of n, the worse our approximating codevectors will perform in general, and the poorer the quality of the decoded image will be. This gives a lower bitrate and thus a higher compression ratio. To decode an image, we simply "walk" through the output array, using each entry to index a codevector in the codebook, and "paste" the codevectors together to form a decoded image.

3.2.2 Traditional Codebook Design

The more difficult part of vector quantization is designing the codebook. We wish to design the codebook so as to minimize the average distortion (error) resulting from encoding images. In other words, for a sequence of images X_1, X_2, …, X_n, we would like the longterm average

$$\lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} d(X_i, \widehat{X}_i)$$

to be small, assuming that the limit exists, where $\widehat{X}_i, i = 1, \ldots, n$ are the reconstructed images corresponding to the images $X_i, i = 1, \ldots, n$. The approach taken is to use long training sequences and design a codebook that minimizes the average distortion for the training sequence. The codebook is then tested on images outside the training set, in the hope that the results will be reasonably close to those obtained with the training set. If so, the codebook is accepted. If, on the other hand, the results differ significantly, the training sequence was probably not long enough, and a new codebook is designed based on a longer training sequence. Before designing a codebook, one decides on the dimensions of the codevectors (and thus also the target vectors), for instance 4×4, see [5] or 4×3, see [4]; let it be $m \times n$. The training set (set of training vectors) are then all non-overlapping $m \times n$ submatrices from a handful (e.g. five) of training images. One of course also has to decide on the number of codevectors that the codebook should consist of.

Algorithm 3.1, due to Linde, Buzo and Gray [4], is traditionally used during the construction of a codebook.

```
Input: A training sequence and an initial codebook
1. Encode the training sequence with the current codebook. If the average
   distortion is small enough, quit;
2. Replace each codevector with the centroid of all the training vectors which were
   mapped to it (were closest to it). Go to 1.;
```

Algorithm 3.1: LBG algorithm

If N vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ were mapped to a certain codevector, their centroid is given by

$$\overline{\mathbf{x}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i.$$

The algorithm iteratively improves the initial codebook. A typical iteration of the algorithm is illustrated in Figures 3.13 and 3.14, where the vectors are for simplicity 2-dimensional. The codevectors are represented by ‘o’s and the training vectors by ‘x’s. In Figure 3.13 the training vectors in the first quadrant are closest to the codevector in the first quadrant and will thus be mapped to it. Similarly, the training vectors in the second quadrant will be mapped to the codevector in the second quadrant, etc. In Figure 3.14 the old codevectors have been replaced by the centroids of the training vectors that mapped to them (in Figure 3.13). Thus, if we now encoded the training vectors with these new codevectors, the average distortion would decrease. The average distortion decreases monotonically with each iteration, and we terminate the algorithm when such a decrease falls below some threshold.

There are various ways of acquiring an initial codebook. The simplest approach is to either take the first 2^R, or some randomly chosen 2^R, training vectors from the training set, where 2^R is the desired number of elements for the codebook. Another approach is known as *splitting*, where we first calculate the centroid of the entire training sequence — yielding the optimum single-element codebook for that training sequence. This codevector is then split: an additional codevector is formed from it by adding a small amount of “noise”

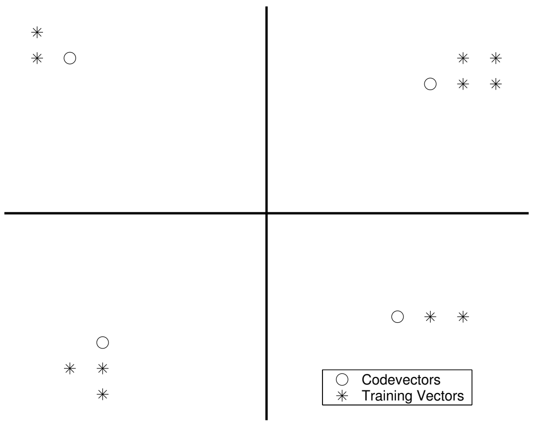


Figure 3.13: First step in an iteration of the LBG algorithm

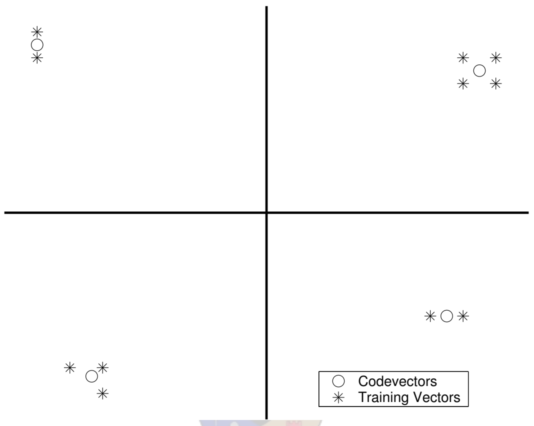


Figure 3.14: Second step in an iteration of the LBG algorithm

to it, thus perturbing it. This gives an initial codebook for the LBG algorithm, which is then run to obtain an optimal 2-element codebook. This process continues until the desired optimal 2^R-element codebook is obtained. Figure 3.15 illustrates the first 2 stages in such a splitting procedure, resulting in an optimal 4-element codebook.

The LBG algorithm is, however, expensive in terms of computation and storage requirements, both of which increase exponentially with the codebook size and dimension of the vectors. Many algorithms that are variations of the LBG algorithm have been developed to reduce the computational compexity in exchange for small decreases in image quality;

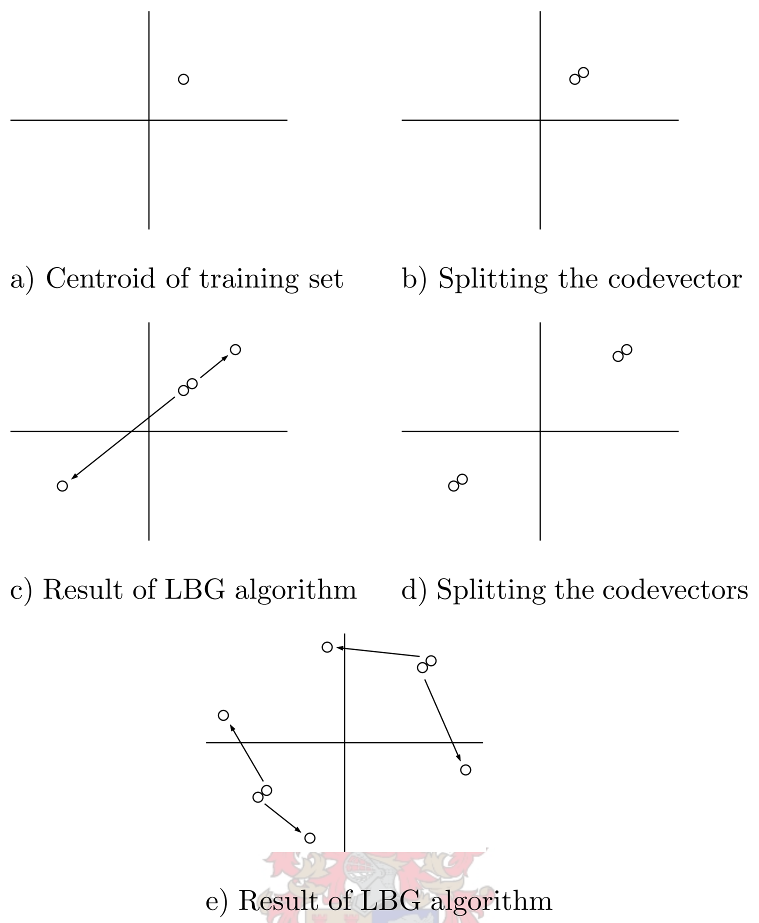

a) Centroid of training set
b) Splitting the codevector
c) Result of LBG algorithm
d) Splitting the codevectors
e) Result of LBG algorithm

Figure 3.15: Splitting technique

for details on some of these, see [4]. The technique that we will discuss next is a more recent codebook design algorithm and is not a variation of the LBG algorithm. It was published by Chaur-Heh Hsieh [5], requires no intial codebook, and is considerably faster than the algorithms that improve on the LBG algorithm. The resulting codebook can also be used as an initial codebook for the LBG algorithm, if one wishes to do so.

3.2.3 DCT-based Codebook Design

This technique makes use of the 2D DCT: Training images are divided into non-overlapping $m \times m$ training vectors, then features are extracted from the training vectors by applying a 2D DCT to each, with the DCT coefficients stored in a 1D array by following the zigzag order illustrated in Figure 2.8. Each DCT coefficient is regarded as a *feature* of its training vector, and the 1D vector as the *feature vector*. The codebook is now constructed by partitioning the training vectors into a binary tree (this is a form of *binary tree classification*): Each node in the tree represents a subset of the entire set of training vectors, with all subsets at the same height being mutually exclusive and their union forming the entire training set. The root of the tree represents the entire set. At each node the mean and variance of the features of the vectors at that node are calculated. If there are N training vectors at node n with corresponding feature vectors $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N$ the mean $M(j)$ and variance $V(j)$ for the jth feature are defined as

$$M(j) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{z}_i(j),\ j = 1, \ldots, m^2,$$

$$V(j) = \frac{1}{N}\sum_{i=1}^{N} [\mathbf{z}_i(j) - M(j)]^2,\ j = 1, \ldots, m^2.$$

The feature, say the kth, with the largest variance is used as the *split key* for this node: The vectors falling on this node are partitioned by comparing their kth feature with a *split threshold*. Here we choose the mean of all the kth features at this node, $M(k)$, as the split threshold, so that we compare each feature vector's kth element with $M(k)$. Vectors whose kth feature is less than $M(k)$ are placed in the set corresponding to the node's left child, the others in the right child's set. An example of this is shown in Figure 3.16 where the training set consists of 12 vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{12}$. The partitioning operation is performed at node 1 (the root), node 2 and node 3, resulting in 4 leaf nodes (4, 5, 6 and 7).

The algorithm is repeated until the same number of leaves is obtained as the desired number of codevectors for the codebook. The centroid of each cluster of training vectors falling on a leaf is then calculated and used as a codevector for the codebook. In the case of our example in Figure 3.16, if we required 4 codevectors, our codebook would consist of the codevectors

$$\mathbf{X}_1 = \tfrac{1}{3}(\mathbf{x}_1 + \mathbf{x}_4 + \mathbf{x}_6),$$

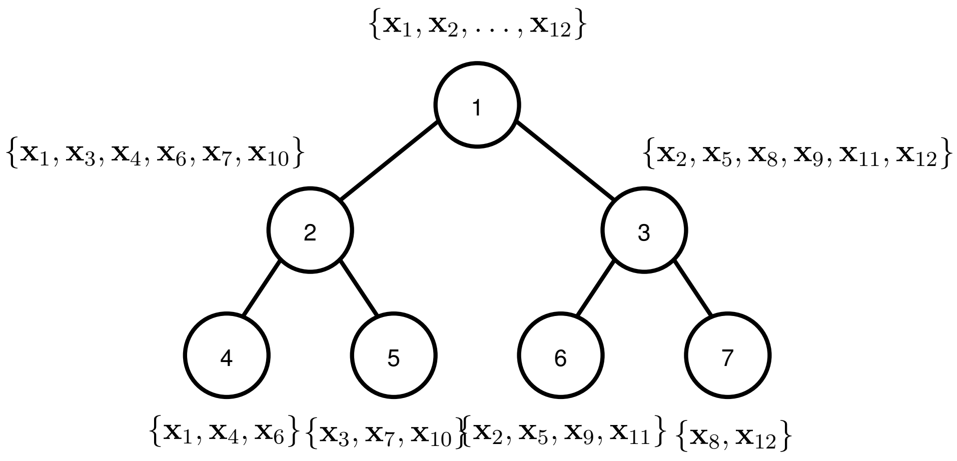

$\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{12}\}$
1
$\{\mathbf{x}_1, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_6, \mathbf{x}_7, \mathbf{x}_{10}\}$
$\{\mathbf{x}_2, \mathbf{x}_5, \mathbf{x}_8, \mathbf{x}_9, \mathbf{x}_{11}, \mathbf{x}_{12}\}$
2
3
4
5
6
7
$\{\mathbf{x}_1, \mathbf{x}_4, \mathbf{x}_6\}$
$\{\mathbf{x}_3, \mathbf{x}_7, \mathbf{x}_{10}\}$
$\{\mathbf{x}_2, \mathbf{x}_5, \mathbf{x}_9, \mathbf{x}_{11}\}$
$\{\mathbf{x}_8, \mathbf{x}_{12}\}$

Figure 3.16: Partitioning of training vectors

$$\mathbf{X}_2 = \tfrac{1}{3}(\mathbf{x}_3 + \mathbf{x}_7 + \mathbf{x}_{10}),$$

$$\mathbf{X}_3 = \tfrac{1}{4}(\mathbf{x}_2 + \mathbf{x}_5 + \mathbf{x}_9 + \mathbf{x}_{11}),$$

$$\mathbf{X}_4 = \tfrac{1}{2}(\mathbf{x}_8 + \mathbf{x}_{12}).$$

Chapter 4

Wavelet Image Compression

Like JPEG, the wavelet image compression technique discussed in this chapter follows the paradigm of transform coding of Figure 3.5, once again illustrated in Figure 4.1. Here the forward transform is a 2D discrete wavelet transform (DWT) and quantization is done with the embedded zerotree wavelet (EZW) algorithm, generating image descriptors over a 6-symbol alphabet. Once again, the image descriptors are entropy coded at the final stage, yielding the compressed image data. Arithmetic coding is used for the entropy coding, enabling us to calculate, during the execution of the EZW algorithm, the size of the output file and then stop the algorithm whenever we want, e.g. as soon as a target bitrate has been achieved.

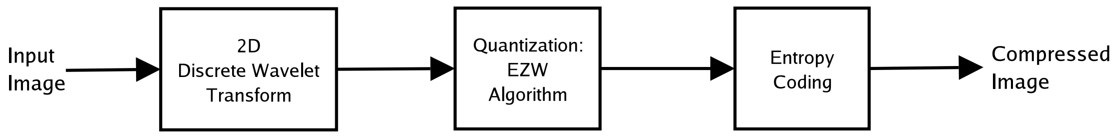


Figure 4.1: Transform coding for wavelet image compression

4.1 Discrete Wavelet Transform

We once again restrict our attention to grayscale images. The 2D DWT is applied a certain number of times to the input image, say n, where n is small, resulting in an n-level decomposition of the image. An example is shown in Figure 4.2, where we have a 2-level decomposition of the Lenna image, interpreted as an image.

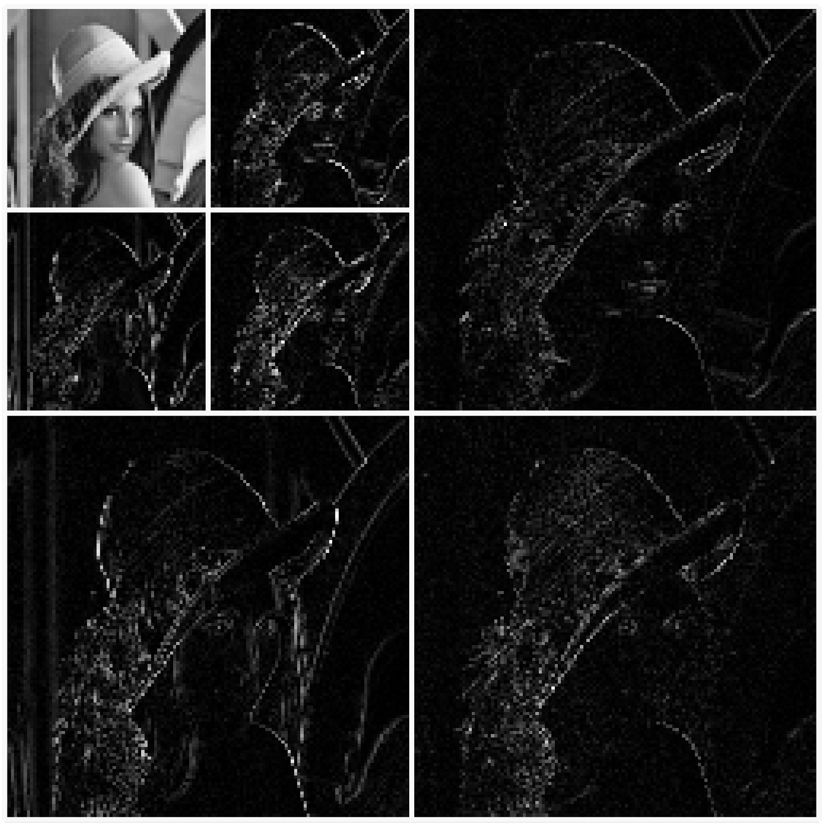


Figure 4.2: 2-level wavelet decomposition of Lenna

A 2D DWT is accomplished by first performing a 1D DWT to the rows and then the columns of the image (or any 2-dimensional signal for that matter). Before explaining a 1D DWT, we first introduce some terminology and notation [6].

The vector space M(ℤ) is defined as the set of all bi-infinite sequences, i.e.

$$M(\mathbb{Z}) = \{a = \{a_j : j \in \mathbb{Z}\} : a_j \in \mathbb{R}, j \in \mathbb{Z}\},$$

and the inner product space $\ell^2(\mathbb{Z})$ of square-summable bi-infinite sequences is defined by

$$\ell^2(\mathbb{Z}) = \left\{ c \in M(\mathbb{Z}) : \sum_{j\in\mathbb{Z}} c_j^2 < \infty \right\},$$

with corresponding inner product

$$\langle c, d\rangle_{\ell^2(\mathbb{Z})} = \sum_{j\in\mathbb{Z}} c_j d_j, \quad c, d \in \ell^2(\mathbb{Z}).$$

Then $\ell^2(\mathbb{Z})$ is a normed linear space with respect to the norm

$$||c||_{\ell^2(\mathbb{Z})} = \sqrt{\langle c, c\rangle_{\ell^2(\mathbb{Z})}} = \sqrt{\sum_{j\in\mathbb{Z}} c_j^2}, \quad c \in \ell^2(\mathbb{Z}).$$

The inner product space $L^2(\mathbb{R})$ of square-integrable functions on $\mathbb{R}$ is defined by

$$L^2(\mathbb{R}) = \left\{ f : \mathbb{R} \to \mathbb{R} : \int_{-\infty}^{\infty} [f(t)]^2 dt < \infty \right\},$$

with corresponding inner product

$$\langle f, g\rangle_{L^2(\mathbb{R})} = \int_{-\infty}^{\infty} f(t)g(t)dt, \quad f, g \in L^2(\mathbb{R}).$$

Then $L^2(\mathbb{R})$ is a normed linear space with respect to the norm

$$||f||_{L^2(\mathbb{R})} = \sqrt{\langle f, f\rangle_{L^2(\mathbb{R})}} = \sqrt{\int_{-\infty}^{\infty} [f(t)]^2 dt}, \quad f \in L^2(\mathbb{R}).$$

A sequence $a \in M(\mathbb{Z})$ is said to be *finitely supported* if there exist integers R and Q such that

$$a_j = 0, \ j \notin [R, Q].$$

A function ϕ is said to be *finitely supported* if there exist integers R and Q such that

$$\phi(t) = 0, \ t \notin [R, Q],$$

and *refinable* if

$$\phi(t) = \sum_{j\in\mathbb{Z}} a_j \phi(2t - j), \ t \in \mathbb{R}$$

for some finitely supported sequence $a \in M(\mathbb{Z})$. A scaling function ϕ is a (piecewise) continuous, finitely supported, refinable function also satisfying the *partition of unity* property

$$\sum_{j\in\mathbb{Z}} \phi(x-j) = 1,\ x \in \mathbb{R}.$$

Given a scaling function ϕ, we generate a sequence of vector spaces $\left\{V^{(r)} : r \in \mathbb{Z}\right\}$, with

$$V^{(r)} = \left\{\sum_{j\in\mathbb{Z}} c_j \phi(2^r \cdot -j) : c \in \ell^2(\mathbb{Z})\right\}, \quad r \in \mathbb{Z}.$$

These spaces are nested in the sense that

$$V^{(r)} \subset V^{(r+1)},\ r \in \mathbb{Z},$$

and we also have that

$$\bigcap_{r\in\mathbb{Z}} V^{(r)} = \{0\}.$$

Each space $V^{(r)} \subset L^2(\mathbb{R})$; moreover their union is dense in $L^2(\mathbb{R})$, i.e.

$$\overline{\bigcup_{r\in\mathbb{Z}} V^{(r)}} = L^2(\mathbb{R}).$$

As a result, we have

$$\{0\} \longleftarrow \ldots V^{(-1)} \subset V^{(0)} \subset V^{(1)} \subset \ldots \longrightarrow L^2(\mathbb{R}).$$

Such a nested sequence $\left\{V^{(r)} : r \in \mathbb{Z}\right\}$ of linear subspaces of $L^2(\mathbb{R})$ is called a *multiresolution analysis* (MRA) of $L^2(\mathbb{R})$, and is central to wavelet decomposition.

The *orthogonal wavelet spaces* corresponding to $\left\{V^{(r)} : r \in \mathbb{Z}\right\}$ are defined as the orthogonal complements of each $V^{(r)}$ with respect to $V^{(r+1)}$, in other words, for each $V^{(r)}$,

$$W^{(r)} = (V^{(r)})^{\perp} = \left\{f \in V^{(r+1)} : \langle f, g\rangle = 0,\ g \in V^{(r)}\right\},\ r \in \mathbb{Z}.$$

This leads to the following orthogonal decomposition result: if $f \in V^{(r+1)}$, then there exist unique functions $g \in V^{(r)}$ and $h \in W^{(r)}$ such that

$$f = g + h.$$

A finitely supported function ψ such that

$$W^{(r)} = \left\{ \sum_{j\in\mathbb{Z}} d_j \psi(2^r \cdot - j) : d \in \ell^2(\mathbb{Z}) \right\}, \ r \in \mathbb{Z}$$

is called a *wavelet*. It can be proven that the following decomposition result, fundamental to wavelet analysis, holds:

$$\phi(2^{r+1}t - j) = \sum_{k\in\mathbb{Z}} \alpha_{2k-j} \phi(2^r t - k) + \sum_{k\in\mathbb{Z}} \beta_{2k-j} \psi(2^r t - k), \quad t \in \mathbb{R}, \quad r, j \in \mathbb{Z},$$

where α and β are known as the *decomposition sequences*, and are uniquely determined by ϕ and ψ. Before implementing a wavelet decomposition on a signal, one thus first decides on a scaling function and corresponding wavelet.

Given a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, a 1D DWT is implemented as follows: The vector $\mathbf{x}$ can be converted to an element of $\ell^2(\mathbb{Z})$, say $c^{(N)} \in \ell^2(\mathbb{Z}), N \in \mathbb{N}$, with

$$c_j^{(N)} = \begin{cases} x_j, & j = 1, \ldots, n, \\ 0, & \text{otherwise} \end{cases}$$

where $c_j^{(N)}$ is interpreted as the discrete samplings of a function $f \in L^2(\mathbb{R})$ such that

$$f_N(t) = \sum_{j\in\mathbb{Z}} c_j^{(N)} \phi(2^N t - j), \ t \in \mathbb{R}$$

with

$$||f_N - f||_{L^2(\mathbb{R})} < \epsilon,$$

for some or other real $\epsilon > 0$. For example, f could be a function such that

$$c_j^{(N)} = f\left(\frac{j}{2^N}\right), \ j \in \mathbb{Z}.$$

Thus, f_N is an approximation to f from the space $V^{(N)}$, and, as a result of the density of the MRA, we can approximate f arbitrarily closely by increasing the value of N.

Therefore, due to the orthogonal decomposition result, we have

$$f_N = f_{N-1} + g_{N-1}, \quad f_{N-1} \in V^{(N-1)}, \quad g_{N-1} \in W^{(N-1)},$$

with

$$f_{N-1}(t) = \sum_{j\in\mathbb{Z}} c_j^{(N-1)}\phi(2^{N-1}t - j), \quad t \in \mathbb{R}$$

and

$$g_{N-1}(t) = \sum_{j\in\mathbb{Z}} d_j^{(N-1)}\psi(2^{N-1}t - j), \quad t \in \mathbb{R}.$$

The sequences $c^{(N-1)}$ and $d^{(N-1)}$ are given by the formulas

$$c_j^{(N-1)} = \sum_{k\in\mathbb{Z}} \alpha_{2j-k} c_k^{(N)}, \quad j \in \mathbb{Z},$$

$$d_j^{(N-1)} = \sum_{k\in\mathbb{Z}} \beta_{2j-k} c_k^{(N)}, \quad j \in \mathbb{Z},$$

because

$$\begin{aligned} f_N(t) &= \sum_{j\in\mathbb{Z}} c_j^{(N)}\phi(2^N t - j) \\ &= \sum_{j\in\mathbb{Z}} c_j^{(N)} \left[\sum_{k\in\mathbb{Z}} \alpha_{2k-j}\phi(2^{N-1}t - k) + \sum_{k\in\mathbb{Z}} \beta_{2k-j}\psi(2^{N-1}t - k)\right] \\ &= \sum_{k\in\mathbb{Z}} \left[\sum_{j\in\mathbb{Z}} \alpha_{2k-j} c_j^{(N)}\right] \phi(2^{N-1}t - k) + \sum_{k\in\mathbb{Z}} \left[\sum_{j\in\mathbb{Z}} \beta_{2k-j} c_j^{(N)}\right] \psi(2^{N-1}t - k), \quad t \in \mathbb{R}. \end{aligned}$$

This decomposition can be repeated an arbitrary number of times, say M, yielding the sequences of functions

$$\{f_r : r = N, N-1, \ldots, N-M\},$$

$$\{g_r : r = N-1, N-2, \ldots, N-M\},$$

with

$$f_{r+1} = f_r + g_r, \; r = N-1, N-2, \ldots, N-M$$

and

$$f_r(t) = \sum_{j\in\mathbb{Z}} c_j^{(r)}\phi(2^r t - j), \; t \in \mathbb{R}, \; r = N, N-1, \ldots, N-M$$

$$g_r(t) = \sum_{j\in\mathbb{Z}} d_j^{(r)}\psi(2^r t - j), \; t \in \mathbb{R}, \; r = N-1, N-2, \ldots, N-M$$

where

$$c_j^{(r)} = \sum_{k\in\mathbb{Z}} \alpha_{2j-k} c_k^{(r+1)},\ j \in \mathbb{Z},\ r = N-1, N-2, \ldots, N-M$$

$$d_j^{(r)} = \sum_{k\in\mathbb{Z}} \beta_{2j-k} c_k^{(r+1)},\ j \in \mathbb{Z},\ r = N-1, N-2, \ldots, N-M.$$

We also see that

$$f_r \in V^{(r)},\ r = N, N-1, \ldots, N-M,$$

$$g_r \in W^{(r)},\ r = N-1, N-2, \ldots, N-M,$$

and

$$f_N = f_{N-M} + g_{N-M} + g_{N-M+1} + \ldots + g_{N-2} + g_{N-1}.$$

The functions $g_{N-M}, g_{N-M+1}, \ldots, g_{N-1}$ are called the *wavelet components* of the function f_N which we have associated with our initial vector $\mathbf{x}$, and the sequences $d^{(N-1)}, d^{(N-2)}, \ldots, d^{(N-M)}$ are called the *wavelet coefficient sequences*. This decomposition algorithm can be illustrated diagrammatically as shown in Figure 4.3.

$$\begin{tikzcd}
 & & & & \boxed{f_N \in V^{(N)}} \arrow[dl] \arrow[dr] & \\
 & & & \boxed{f_{N-1} \in V^{(N-1)}} \arrow[dl] \arrow[dr] & & \boxed{g_{N-1} \in W^{(N-1)}} \\
 & & \boxed{f_{N-2} \in V^{(N-2)}} \arrow[dl] \arrow[dr] & & \boxed{g_{N-2} \in W^{(N-2)}} & \\
 & \vdots \arrow[dl] \arrow[dr] & & \boxed{g_{N-3} \in W^{(N-3)}} & & \\
\boxed{f_{N-M} \in V^{(N-M)}} & & \boxed{g_{N-M} \in W^{(N-M)}} & & &
\end{tikzcd}$$

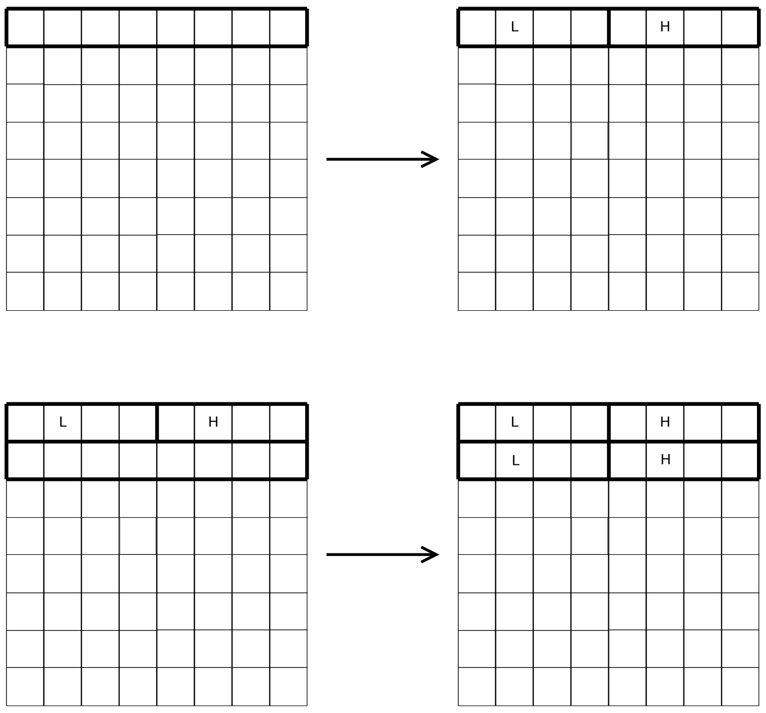
Figure 4.3: Diagram of an M-level wavelet decomposition

Reconstructing the sequence $c^{(N)}$, and thus also the function f_N, from the sequences $c^{(N-1)}, d^{(N-1)}, \ldots, c^{(N-M)}, d^{(N-M)}$ is done as follows: Once again two sequences a and q, uniquely determined by ϕ and ψ, exist, called the *reconstruction sequences.* From these sequences each $c^{(r+1)}, r = N-M, \ldots, N-1$ can be reconstructed by the formula

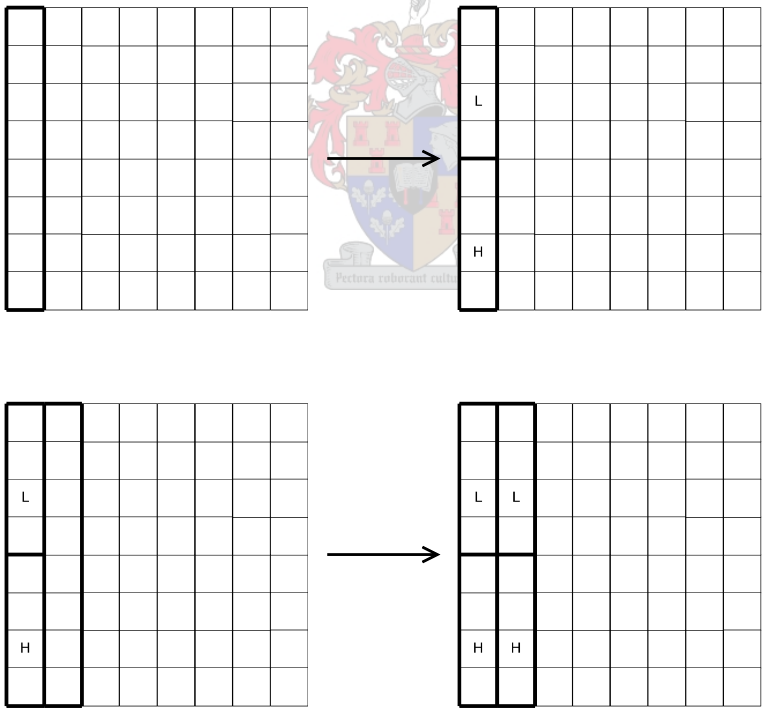
$$c_j^{(r+1)} = \sum_{k\in\mathbb{Z}} a_{j-2k} c_k^{(r)} + \sum_{k\in\mathbb{Z}} q_{j-2k} d_k^{(r)},\ j \in \mathbb{Z},\ r = N-M, \ldots, N-1.$$

The application of the decomposition sequences α and β to the sequences $c^{(r)}$ is equivalent to applying decomposition filters (high pass and low pass) to them and downsampling at each stage. Reconstruction is done by upsampling the output of the high pass decomposition filter and applying a high pass reconstruction filter to it, upsampling the output of the low pass decomposition filter and applying a low pass reconstruction filter to it, and adding these two resulting sequences. Such a process is known as using *filter banks* [7].

When using filter banks, we directly use the vector $\mathbf{x}$, with the downsampling of its decompositions leading to sequences of half the length of their predecessors. Decompositions can thus be done as long as the length of the sequence to be decomposed is divisible by 2. This does not pose a problem in wavelet image compression, since only about two or three levels of decomposition are typically done. If the rows and/or columns are not divisible by 2^r, where r is the number of decompositions to be done, in the worst case we would need to pad with $2^r - 1$ rows and columns. As noted before, a 2D DWT is applied to an image by first applying a 1D DWT to its rows and then a 1D DWT to its columns: Each row is replaced with the output rows of the low pass and high pass decomposition filters applied to it, placed next to each other (low pass left; high pass right). This is illustrated in Figure 4.4(a). Afterwards, each column of the resulting matrix is replaced with the output columns of the low pass and high pass decomposition filters applied to it, one placed on top of the other (low pass top; high pass bottom). This is illustrated in Figure 4.4(b). Subsequent decompositions are repeated on the lowest subband resulting from the previous decomposition. After two levels of decomposition, a matrix of the form shown in Figure 4.5 results. Compare this to Figure 4.2, where we have interpreted the decomposition as an image.



a) Replacing rows with their high and low pass components



b) Replacing columns with their high and low pass components

Figure 4.4: Applying highpass and lowpass filters

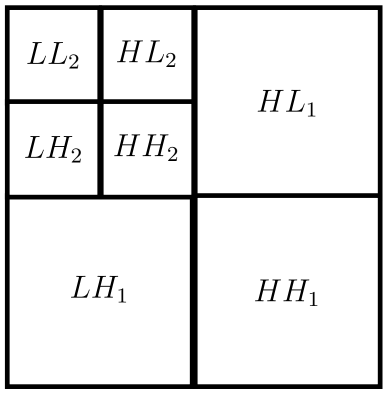


Figure 4.5: Subbbands resulting from a 2-level decomposition

4.2 Quantization: EZW Algorithm

The embedded zerotree wavelet (EZW) algorithm was published by Shapiro in [8] and is, as the name suggests, an embedded coding technique, which is synonymous with progressive coding (explained in Chapter 3.1.2). Here the wavelet coefficients are coded in order of importance, specifically according to precision, magnitude, scale and spatial location, as will be shown. The algorithm exploits the tendency of wavelet coefficients to decrease in magnitude from the lower subbands to the higher subbands. In order to do this a data structure is defined to order the coefficients in a tree structure: Every coefficient at a given scale can be related to a set of coefficients at the next finer scale of similar orientation. Consider the situation depicted in Figure 4.6. Each coefficient at scale (subband) LL_3 will have a descendant coefficient at scales HL_3, LH_3 and HH_3 — three descendants in total. From there on each coefficient has four descendants at the next finer scale of similar orientation, forming a quadtree structure. Each coefficient at scale LL_3 is a root of such a tree, and the resulting data structure is an array of such trees. See Figure 4.7.

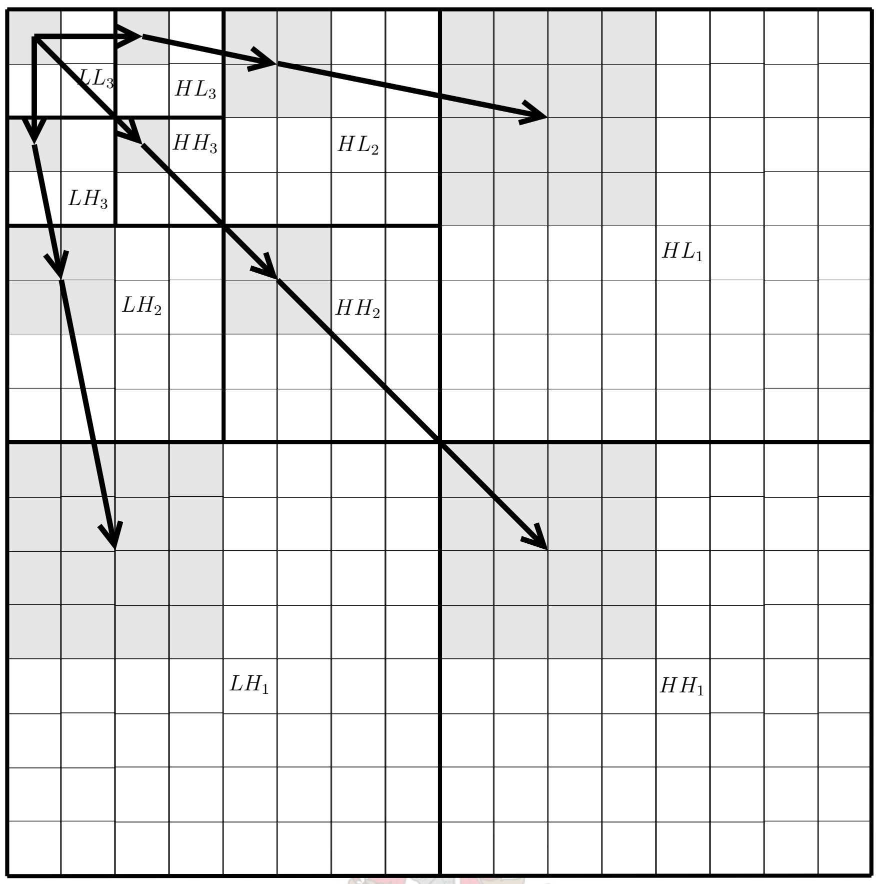


Figure 4.6: Relationship between subbands

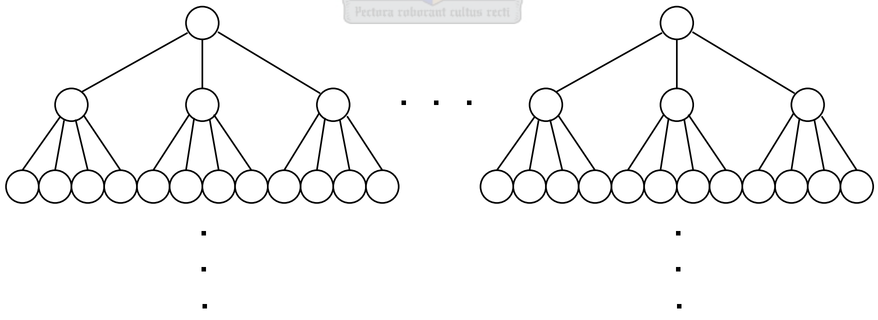


Figure 4.7: Array of quadtrees

This tree structure defines an ordering in which coefficients will be scanned during each iteration of the algorithm: No child node is scanned before its parent and each coefficient within a given subband is scanned before any coefficient in the next subband. This scanning order is illustrated in Figure 4.8. How the scanning is done within a specific subband is determined by choosing either a *raster scan* (Figure 4.9) or a *Morton scan* (Figure 4.10).

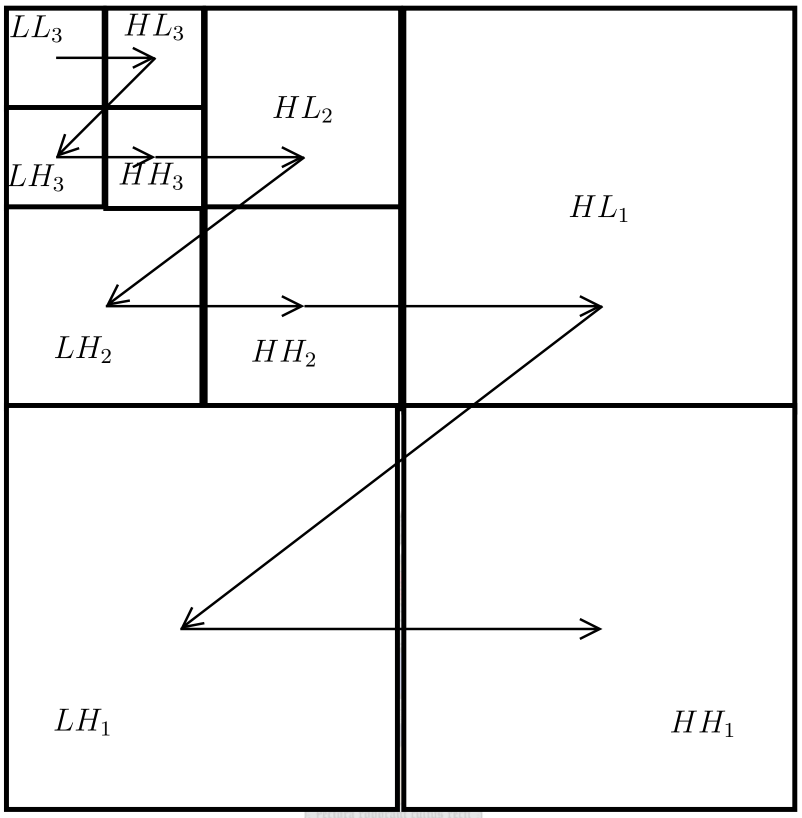


Figure 4.8: Scanning order within subbands

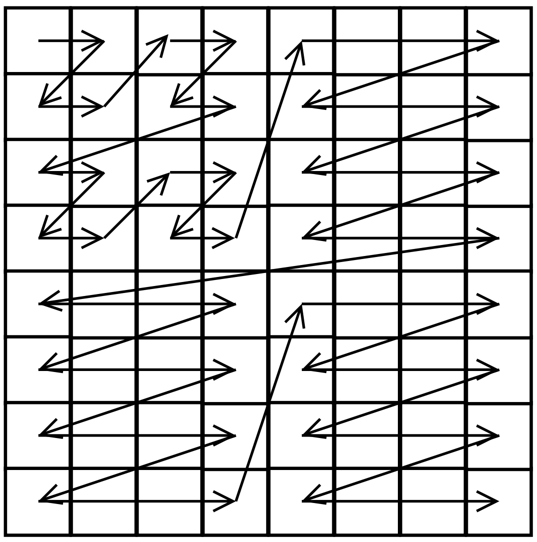


Figure 4.9: Raster scan

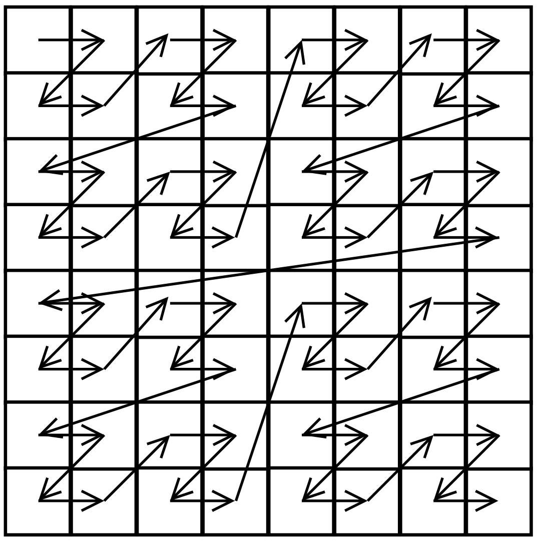


Figure 4.10: Morton scan

This tree structure enables the exploitation of the abovementioned tendency as follows: A wavelet coefficient C is defined to be *insignificant* with respect to a threshold t, if $|C| < t$. If a coefficient is deemed insignificant with respect to a certain threshold, then it is very likely that all its descendants (i.e. all coefficients of the same orientation in the same spatial location at finer scales) will also be insignificant with respect to that threshold. If this is the case, we define such a subtree to be a *zerotree* and that particular coefficient is called the *zerotree root*. Thus, if a coefficient is coded as a zerotree root, we know that it and all its descendants are insignificant with respect to the threshold; all its descendants are implicitly coded as insignificant. The symbol **T** is used to code a zerotree root. An insignificant coefficient which is not a zerotree root (has at least one significant descendant) is called an *isolated zero* and is coded with the symbol **Z**. A significant coefficient (one which is not insignificant with respect to the current threshold), on the other hand, is coded with a **P** if it is positive, and an **N** if it is negative. The process outlined above, known as encoding a *significance map*, is used in conjunction with successive approximation quantization during each iteration of the EZW algorithm — the threshold's magnitude is reduced from one iteration to the next, and a next significant bit is outputted for each significant coefficient, adding more and more precision to the coefficients and thus more and more detail to the encoded image. Typically the threshold is reduced by halving it during each iteration. Its initial value is then the largest power of two less than the maximum absolute value of the wavelet coefficients. The iterations of the algorithm are illustrated in Figure 4.11. As stopping condition we can use a target bitrate

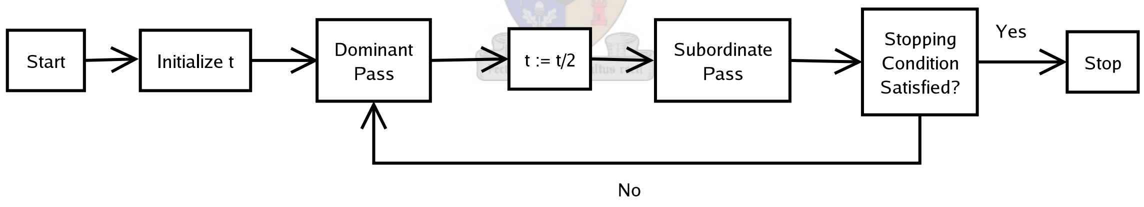


Figure 4.11: EZW algorithm

or some minimum threshold to encode the coefficients at full precision leading to perfect image reconstruction. The *dominant pass* is responsible for encoding the significance map, with the coefficients processed in the selected scanning order and treated as shown in Figure 4.12. Coefficients that have been identified as significant in previous dominant passes, should not be coded again; therefore, once identified as significant, we label these

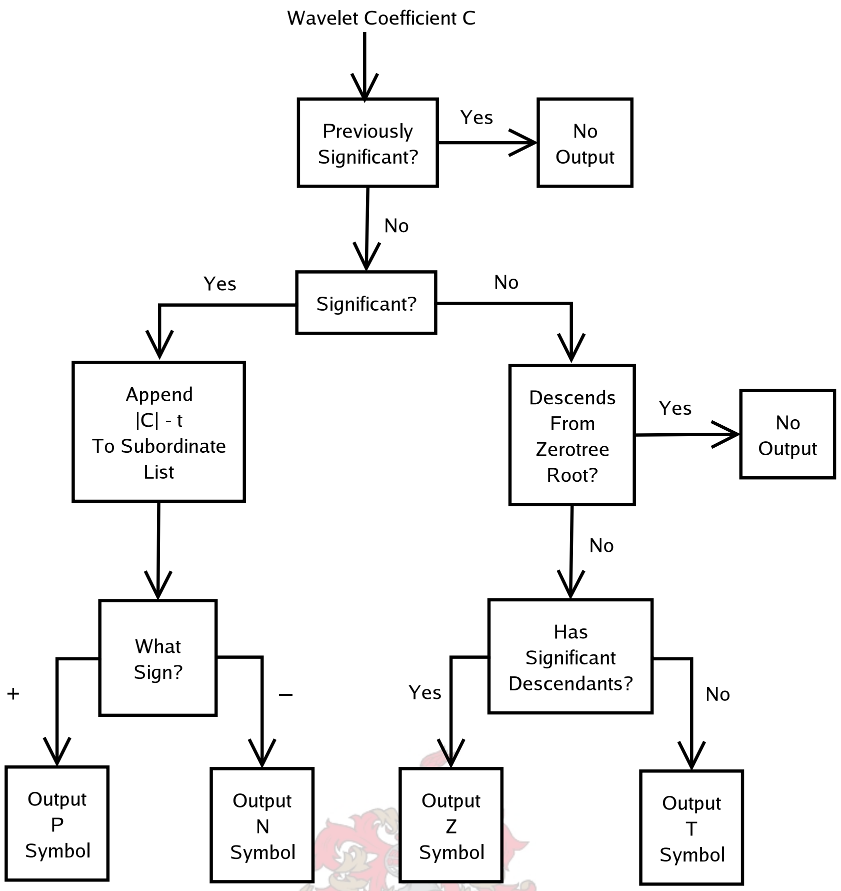


Figure 4.12: Dominant pass

coefficients as such, and no output is generated for them in later dominant passes. No output is generated for insignificant coefficients that descend from zerotree roots, since they are predictably insignificant. The *subordinate pass* is responsible for outputting the remaining significant bits for coefficients c that are deemed significant. Therefore the value $|C| - t$, holding the remaining bits (bits that have not yet been output), is placed in the *subordinate list* by the dominant pass for each new significant coefficient. The subordinate pass processes the items on the subordinate list as shown in Figure 4.13.

We see therefore that the EZW algorithm produces symbols over the 6-element alphabet $\{\mathbf{0}, \mathbf{1}, \mathbf{P}, \mathbf{N}, \mathbf{T}, \mathbf{Z}\}$. These symbols are then compressed using arithmetic coding and we can terminate the algorithm as soon as a target bitrate is reached.

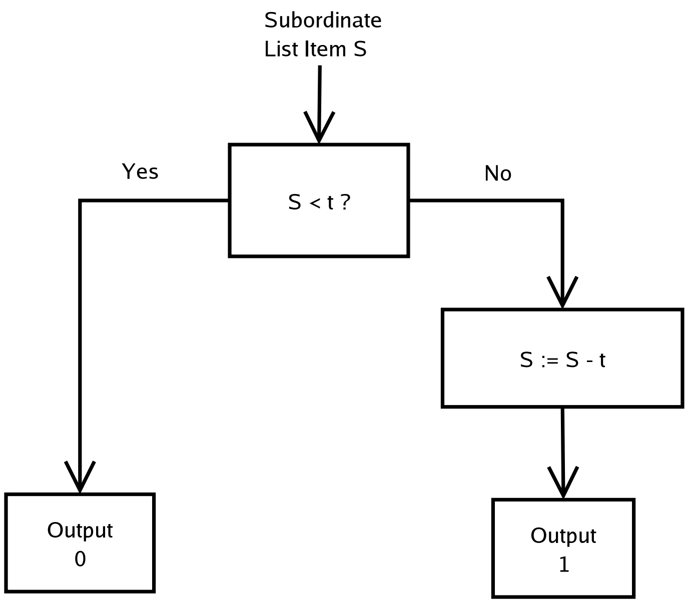


Figure 4.13: Subordinate pass

On a final note, this form of progressive (embedded) coding is similar in nature to the binary representation of a number like π: The more bits we add, the more accurately we represent the number. Similarly, the more scans (passes) we add to the encoding of the image, the more accurately we represent the original image. And in both cases we can stop as soon as our "bit budget" is exhausted. A good informal explanation of the EZW algorithm can also be found in [9].

Chapter 5

Fractal Image Compression

Fractal image compression is motivated by the observation that fractals, which are usually visually complex and intricate and contain detail at all resolutions, also have simple algorithmic descriptions. Of particular importance are those generated by collections of *affine transformations* $w_i, i = 1, \ldots, n$, with

$$w_i(\mathbf{x}) = A\mathbf{x} + \mathbf{b} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \mathbf{x} + \begin{bmatrix} e_i \\ f_i \end{bmatrix}, \ \mathbf{x} \in \mathbb{R}^2.$$

A classic example is of the Sierpinski triangle, specified by the three affine transformations for $\mathbf{x} \in [0,1] \times [0,1]$

$$\begin{aligned} w_1(\mathbf{x}) &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \mathbf{x}, \\ w_2(\mathbf{x}) &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix}, \\ w_3(\mathbf{x}) &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \mathbf{x} + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix}. \end{aligned}$$

The process can informally be interpreted as taking the unit square as input, applying the three transformations to the set and "pasting" the results together, giving a mapping from the unit square into itself, as illustrated in Figure 5.1. All three transformations reduce the unit square by $\frac{1}{2}$, as a result of the matrix multiplication, and w_2 and w_3 translate it with offsets $\begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix}$ and $\begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix}$, respectively. The resulting output is then used as input, i.e. the process is run in a feedback loop. The image that this converges to, the Sierpinski triangle, is shown in Figure 5.2.

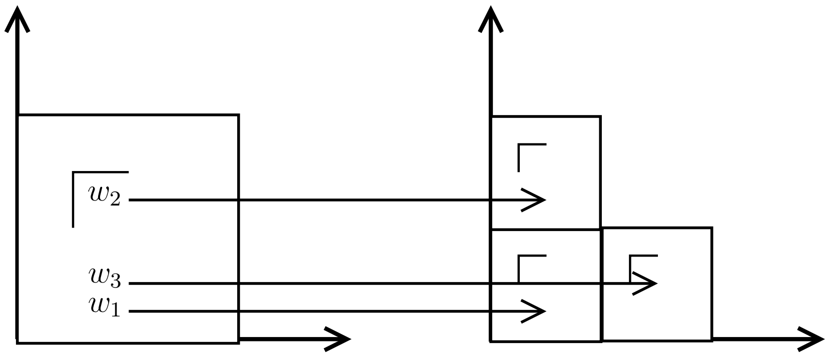


Figure 5.1: Affine transformations of the unit square

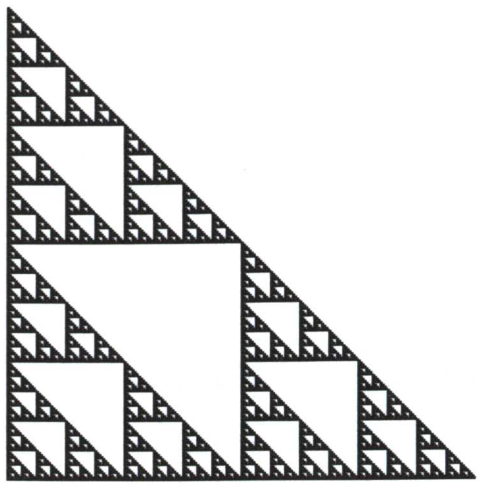


Figure 5.2: Sierpinski triangle

Notice that storing this fractal as a 256 × 256 digital image would require 524288 bits (65536 bytes), but its algorithmic description consists of only three affine transformations. If we represent each of the numbers specifying an affine transformation with 32 bits, storing them would only require 6×3 numbers $\times$ 32 bits/number = 576 bits (72 bytes). The idea of fractal image compression is that, given a "real life" image, hopefully a small number of affine transformations could be also found that would approximate it, enabling us to store it compactly similar to the Sierpinski triangle.

5.1 Mathematical Background

The affine transformations noted above are examples of *contractive mappings*: two points in the unit square, and $\mathbb{R}^2$ in general, are brought closer together by the transformation.

Definition: A *metric space* (X, d) is a set X on which a distance function $d : X \times X \to \mathbb{R}$, called a *metric*, is defined, such that, for any $a, b, c \in X$

$$\begin{aligned} &1) \quad d(a,b) \geq 0,\\ &2) \quad d(a,b) = 0 \Leftrightarrow a = b,\\ &3) \quad d(a,b) = d(b,a),\\ &4) \quad d(a,c) \leq d(a,b) + d(b,c). \end{aligned}$$

Thus, $\mathbb{R}^2$ with the Euclidean metric $d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} = ||\mathbf{x} - \mathbf{y}||_2$ is a metric space.

Definition: Let (X, d) be a metric space. A mapping $w : X \to X$ is *Lipschitz* with *Lipschitz factor* s, if there exists a positive real number s such that

$$d(w(x), w(y)) \leq sd(x, y), \ \ x, y \in X.$$

If $s < 1$, w is said to be *contractive* with *contractivity* s.

Thus an affine transformation

$$w(\mathbf{x}) = A\mathbf{x} + \mathbf{b}, \ \ \mathbf{x} \in (\mathbb{R}^2, \text{Euclidean})$$

is Lipschitz with Lipschitz factor $s = ||A||_2$, where $||A||_2$ denotes the induced 2-norm of matrix A. This is so because

$$d(A\mathbf{x} + \mathbf{b}, A\mathbf{y} + \mathbf{b}) = ||A\mathbf{x} + \mathbf{b} - A\mathbf{y} - \mathbf{b}||_2$$

$$= ||A\mathbf{x} - A\mathbf{y}||_2 = ||A(\mathbf{x} - \mathbf{y})||_2 \leq ||A||_2||\mathbf{x} - \mathbf{y}||_2 = sd(\mathbf{x}, \mathbf{y}).$$

We see furthermore that such an affine transformation is contractive if $||A||_2 < 1$. Therefore, the transformations w_1, w_2, w_3 of our introductory example, are contractive mappings in ($\mathbb{R}^2$, Euclidean). Similarly, the collection of the contractive mappings w_1, w_2, w_3 ($w_1, \ldots, w_n$ in general) applied to compact subsets of $\mathbb{R}^2$ (in this case, the unit square) is a contractive mapping, as we will see.

Definition: A subset of a finite dimensional metric space is *compact* if it is closed and bounded.

Definition: Let (X, d) be a metric space, and let $w_1, \ldots, w_n$ be contractive mappings. Then $\mathcal{H}(X) := \{S \subset X : S \text{ is compact}\}$, and the mapping $W : \mathcal{H}(X) \to \mathcal{H}(X)$ is defined by

$$W(S) = \bigcup_{i=1}^{n} w_i(S),\ S \in \mathcal{H}(X).$$

We will now introduce the *Hausdorff metric* as a way of measuring the distance between two compact subsets of a metric space. Given a metric space (X, d), the Hausdorff distance between two compact subsets A and B is calculated as follows: For each $x \in A$, find the $y \in B$ closest (in terms of metric d) to it. For each $x \in B$, find the $y \in A$ closest (in terms of metric d) to it. Choose the maximum of these minimal distances as the distance between the sets A and B. Examples of such subsets of $\mathbb{R}^2$ (with the Euclidean metric) are illustrated in Figure 5.3. The Hausdorff distance between them is indicated by a straight line. The Hausdorff metric is formally defined as follows.

Definition: Let (X, d) be a metric space and let A and B be compact subsets of X. With the notation

$$\delta_d(x, A) = \inf_{y \in A} d(x, y),\ x \in X,$$

the Hausdorff distance between A and B is defined as

$$h_d(A, B) = \max\left\{\sup_{x \in A} \delta_d(x, B),\ \sup_{y \in B} \delta_d(y, A)\right\}.$$

We now have the following result; see e.g [10].

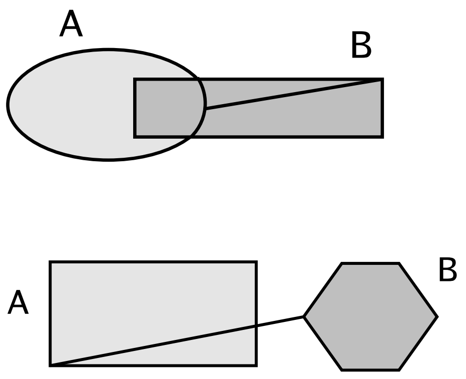


Figure 5.3: Examples of the Hausdorff metric

Theorem: If $w_i : \mathbb{R}^2 \to \mathbb{R}^2$ is contractive with contractivity s_i for $i = 1, \ldots, n$, then $W : \mathcal{H}(\mathbb{R}^2) \to \mathcal{H}(\mathbb{R}^2)$ is contractive in the Hausdorff metric $h_{\text{Euclidean}}$ with contractivity $s = \max\{s_i,\ i = 1, \ldots, n\}$.

Notice that the Sierpinski triangle is the limit set (or "limit image") of the contractive mapping $W = w_1 \cup w_2 \cup w_3$, in other words, iterating the mapping W, converges to the Sierpinski triangle. Also notice that applying the mapping to the limit image leaves it unchanged, i.e.

$$W(\text{Sierpinski}) = \text{Sierpinski}.$$

Furthermore, starting the iteration with different initial images seems to have (and indeed has) no effect on the limit image, as shown in Figure 5.4, where the mapping is applied to a square and a circle. We call such a limit the *fixed point* or the *attractor* of the mapping. The metric space $(\mathcal{H}(\mathbb{R}^2), h_{\text{Euclidean}})$ should be thought of as a space of (black-and-white) images. As we will see, contractive mappings in this space, such as the one yielding the Sierpinski triangle, then converge to a unique limit image in this space.

Before stating the main result, we need two more definitions and an auxiliary result.

Definition: A sequence of points $\{x_n\}$ in a metric space (X, d) is called a *Cauchy* sequence if, for any $\epsilon > 0$, there exists an integer N such that

$$d(x_m, x_n) < \epsilon,\ \forall\ m, n > N.$$

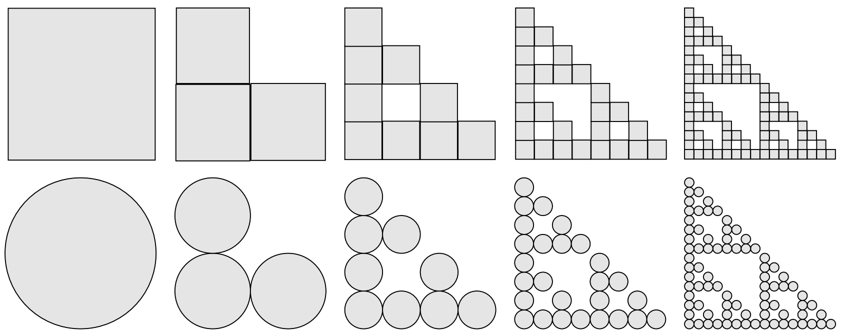


Figure 5.4: Using a square and a circle as initial images

Definition: A metric space (X, d) is *complete* if every Cauchy sequence in X converges to a limit point in X.

Theorem: Let (X, d) be a complete metric space. Then $\mathcal{H}(X)$ with the Hausdorff metric h_d is a complete metric space.

For a proof of this theorem, see e.g. [11].

According to a standard result in real analysis, the metric space $(\mathbb{R}^2, \text{Euclidean})$ is a complete metric space; therefore we have, from the previous theorem, that $(\mathcal{H}(\mathbb{R}^2), h_{\text{Euclidean}})$ is a complete metric space. We are now ready to state the main result, as proved in [10].

Theorem (The Contractive Mapping Fixed-Point Theorem): Let (X, d) be a complete metric space and $f : X \to X$ be a contractive mapping. Let $f^{\circ n}$ denote the nth iterate of the mapping f. Then there exists a unique point $x_f \in X$ such that for any point $x \in X$

$$x_f = f(x_f) = \lim_{n\to\infty} f^{\circ n}(x).$$

Such a point is called the *fixed point* or the *attractor* of the mapping f.

Thus, given any initial image in our space of images $(\mathcal{H}(\mathbb{R}^2), h_{\text{Euclidean}})$, a contractive mapping, such as the set of affine transformations specifying the Sierpinski triangle, is guaranteed to converge to a unique limit image.

Definition: Let (X, d) be a complete metric space. An *iterated function system* (IFS) is a collection of contractive maps $w_i : X \to X$, for $i = 1, \ldots, n$.

The problem of fractal image compression can thus be stated as follows: Given an image, we would like to find an IFS with a fixed point or attractor that is a good approximation to the image.

5.2 Generalization to Grayscale Images

So far, since the images we have considered have been subsets of $\mathbb{R}^2$, they have been restricted to being black-and-white: If an $\mathbf{x} \in \mathbb{R}^2$ is in the set, we colour it black, otherwise we leave it white. A grayscale image, on the other hand, can be interpreted as a graph of a function, say f, on some or other subset of $\mathbb{R}^2$ (viewed from above); in other words, it is a subset of $\mathbb{R}^3$ of the form $\{(x, y, f(x, y))\}$. Our affine transformations therefore have to be generalised to 3D affine transformations. However, since we want the output of such a transformation to once again be a graph of a function defined on a subset of $\mathbb{R}^2$, we restrict our 3D affine transformations to those of the form

$$w_i\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix}.$$

This can also be viewed as a combination of the affine transformations

$$u_i\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix}$$

and

$$v_i(z) = s_i z + o_i.$$

The first of these transformations, u_i, is an affine transformation of the subset of $\mathbb{R}^2$ on which f is defined, while v_i is an affine transformation of the function values themselves — it is a contrast and brightness adjustment of the grayscale values.

Notice, furthermore, that the attractors of such transformations as the one generating the Sierpinski triangle, are usually fractals — they are self-similar. Natural (real life) images do not contain such self-similarity and using IFSs to approximate them would therefore not work very well. Such images do, however, often contain a different, more limited, sort of self-similarity, as illustrated by the highlighted regions of Figure 5.5. Whereas the

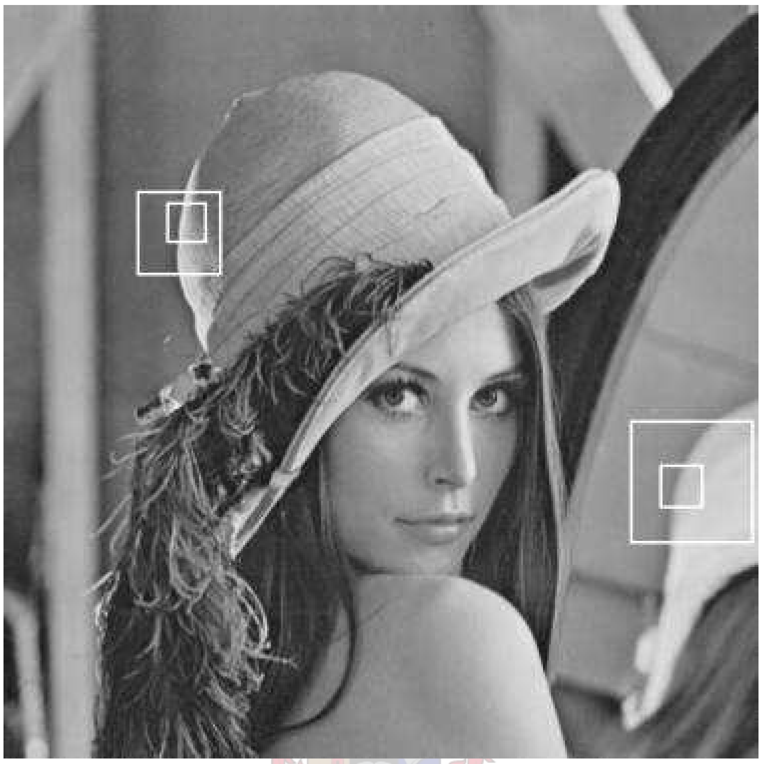


Figure 5.5: Self-similar portions of Lenna

Sierpinski triangle is formed by transformed copies of its <u>whole</u> self, we want to encode images so that their approximations will be formed by transformed <u>parts</u> of themselves. To do this, we generalize the concept of an IFS to that of a *partitioned iterated function system* (PIFS), thereby restricting the domains of the transformations u_i.

Definition: Let (X, d) be a complete metric space, and let $D_i \subset X$, for $i = 1, \ldots, n$. A *partitioned iterated function system* is a collection of contractive maps $w_i : D_i \to X$, for $i = 1, \ldots, n$.

Fisher says in [10] of a PIFS: "It partitions an image into pieces which are each transformed separately. By partitioning the image into pieces, we allow the encoding of many shapes that are impossible to encode using an IFS." During each iteration of the contractive mapping defined by a PIFS, each w_i will now map a portion, D_i, of the spatial region of the input image, to some portion, R_i, of the spatial region of the output image. At the same time, each w_i also applies a contrast and brightness transformation to the grayscale values of the transformed D_i, thus yielding the grayscale values of the corresponding R_i. The D_is are called the *domains* or *domain blocks*, and the R_is are called the *ranges* or *range blocks*. Notice that $u_i(D_i) = R_i$. Furthermore, we require that the R_is are disjoint (non-overlapping) and that they *tile* the spatial region of the input and output image of the contractive mapping. Thus, if the image to be compressed has spatial region $D \subset \mathbb{R}^2$, we have that $\bigcup R_i = D$.

5.3 The Encoding Procedure

The image to be encoded is firstly partitioned into a set of disjoint (non-overlapping) range blocks R_i. For each range block R_i we would like to find a domain block D_i so that the error between the pixel values of R_i and the pixel values of $u_i(D_i)$ is small. To this end, a *domain pool* is kept and for each R_i a search is performed through the domain pool to find a domain block minimizing the resulting error.

Consider for example the following encoding of the 256×256 Lenna image: As ranges we could choose the 1024 8×8 non-overlapping subsquares of the image, i.e. $R_1, R_2, \ldots, R_{1024}$. We want both transformations u_i and v_i that will make up w_i to be contractive; therefore domains are always chosen larger than ranges to ensure contractivity of the first. They are typically larger by the following possible factors: 2×2, 2×3, 3×2, and 3×3. Let us choose for this example all 16×16 subsquares of the image as domain pool; thus for this example domains are larger by a factor of 2×2. Since domains and ranges are square (rectangular in general), the matrix multiplication of u_i is restricted to the eight possible ways of mapping one square to another (four rotations or a reflection combined with one of four rotations). The contractivity of the mapping is implemented by subsampling (reducing) a domain block when mapping it to a range block. Thus, before a domain-range comparison is done, a candidate domain is subsampled to match the dimensions of the range in question.

A domain-range comparison is then done by calculating, for each of these eight reorientations, the optimal values of s_i and o_i in a least squares sense: If D_i (after reduction) and R_i contain n grayscale values, $d_1, d_2, \ldots, d_n$ and $r_1, r_2, \ldots, r_n$, respectively, we seek s_i and o_i to minimize the squared error E_i^2 given by the formula

$$E_i^2 = \sum_{k=1}^{n} (s_i d_k + o_i - r_k)^2.$$

The solutions are

$$s_i = \frac{n\sum_{k=1}^{n} d_k r_k - \sum_{k=1}^{n} d_k \sum_{k=1}^{n} r_k}{n\sum_{k=1}^{n} d_k^2 - \left(\sum_{k=1}^{n} d_k\right)^2},$$

and

$$o_i = \frac{1}{n}\left[\sum_{k=1}^{n} r_k - s_i \sum_{k=1}^{n} d_k\right],$$

and the resulting squared error is

$$E_i^2 = \frac{1}{n}\left[\sum_{k=1}^{n} r_k^2 + s_i\left(s_i \sum_{k=1}^{n} d_k^2 - 2\sum_{k=1}^{n} d_k r_k + 2o_i \sum_{k=1}^{n} d_k\right) + o_i\left(n o_i - 2\sum_{k=1}^{n} r_k\right)\right].$$

The minimized error E_i is then the square root of this value.

Once the D_i yielding the smallest such E_i has been found, the transformation w_i is completely specified, in this example, by the corresponding values of s_i and o_i, a 3-bit value indicating which reorientation was chosen, and two values specifying the position of the lower left corner of D_i relative to the lower left corner of the image — representing the values e_i and f_i of the transformation. Such a domain D_i is said to *cover* the range R_i. This information is then stored sequentially in the output file for each w_i (and thus sequentially for each corresponding R_i), thereby implicitly storing the positions of each R_i in this ordering. Since we also want the transformation v_i to be contractive, we require that $|s_i| < 1$. Any $|s_i|$ larger than some value s_{max} is truncated to s_{max}, where $s_{max} < 1$ is specified by the user. According to [10] 7 bits are generally optimal for storing an o_i and 5 bits are sufficient for storing an s_i. In the case of this example, the offset values e_i and f_i can be represented by 8-bit values each. This results in an output file of 3968 bytes, and thus a compression ratio of 16.5:1.

The image can be decoded by iterating the map W starting with any initial image. More or less ten iterations are usually sufficient for the convergence to stabilize. Figure 5.6, taken from [10], shows an initial image f_0, the first iteration $W(f_0)$, the second iteration $W^{\circ 2}(f_0)$, and the tenth iteration $W^{\circ 10}(f_0)$.

The above example is the simplest fractal encoding scheme and does not allow us to manipulate the quality versus compression trade-off. In more sophisticated schemes, this is done by choosing a covering tolerance that each approximation $R_i \approx u_i(D_i)$ should satisfy. If the optimal w_i for the R_i in question satisfies this criterion, it is accepted and written to the output file; if not, it is rejected and the range R_i is partitioned further (subdivided) and its subblocks are then used as potential range blocks instead. To this end one also selects a partitioning scheme, and it is therefore the partitioning scheme that determines how the range blocks are chosen.

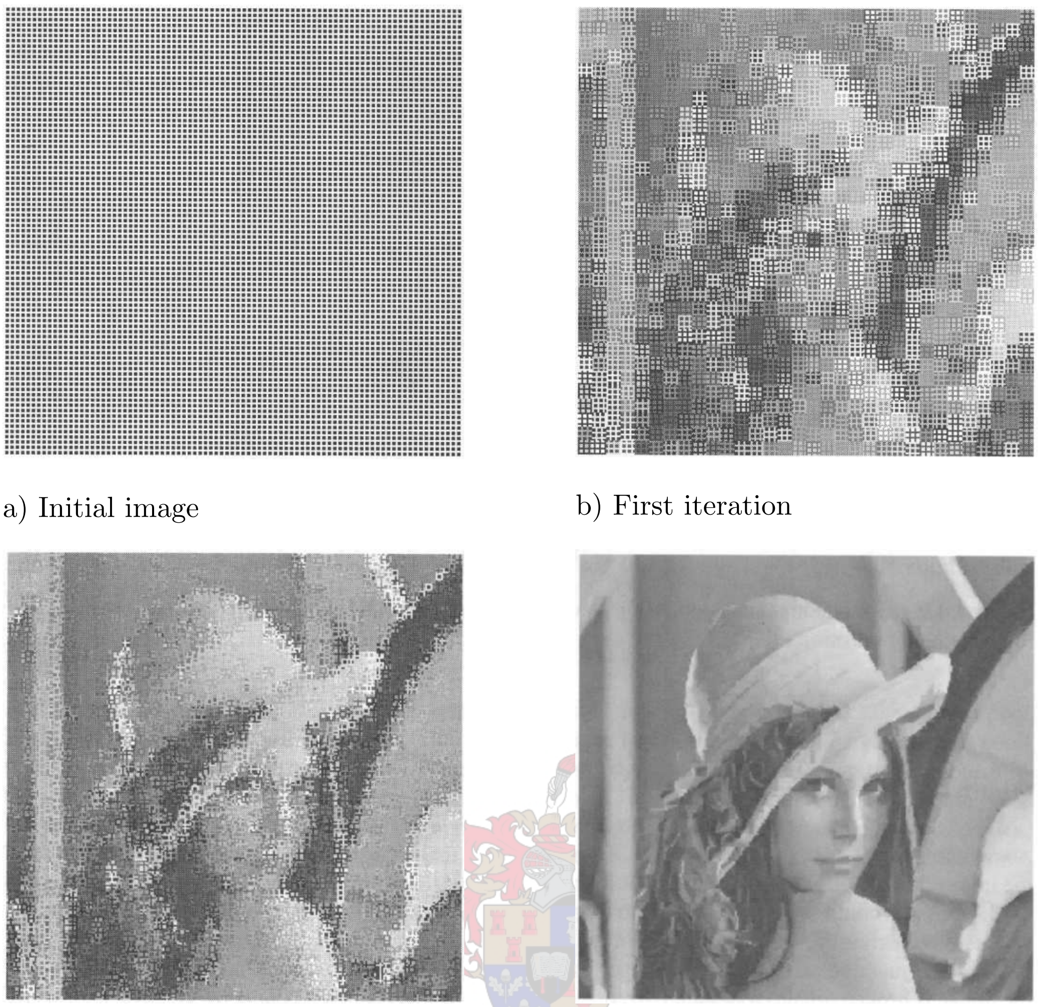
We can now summarize the encoding algorithm as shown in Algorithm 5.1.

5.4 Partitioning Schemes

The two most popular partitioning schemes are *quadtree* partitioning and *HV* partitioning.

5.4.1 Quadtree Partitioning

Fisher [10] explains quadtree partitioning as follows: “In a quadtree partition, a square in the image is broken up into four equal-sized sub-squares when it is not covered well enough by some domain. The process repeats recursively starting from the whole image



a) Initial image b) First iteration

c) Second iteration d) Tenth iteration

Figure 5.6: Several iterations of the decoding process [10]

and continuing until the squares are small enough to be covered within some specified rms tolerance." The quadtree scheme will not be discussed further here, since it is somewhat limited (limited to square images) and does not perform as well as HV partitioning.

5.4.2 HV Partitioning

Fractal encoding with HV partitioning gives the best results of all fractal image compression schemes. A rectangle in the image (to be encoded) is partitioned either horizontally

```
Data: $r_{max}$ = maximum allowable size of a range
Data: $r_{min}$ = minimum allowable size of a range
(* In the case of rectangles, size($R_i$) = length of the longest side *)
Data: $\epsilon$ = covering tolerance
Data: list of uncovered ranges (initially empty)
Insert whole image in list of uncovered ranges;
while there are uncovered ranges $R_i$ do
    Extract first $R_i$;
    if $size(R_i) > r_{max}$ then
        Partition $R_i$ into smaller ranges that are inserted into the list;
    else
        Find the $D_i$ and corresponding $w_i$ that best cover $R_i$;
        if $E_i < \epsilon$ or $size(R_i) \leq r_{min}$ then
            Write out $w_i$;
        else
            Partition $R_i$ into smaller ranges that are inserted into the list;
        end
    end
end
```

Algorithm 5.1: Fractal encoding algorithm

or vertically to form two new rectangles, and is done in such a way that partitions tend to be along strong vertical or horizontal edges in the image and that narrow rectangles are avoided. Once again partitioning repeats until a covering tolerance is satisfied. The partitioning is selected using the following technique: Given a range to be subdivided, a horizontal or vertical position needs to be selected at which to make the partition. If the range consists of the pixels $r_{i,j}, 0 \leq i \leq M, 0 \leq j \leq N$, this is done by calculating the *biased horizontal differences*

$$h_j = \frac{\min(j, N-j-1)}{N-1} \left(\sum_i r_{i,j} - \sum_i r_{i,j+1} \right),$$

and *biased vertical differences*

$$v_i = \frac{\min(i, M-i-1)}{M-1}\left(\sum_j r_{i,j} - \sum_j r_{i+1,j}\right).$$

The terms $\frac{1}{N-1}\left(\sum_i r_{i,j} - \sum_i r_{i,j+1}\right)$ and $\frac{1}{M-1}\left(\sum_j r_{i,j} - \sum_j r_{i+1,j}\right)$ represent successive differences between the horizontal and vertical averages, whereas the terms $\min(j, N-j-1)$ and $\min(i, M-i-1)$ represent their distances from the nearest side of the range. The position j or i for the maximum $|h_j|$ or $|v_i|$, depending on which is larger, is chosen as the position along which to make the partition.

To quote Fisher [10], "The range position is stored implicitly by specifying the partition type, horizontal or vertical, and its position, an offset from the upper or left side. This offset requires fewer bits to store as the partition gets refined. The range partitions are stored recursively from the top left to the bottom right, so the position information can be derived from the partition direction and offset ... The domain position is specified by specifying an index into a list of all possible domains".

5.5 Choosing the Domain Pool

Since the ranges can be rectangles of a wide variety of dimensions, the domain pool can be, for instance, chosen to be all sub-rectangles of the image. This would, however, result in a very large domain pool, leading to very high encoding times, so it is not done. The domain pool is instead chosen as some subset of this set. Three major types of domain pools are conventionally used. One of these is selected by the user and its number is specified at the beginning of the output file to notify the decoder.

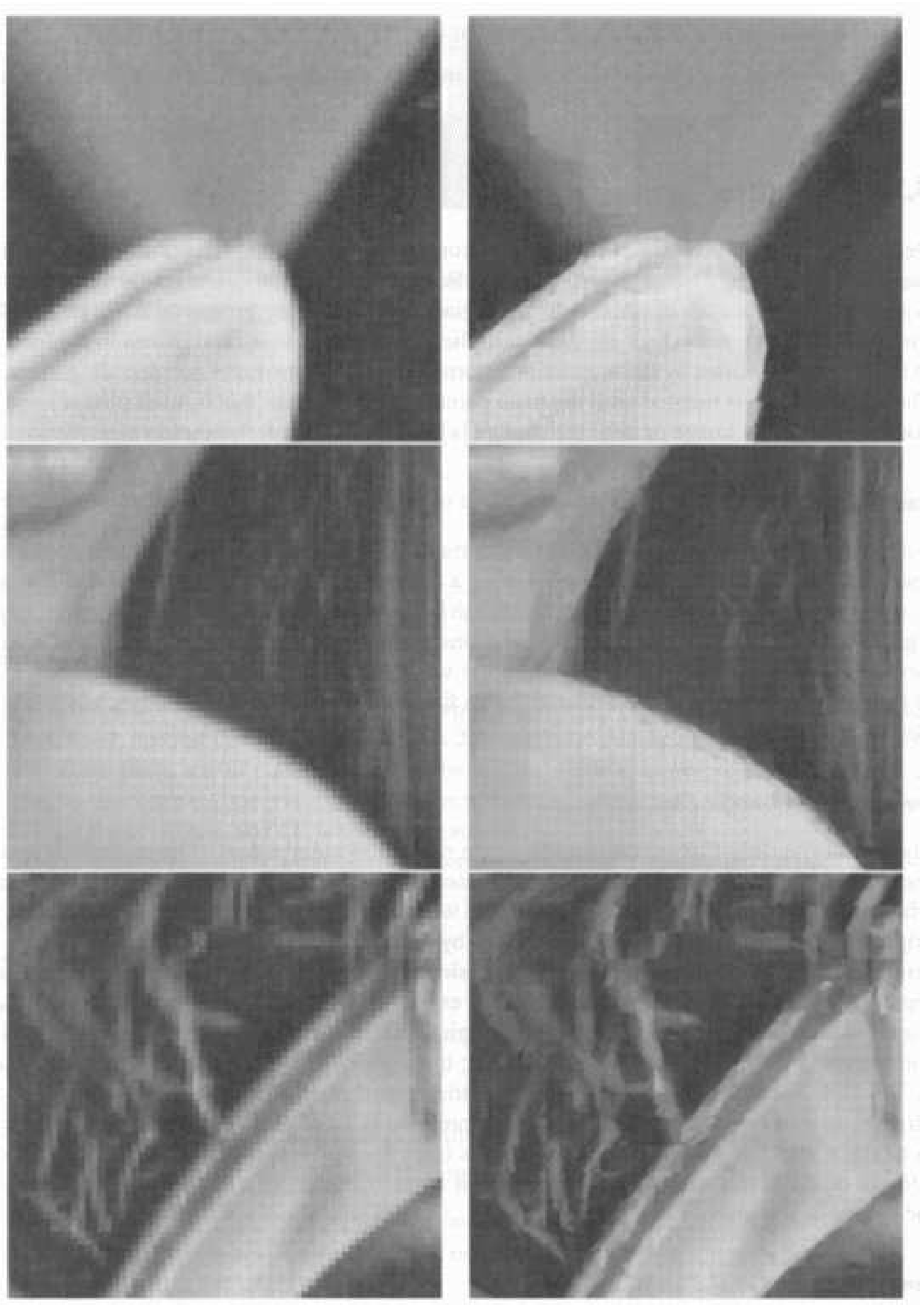
In all three cases the domain pools are determined by a *lattice spacing* determined by a parameter l: the upper left corners of the domains are positioned on the lattice. (The parameter l is chosen by the user and written to the output file.) The first type, $\mathbf{D_1}$, has every spacing equal to l, resulting in a more or less equal number of domains of each size. $\mathbf{D_2}$ has more large domains and fewer small domains — the lattice spacing is the domain size divided by l. For $\mathbf{D_3}$ the lattice spacing is calculated the same way as for $\mathbf{D_2}$, but the assignment of a spacing to a domain is done in the reverse order: The largest domains have a lattice corresponding to the smallest domain size divided by l, and vice versa. Therefore $\mathbf{D_3}$ has more small domains and fewer large domains.

Domain-range comparisons can be further sped up by classification techniques: only domains in the same class as the range in question are compared with it. Various classification schemes are discussed in [10].

5.6 Resolution Independence

An interesting attribute of fractally encoded images is that they are resolution independent — they can be decoded at any resolution. This is a result of the encoding affine transformations generating detail at all scales. Thus, just like the Sierpinski triangle, which was the attractor of a set of affine transformations, had detail at all scales, the attractor constituting a decoded fractally encoded image will have "detail" at all scales.

For instance, the encoding of an image of resolution 256×256 can be decoded at resolution 512×512. Examples of such "artificial" detail are shown in Figure 5.7, taken from [10], where portions of an encoding of Lenna are shown decoded at four times the original size. Magnifications of the original portions are shown as well, showing pixellation.



Original portions Decoded counterparts

Figure 5.7: Artificially created detail [10]

Chapter 6

Image Compression using Weighted Finite Automata

Similarly to fractal image compression, encoding images with weighted finite automata (WFA) is a “fractal-type” technique: Subimages are described in terms of other subimages (and sometimes themselves), and fractals are thus very naturally described by WFA. Here subimages are related through linear combinations instead of through affine transformations. Although this technique is limited to $2^n \times 2^n$ images, when used in conjunction with a wavelet transform, it is alleged in [12] and [13] to give better results for such images than any other image compression technique. On its own, it is alleged to give results comparable to those obtained with fractal image compression with HV partitioning. (As we will see in the next chapter, the latter does not appear to be true.) This is also the technique that has been implemented for this thesis, and a detailed description of the implementation will be given in Section 6.6.

6.1 Finite State Machines

A *finite state machine* (FSM) or *finite automaton* is a model used in theoretical computer science to describe the execution of an algorithm. Its behaviour is specified by a finite set of input symbols, Σ, called an *alphabet*, a finite set of states, Q, and a state function, $f_{st} : \Sigma \times Q \to Q$ (and sometimes also a finite set of output symbols, O, and a machine function, $f_{ma} : \Sigma \times Q \to O$). At any given time the FSM is in exactly one state; when the machine receives an input, it will alter its state according to the state function (and in the case of an output set, generate an output symbol according to the machine function). One of the states is called the *initial state* — given it and a sequence of input symbols, the behaviour of the FSM can be determined. Some of the states can also be defined as *final states* — once in one of these states, the machine can never reach other states. Final states model the successful termination of an algorithm (or sometimes the *acceptance* of an input sequence).

(Here we will restrict our discussion to FSMs that do not generate an output and hence have no machine function.)

An FSM is represented graphically by a *transition graph*. Each state is represented by a vertex in the graph, and a directed edge from say state s_i to state s_j is used to indicate that an input symbol, say $a \in \Sigma$, causes the machine to change from state s_i to state s_j. This is referred to as a *transition*, and it is labelled by the symbol a that causes it. (If input symbol $a \in \Sigma$ causes the output $o \in O$, the transition is labelled by $a \to o$.) The vertex representing the initial state is often indicated by a square, while the vertex representing a final state is indicated by two concentric circles.

Consider, for example, an FSM with $\Sigma = \{0, 1\}$; $Q = \{s_0, s_1, s_2, s_3\}$, where s_0 is the initial state and s_3 is the final state; and a state function defined by the *state table* shown in Table 6.1. (If the FSM had had a machine function, it too would be defined by a similar state table.) This FSM is then represented by the transition graph shown in Figure 6.1.

The information contained in a transition graph can also be specified with a *transition*

	s_0	s_1	s_2	s_3
0	s_1	s_3	s_2	s_3
1	s_2	s_1	s_3	s_3

Table 6.1: Example state table

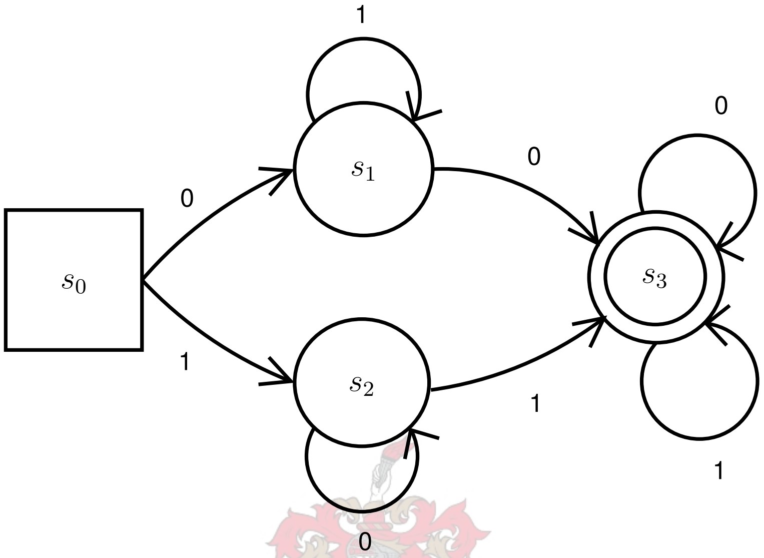


Figure 6.1: Example transition graph

matrix. Given an FSM consisting of n states, a transition matrix will consist of $n \times n$ entries, with each entry (i, j) recording the input symbol(s) causing a transition from state s_i to state s_j. (In the case of an FSM that generates output symbols, each generated output symbol would also be recorded in the entry (i, j) next to the input symbol that caused it.) The transition matrix for our example FSM above, and thus corresponding to the graph in Figure 6.1, is illustrated in Table 6.2. A '-' in an entry (i, j) indicates that no input symbol causes a transition from state s_i to state s_j.

The number of symbols in Σ is denoted by $|\Sigma|$. A *word* w over Σ is any finite sequence of symbols from Σ. The empty word, ϵ, is the word consisting of no symbols. The length of a word w is denoted by $|w|$. Σ^k is the set of all words of length k over Σ, while Σ^* is the

	s_0	s_1	s_2	s_3
s_0	-	0	1	-
s_1	-	1	-	0
s_2	-	-	0	1
s_3	-	-	-	0;1

Table 6.2: Example transition matrix

set of all words over Σ, including the empty word ϵ.

6.2 Weighted Finite Automata

We now generalize the concept of a finite automaton to that of a *weighted finite automaton* (WFA). Each transition is labelled by both an input symbol and a real number, known as the *weight* of the transition. Instead of initial and final states, we have an initial and final *distribution* value associated with each state. An example of a transition graph for a WFA with $\Sigma = \{0, 1, 2, 3\}$ and two states is shown in Figure 6.2. Initial and final distribution values are written inside each state.

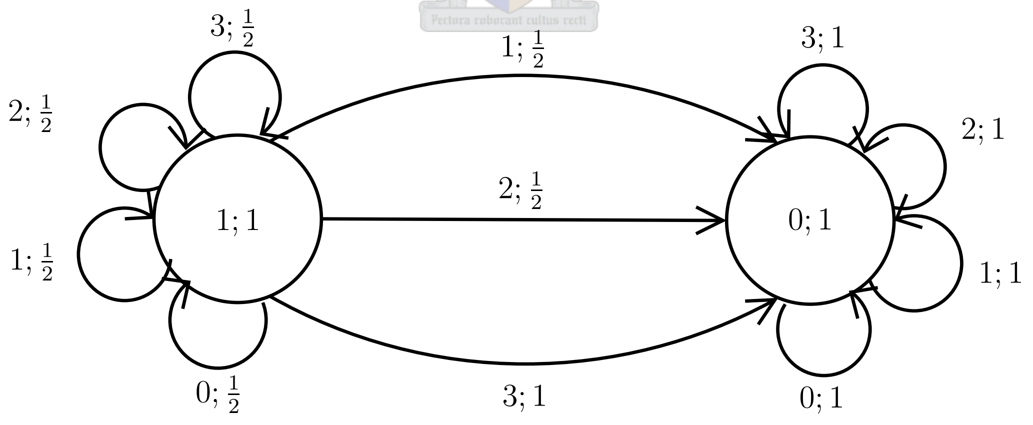


Figure 6.2: Example WFA

Given a WFA over an alphabet Σ consisting of n states, it is represented by $|\Sigma|$ transition matrices W_a, $a \in \Sigma$, each of size $n \times n$, a $1 \times n$ row vector I, and an $n \times 1$ column vector F. The entry $W_a(i,j)$ is the weight of the transition from state i to state j for input symbol a, if it is non-zero. If the value is zero, there is no transition from state i to state j for input symbol a. The ith elements of vector I and F denote the initial and final distribution values, respectively, of state i.

A WFA can be used to specify a grayscale image of resolution $2^n \times 2^n$. Let $\Sigma = \{0,1,2,3\}$ and let each $a \in \Sigma$ denote a quadrant of a square (our image), as shown in Table 6.3. This

1	3
0	2

Table 6.3: Quadrant addressing

is applied recursively until the square has been divided into $2^n \times 2^n$ sub-squares; a word w of length n over Σ then represents the address of one such sub-square. An example of the sub-squares addressed by words of length 2 is shown in Table 6.4. In general, if a

11	13	31	33
10	12	30	32
01	03	21	23
00	02	20	22

Table 6.4: Addresses specified by words of length 2

sub-square is addressed by word w, its quadrants will be addressed by $w0$, $w1$, $w2$ and $w3$. The empty word ϵ is used to address the entire image. This also leads to a quadtree representation of such an image: The root of the tree has address ϵ, the four quadrants of the image correspond to the four children of the root, etc. Thus, each word of length k is the address of a unique node in the tree at depth k.

We now define a function $f : \Sigma^* \to \mathbb{R}$ by

$$f(w) = f(a_1 a_2 \ldots a_k) = IW_{a_1} W_{a_2} \ldots W_{a_k} F,\ w \in \Sigma^*,\ |w| = k.$$

In other words, f specifies the grayness value of any square lying in resolution $2^k \times 2^k$, therefore defining a multiresolution image. For the different resolutions to be compatible, we require f to be *average preserving*:

$$f(w) = \frac{1}{4}[f(w0) + f(w1) + f(w2) + f(w3)].$$

The example 4×4 image shown in Table 6.5 will therefore have the corresponding quadtree representation shown in Figure 6.3.

8	8	9	10
8	8	9	9
9	7	7	8
10	8	9	7

Table 6.5: Example 4×4 image

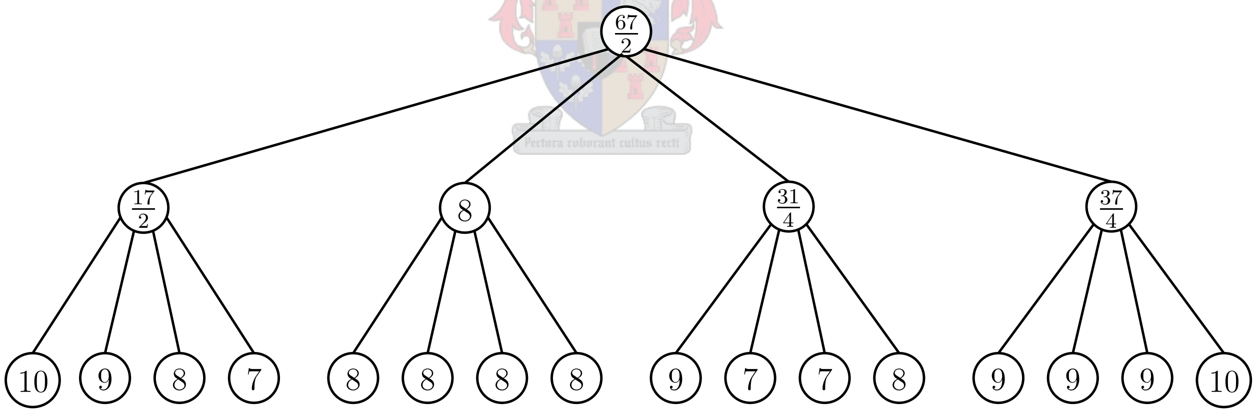


Figure 6.3: Quadtree representation of example 4×4 image

The grayness value of a square with address w is thus the average of the grayness values of its four quadrants (sub-squares) $w0$, $w1$, $w2$ and $w3$. Specifically, the value of $f(\epsilon)$ is the average grayness of the entire image.

The way the WFA defines an image should also be interpreted as follows: Each state i in the WFA corresponds to some sub-image of the entire image, with $F(i)$ equal to the average grayness of that sub-image. One state in particular corresponds to the entire image, and the vector I indicates which one that is: If state k corresponds to the entire image, then

$$I(k) = 1, \quad \text{and } I(l) = 0, \quad l \neq k.$$

Furthermore, the weighted transitions indicate how quadrants of sub-images are expressed as linear combinations of other sub-images (possibly including the sub-image that the quadrant belongs to), with the weights specifying the coefficients of the linear combinations, and the input symbol specifying the quadrant in question, i.e.

$$(\phi_i)_a = W_a(i,1)\phi_1 + W_a(i,2)\phi_2 + \ldots + W_a(i,n)\phi_n$$

where ϕ_i denotes the image associated with state i and $(\phi_i)_a$ denotes quadrant a of ϕ_i, for $i = 1, \ldots, n$. Thus, the quadrant a of the image ϕ_i is the linear combination of images $\phi_1, \phi_2, \ldots, \phi_n$ with coefficients given by the ith row of matrix W_a.

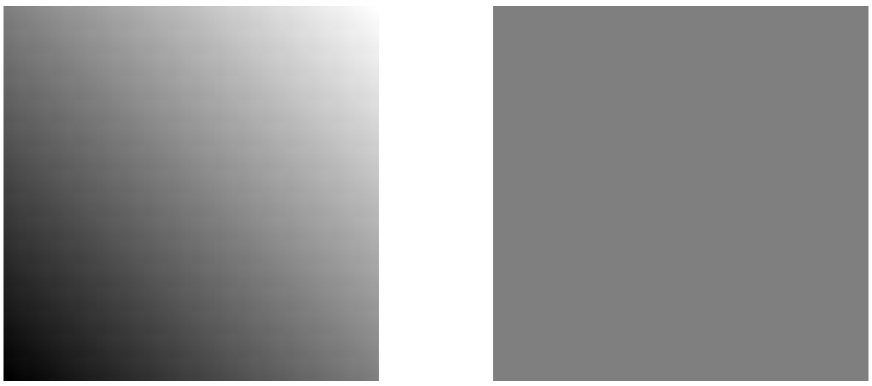
Consider as an example the WFA shown in Figure 6.2 above. This WFA is specified by

$$I = [1 \quad 0], \quad F = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

$$W_0 = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix}, \quad W_1 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix},$$

$$W_2 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix}, \quad W_3 = \begin{bmatrix} \frac{1}{2} & 1 \\ 0 & 1 \end{bmatrix}.$$

This WFA is actually an encoding of the "linear grayness" function $f(x, y) = x+y$ defined on the unit square and shown (at resolution 128×128) in Figure 6.4(a). (The function values range between 0 and 2; therefore they have been rescaled to pixel values in Figure 6.4(a) by multiplying them by 128.) Since the initial distribution is $I = [1 \quad 0]$, ϕ_1 (the image of state 1) is also f (the entire image). The image of state 2, ϕ_2, is shown in Figure



a) Linear grayness function and ϕ_1 b) ϕ_2

Figure 6.4: Images of the states of the example WFA

6.4(b). (All of its function values are 1; therefore they have also been rescaled in Figure 6.4(b) by multiplication by 128. Had they been multiplied by 255, the image would have been uniformly white.) The vector F states that the average grayness (average of the pixel values) of ϕ_1 (and thus the entire image) is 1 and that of ϕ_2 is also 1. The information contained in the first rows of matrices W_0, W_1, W_2 and W_3 is illustrated in Figure 6.5: The first row of W_0 states that quadrant 0 of the whole image is equal to the whole image with grayness uniformly scaled by $\frac{1}{2}$, i.e.

$$(\phi_1)_0 = \frac{1}{2}\phi_1.$$

According to the first rows of W_1 and W_2 both quadrants 1 and 2 of the whole image can be expressed by the linear combination

$$(\phi_1)_1 = (\phi_1)_2 = \frac{1}{2}\phi_1 + \frac{1}{2}\phi_2.$$

Finally, quadrant 3 of $\phi_1 = f$ can be written as

$$(\phi_1)_3 = \frac{1}{2}\phi_1 + \phi_2.$$

The second rows of $W_0, \ldots, W_3$ (and the outgoing transitions from state 2 to itself) indicate that all four quadrants of ϕ_2 are identical to ϕ_2. Notice that ϕ_2 is perfectly self-similar (completely described in terms of itself) — therefore WFA are in general a way of describing fractals.

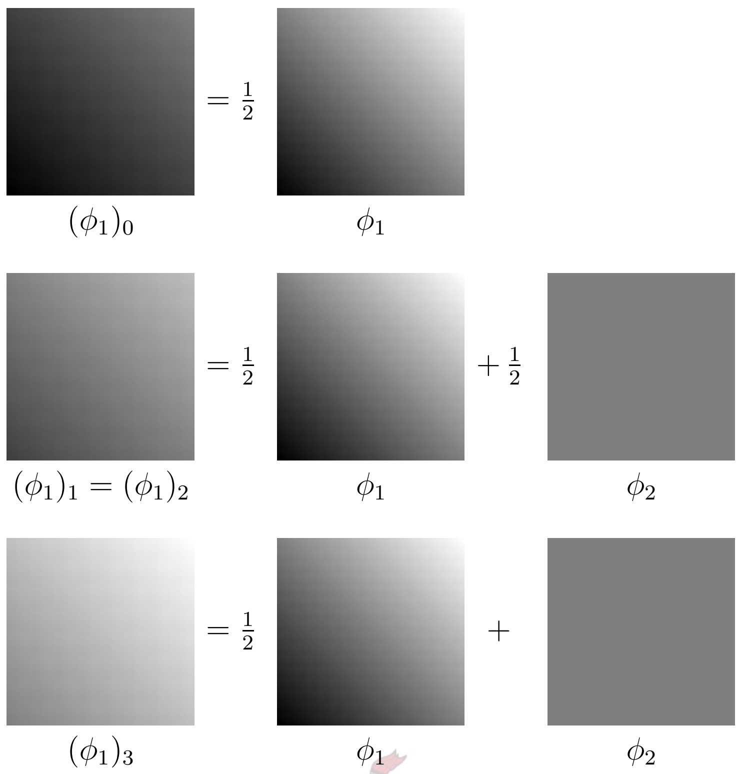

$(\phi_1)_0 = \frac{1}{2}\phi_1$
$(\phi_1)_1 = (\phi_1)_2 = \frac{1}{2}\phi_1 + \frac{1}{2}\phi_2$
$(\phi_1)_3 = \frac{1}{2}\phi_1 + \phi_2$

Figure 6.5: Quadrants of ϕ_1 expressed as linear combinations of ϕ_1 and ϕ_2

Furthermore, the images described (encoded) by WFA can be decoded at any resolution $2^n \times 2^n$. Let us decode the image defined by the WFA in the example above at resolution 4×4; the addresses of the pixels are then shown by Table 6.4. The grayness values are computed as

$$f(00) = IW_0W_0F = [1 \quad 0] \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{4},$$

$$f(01) = IW_0W_1F = [1 \quad 0] \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{2},$$

and similarly for $f(02), \ldots, f(33)$. The resulting decoded image is illustrated by Table 6.6. Therefore, just like fractally encoded images, WFA encoded images are resolution independent. We see that fractal image compression and WFA image compression share similarities — we can collectively refer to them as “fractal type” techniques. Indeed,

1	$\frac{5}{4}$	$\frac{3}{2}$	$\frac{7}{4}$
$\frac{3}{4}$	1	$\frac{5}{4}$	$\frac{3}{2}$
$\frac{1}{2}$	$\frac{3}{4}$	1	$\frac{5}{4}$
$\frac{1}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	1

Table 6.6: Decoded linear grayness function at resolution 4×4

image compression with WFA is treated in its own chapter in [10].

6.3 Upsampling and Downsampling

Notice from the quadtree in Figure 6.3 that a $2^n \times 2^n$ image is mapped to an array of 4^n elements by associating it with the leaves of its quadtree representation. Furthermore, given such an array of leaves, other vectors of nodes of the quadtree can be calculated by *downsampling* the vector the required number of times. This downsampling is done by averaging groups of four consecutive elements to form new elements. For instance, given the array of leaves of the quadtree in Figure 6.3, shown in Table 6.7, downsampling it once yields the array of nodes at depth 1, shown in Table 6.8, which are also the values of the example image at resolution 2×2, rearranged as an array.

10	9	8	7	8	8	8	8	9	7	7	8	9	9	9	10

Table 6.7: Array of leaves of the quadtree in Figure 6.3

$\frac{17}{2}$	8	$\frac{31}{4}$	$\frac{37}{4}$

Table 6.8: Array resulting from downsampling the array in Table 6.7

On the other hand, values of an image of resolution $2^n \times 2^n$ at higher resolutions (equivalently, nodes in its quadtree at depth further than n) can be artificially constructed by *upsampling.* The upsampling is done in such a way that the average preserving property of the image is preserved: Given a node at depth n, its children (at depth $n + 1$) inherit its grayness value. For instance, the image of resolution 2×2 shown in Table 6.9 and represented by the quadtree in Figure 6.6, will have the quadtree representation shown in Figure 6.7 after upsampling once.

b	d
a	c

Table 6.9: Example 2×2 image

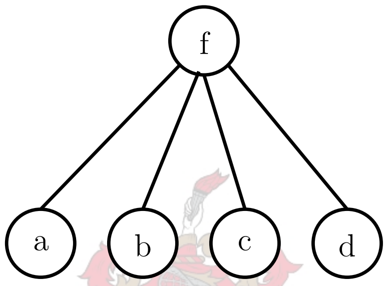


Figure 6.6: Quadtree representation of example 2×2 image; $\mathrm{f} = \frac{1}{4}(\mathrm{a} + \mathrm{b} + \mathrm{c} + \mathrm{d})$

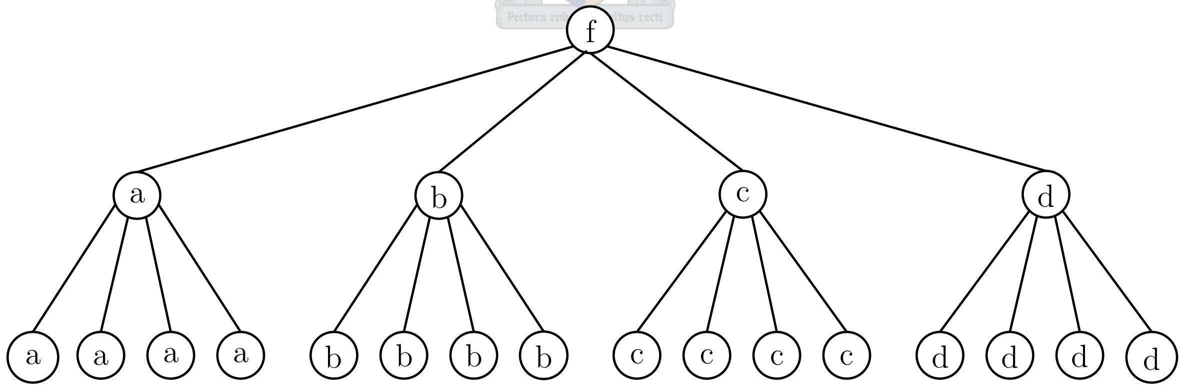


Figure 6.7: Resulting quadtree after upsampling once; $\mathrm{f} = \frac{1}{4}(\mathrm{a} + \mathrm{b} + \mathrm{c} + \mathrm{d})$

Equivalently, its array of leaves, shown in Table 6.10, yields the array shown in Table 6.11

after upsampling once. Thus, in terms of an array, upsampling is done by replicating each entry four times.

a	b	c	d

Table 6.10: Array of leaves of the quadtree in Figure 6.6

a	a	a	a	b	b	b	b	c	c	c	c	d	d	d	d

Table 6.11: Array resulting from upsampling the array in Table 6.10

Associating squares with arrays in this way, along with upsampling and downsampling, will be fundamental when attempting to express a square as a linear combination of other squares. These concepts will therefore play an important role in our encoding and decoding algorithms (as will be seen in subsequent sections).

6.4 Encoding (Inference) Algorithms

Given a $2^n \times 2^n$ grayscale image, we would like to find, or *infer*, a WFA approximating it. The inference algorithms published by Culik and Kari (both iterative [14] and recursive [12]) do this by setting the first state ϕ_1 equal to the whole image. All four quadrants of a state are processed by

1. trying to approximate the quadrant with a linear combination of existing states (images) and/or

2. choosing the quadrant as a new state which is also processed.

In the case of the iterative (older) algorithm [14], if the quadrant is adequately approximated (in terms of some error tolerance), the linear combination is accepted; if not, the quadrant is chosen as a new state and is placed in a list of states that still have to be processed. All four quadrants of a state are processed before moving on to a next state. Once all newly created states have been processed, the algorithm terminates.

The recursive algorithm [12], on the other hand, is the more recent of the two, and is the one implemented for this thesis. It also takes into consideration the increase in the size of the compressed output file resulting from steps 1 and 2. The algorithm is recursive because for each newly created state, a new instance of the algorithm is invoked to process it. Each invocation attempts to minimize the quantity

$$cost = error + G.s.$$

Whichever alternative (1 or 2) gives the smaller value for *cost*, is chosen. The variable *error* denotes the error in the approximation to the quadrant in question, and s denotes the number of bits required to store the new information. The number $G \in \mathbb{R}$ is a parameter controlling the quality versus compression trade-off. The larger the value of G, the "smaller" the produced WFA will be and the poorer the approximation to the original image. The smaller the value of G, the better the approximation will be and the "larger" the WFA will be. In particular, choosing $G = 0$ will result in perfect image reconstruction.

The recursive inference algorithm is given in pseudocode form in Algorithm 6.1 (make_wfa). At any given time during the execution of the algorithm, global variable n keeps track of the number of states in the WFA thus far. The parameters i, k and max passed to a recursive call indicate that the new invocation should attempt to approximate the image ϕ_i at level k (resolution $2^k \times 2^k$) with a resulting *cost* no larger than max. If so, the value of the *cost* is returned, otherwise ∞ is returned.

For the calculation of the value *error* we introduce the notation

$$d_k(f,g) = \sum_{w \in \Sigma^k} [f(w) - g(w)]^2, \quad k = 0, 1, 2, \ldots$$

denoting the distance between two (sub)images f and g at level k (resolution $2^k \times 2^k$).

The global variable n is initialized to $n := 1$. Given a $2^k \times 2^k$ image to be encoded, a call is then made to *make_wfa(1, k, ∞)* indicating that the entire image, ϕ_1, should be approximated at level k with no restriction on the *cost* (∞). The initial distribution of the WFA produced by the algorithm is

$$I(1) = 1, \quad I(i) = 0, \; i = 2, 3, 4, \ldots$$

and the final distribution is

$$F(i) = \phi_i(\epsilon), \; i = 1, 2, 3, \ldots \; .$$

How the coefficients $r_1, \ldots, r_n$ are calculated when approximating ψ with $r_1\phi_1 + \ldots + r_n\phi_n$ depends on the implementation, as does the determination of the increase s in the size of the WFA. They will therefore be discussed in Section 6.6.

```
Data: Global variable $n$ = number of states in WFA
Data: Global variable $G$ = parameter given as input
function make_wfa($i, k, max$)
if $max \leq 0$ or $k = 0$ then return $\infty$;
$cost := 0$;
forall $a \in \Sigma$ do
    $\psi := (\phi_i)_a$;
    Approximate $\psi$ with $r_1\phi_1 + \ldots + r_n\phi_n$;
    $error := d_{k-1}(\psi, r_1\phi_1 + \ldots + r_n\phi_n)$;
    $s$ := increase caused by adding transitions from state $i$ to states $1, \ldots, n$ with
    label $a$ and weights $r_1, \ldots, r_n$;
    $cost_1 := error + G.s$;
    $n_0 := n$;  $n := n + 1$;
    New state $\phi_n := \psi$;
    Add transition from state $i$ to new state $n$ with label $a$ and weight 1;
    $s$ := increase caused by new state and transition;
    $cost_2 := G.s$ + make_wfa$(n, k-1, \min\{max - cost, cost_1\} - G.s)$;
    if $cost_2 \leq cost_1$ then
        $cost := cost + cost_2$ (accept new state);
    else
        $cost := cost + cost_1$;
        Remove all outgoing transitions from states $n_0 + 1, \ldots, n$;
        Remove transition from state $i$ with label $a$ and weight 1;
        $n := n_0$ (remove all newly created states $n_0 + 1, \ldots, n$);
        Add the transitions from state $i$ with label $a$ to states $1, \ldots, n$ with weights
        $r_1, \ldots, r_n$ (accept linear combination approximation);
    end
end
if $cost \leq max$ then return $cost$ else return $\infty$;
```

Algorithm 6.1: Recursive inference algorithm (make_wfa)

6.5 Initial Basis and Subsequent New Decoding Algorithm

The results achieved by the recursive algorithm above are further improved by introducing an *initial basis*. A set of N initial "images" $\phi_1, \phi_2, \ldots, \phi_N$ are added to the encoder (and decoder) as global variables. The global variable n now refers to the number of non-initial states in the WFA at any given time, and is still initialized to $n := 1$ before calling *make_wfa* for the first time. However, each ψ is now approximated with the linear combination $r_1\phi_1 + \ldots + r_{N+n}\phi_{N+n}$. The initial images need not be and are not defined by WFA. The choice of initial basis will be discussed in Section 6.6. The initial distribution produced will now be

$$I(N+1) = 1, \quad I(i) = 0, \; i \neq N+1$$

while $F(i)$ is still $\phi_i(\epsilon)$, $i = 1, 2, \ldots$, in other words, including the initial images $\phi_1, \phi_2, \ldots, \phi_N$.

The matrices W_a, $a = 0, 1, 2, 3$, are affected as follows: Assuming that n (new) states were created during the execution of the algorithm, the resulting WFA will consist of $N + n$ states (including the initial states). The matrices W_a will therefore all consist of $N + n \times N + n$ entries. However, since the initial states are not specified by WFA, their first N rows are meaningless. We therefore let the encoder produce matrices W_a of dimensions $n \times N + n$. We can now no longer decode by matrix multiplication, and a new decoding algorithm is needed. The decoding algorithm published in [14] satisfies our requirements. It does not make use of matrix multiplication, but instead "fills in" the values in the quadtrees associated with each state in a WFA, from the root (lowest resolution, i.e. level (depth) 0) to a desired level (depth) k, where the desired resolution to decode the image at is $2^k \times 2^k$.

Let $f(s, w)$ denote the function value of state s evaluated at word w. Let N denote the number of initial states and n the number of additional states. The decoding algorithm can then be stated in pseudocode form as shown in Algorithm 6.2. All resolution levels for the initial images can be calculated from the matrices (images) specifying them. The other states ($N+1, \ldots, N+n$) are reconstructed by calculating their values at any given level i in terms of values at level $i-1$ taken from states including the initial states.

```
for $s := 1$ to $N+n$ do
    $f(s, \epsilon) := F(s);$
end
Fill in further levels for initial states by upsampling and downsampling;
for $i := 1$ to $k-1$ do
    for $s := N+1$ to $N+n$ do
        forall $w \in \Sigma^{i-1}$ do
            forall $a \in \Sigma$ do
                $f(s, aw) := \sum_{t=1}^{N+n} W_a(s-N, t) f(t, w);$
            end
        end
    end
end
(* Fill in level $k$ for state $N+1$ associated with entire image *)
$s := N+1;$
forall $w \in \Sigma^{k-1}$ do
    forall $a \in \Sigma$ do
        $f(s, aw) := \sum_{t=1}^{N+n} W_a(s-N, t) f(t, w);$
    end
end
Rearrange the values of $f(s, w), s = N+1, \forall w \in \Sigma^k$, into matrix form;
```

Algorithm 6.2: WFA decoding algorithm

Notice that we no longer need the initial distribution I — the decoder can determine which state corresponds to the entire image since it knows the initial basis and thus the number of initial states. Therefore the encoder does not need to write it to an output file anymore.

Let us illustrate this decoding algorithm by decoding the example WFA of Figure 6.2 in

Section 6.2 with it, at resolution 4×4. The WFA was specified by

$$I = [1 \quad 0], \quad F = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

$$W_0 = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & 1 \end{bmatrix}, \quad W_1 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix},$$
$$W_2 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix}, \quad W_3 = \begin{bmatrix} \frac{1}{2} & 1 \\ 0 & 1 \end{bmatrix}.$$

Notice that this WFA does not have any initial states (images); an example of decoding such a WFA would be very similar to this one. Here we thus have $N = 0$, and, since we wish to decode at resolution $4 \times 4 = 2^2 \times 2^2$, $k = 2$. Decoding proceeds as follows.

Level 0

$$f(1, \epsilon) := F(1) = 1$$

$$f(2, \epsilon) := F(2) = 1$$

Our decoded datastructure thus far is illustrated in Figure 6.8.

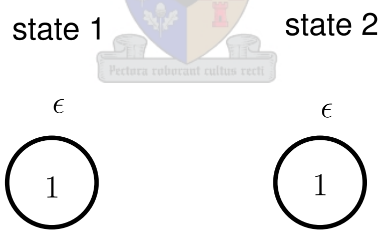


Figure 6.8: Level 0 decoded

```
Level 1
$i := 1$:
    $s := 1$:
        $w := \epsilon$:
            $a := 0$:
                $f(1, 0\epsilon) = f(1, 0) := \sum_{t=1}^{2} W_0(1, t) f(t, \epsilon) = (\frac{1}{2})(1) + (0)(1) = \frac{1}{2}$
            similarly for $a := 1, 2, 3$
    $s := 2$:
        $w := \epsilon$:
            $a := 0$:
                $f(2, 0\epsilon) = f(2, 0) := \sum_{t=1}^{2} W_0(2, t) f(t, \epsilon) = (0)(1) + (1)(1) = 1$
            similarly for $a := 1, 2, 3$
```

Our decoded datastructure now looks as illustrated in Figure 6.9.

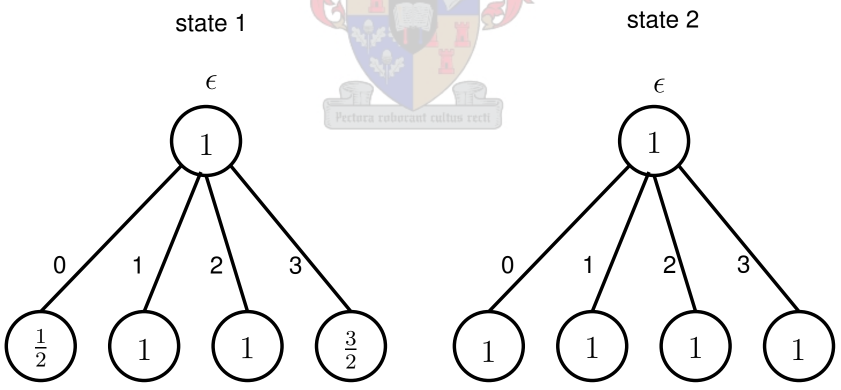


Figure 6.9: Level 1 decoded

```
Level 2
$s := 1$:
    $w := 0$:
        $a := 0$:
            $$f(1,00) := \sum_{t=1}^{2} W_0(1,t)f(t,0) = (\tfrac{1}{2})(\tfrac{1}{2}) + (0)(1) = \tfrac{1}{4}$$
        similarly for $a := 1, 2, 3$
    $w := 1$:
        $a := 0$:
            $$f(1,01) := \sum_{t=1}^{2} W_0(1,t)f(t,1) = (\tfrac{1}{2})(1) + (0)(1) = \tfrac{1}{2}$$
        similarly for $a := 1, 2, 3$
    $w := 2$:
        $a := 0$:
            $$f(1,02) := \sum_{t=1}^{2} W_0(1,t)f(t,2) = (\tfrac{1}{2})(1) + (0)(1) = \tfrac{1}{2}$$
        similarly for $a := 1, 2, 3$
    $w := 3$:
        $a := 0$:
            $$f(1,03) := \sum_{t=1}^{2} W_0(1,t)f(t,3) = (\tfrac{1}{2})(\tfrac{3}{2}) + (0)(1) = \tfrac{3}{4}$$
        similarly for $a := 1, 2, 3$
```

The decoded quadtree for state 1, which corresponds to the entire image, is shown in Figure 6.10.

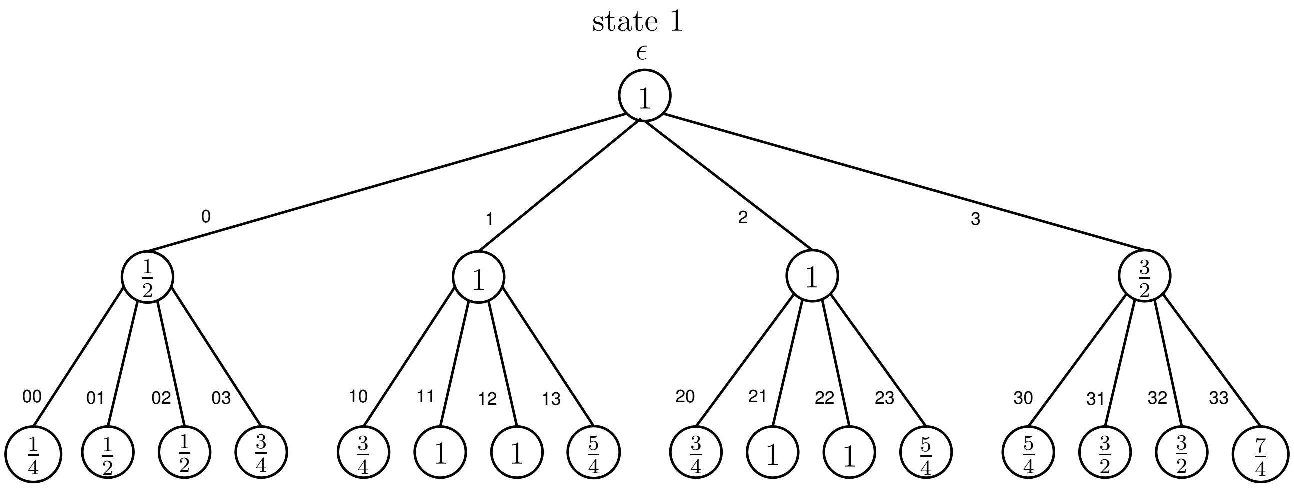


Figure 6.10: Level 2 decoded for state 1

Finally, rearranging the values calculated for level 2, state 1, according to the addressing shown in Table 6.4, yields the same image (matrix) as the one shown in Table 6.6.

6.6 (Matlab) Implementation

Matlab was chosen as the platform to implement the WFA encoding and decoding algorithms in for the following reasons. Matlab offers a very convenient environment for processing images, due to the ease of opening images, displaying images, and accessing pixel values that it provides. Dynamically adding and deleting states in a WFA can easily be implemented in Matlab by dynamically adding and deleting rows and columns in matrices. Matlab is furthermore a powerful tool for approximating vectors with linear combinations of other vectors, and has the built-in symbol ∞ (`Inf`), which is used in the WFA encoding algorithm. Apart from the encoder and decoder each consisting of their own separate m-file, three other auxilliary m-files were written. All m-files are attached in Appendix A.

6.6.1 Auxilliary m-files

The three auxilliary m-files are *get_leaves.m*, *upsample.m* and *downsample.m*. The latter two are respectively responsible for upsampling and downsampling arrays a required number of times, and their implementations are self-explanatory. The m-file *get_leaves.m* calculates the array of leaves associated with an image. It does this by recursively visiting the four quadrants of a sub-image according to the addressing in Table 6.3. If an input sub-image has dimensions 2×2, we have arrived at a set of leaves, and they are entered into the array in the addressing order. Otherwise, its quadrants are given as input to recursive calls in addressing order.

6.6.2 The Encoder

The WFA encoding algorithm (Algorithm 6.1) is implemented in the m-file *wfa_encoder.m*. It takes as input a $2^n \times 2^n$ grayscale image F and gives as output the final distribution FF, the four transition matrices placed in a multidimensional array W, and the variable *bits*, giving the number of bits that the compressed output file is estimated to occupy. Embedded in the m-file is the recursive function *make_wfa*, using parameters ii, k and *maks* and returning the value *cost*. After a sequence of initialization statements, the m-file makes a call to *make_wfa*.

A sequence of global variables are declared so that each invocation of *make_wfa* is aware of them. Variable *tree* is a cell-array implementation of the quadtree associated with input image F, *depth* denotes the depth of the quadtree, and $SymTable$ and $SymCount$ are used for estimating the increase in the size of the output file. Each subimage associated with a newly created state corresponds to a sub-tree of the quadtree *tree*. The position (coordinates) of such a sub-tree's root node in *tree* is stored as a column of the variable *state*. Finally, variables *init_states* and *init_res* indicate the number of initial states and their resolution level, respectively. For instance, if the initial states (images) are defined at resolution $2^k \times 2^k$, then *init_res* equals k.

Among the initialization statements, the following might require clarification.

1. The initial basis consists of 256 linearly independent arrays, each consisting of 256 elements. The values are calculated by sampling cosine functions of increasing frequency and of amplitudes equal to 400 at uniformly spaced points in the interval $[0,\ \pi]$. Any sub-square of 16×16 pixels can therefore be written as a linear combination of these initial states. This is also the initial basis that was finally decided upon.

2. The assumption is made that the numbers stored in W would be compressed with arithmetic coding. We therefore want to keep track of the distinct numbers and their probabilities to calculate the increase in the size of the output file (in bits) when adding new information to W. The variables $SymTable$ and $SymCount$ are used for this — $SymTable$ contains each distinct number (symbol) in its first row and the corresponding number of occurences of each in its second row, while $SymCount$ keeps track of how many numbers (symbols) there are in total. The four components of W are initialized to matrices of dimensions $1 \times init\_states + 1$ each filled with zeros (representing the state in which W occurs before any information has been added to it). $SymTable$ is therefore initialized to recording the number of zeros encountered thus far, as is $SymCount$. The values in FF corresponding to non-initial states also have to be written to the output file, but for the best results they should be stored as (uncompressed) 16-bit values, and are thus not recorded in $SymTable$ and $SymCount$.

3. The values of the cell-array *tree* are filled in by repeatedly downsampling the array of leaves assciated with the image F; the leaves are acquired by making a call to *get_leaves.m* and the downsampling is performed by *downsample.m*.

4. Variable *state* is initialized to column $[1\ \ 1]^{\mathrm{T}}$, recording the root position of state 1 (entire image) in the quadtree.

Finally, after the WFA has been built (upon completion of the overhead call to $make\_wfa$), the variables $SymTable$ and $SymCount$ contain all the statistics necessary to calculate the entropy of the output file. This is then calculated, with $16n$ added to it, yielding the approximate file size stored in variable $bits$.

make_wfa

Each quadrant ψ of a sub-image ϕ_i is approximated with a linear combination of images (states) $\phi_1, \phi_2, \ldots, \phi_{init\_states+n}$ (at level k) in a least squares sense. To do this, the array representing quadrant ψ at level k is read from $tree$ into variable psi, by retreaving the root position of phi_i stored in column ii of the variable $state$. This, combined with the quadrant number a and the level k enables us to read the required number of values at the required depth and offset in $tree$. The arrays representing states $\phi_1, \phi_2, \ldots, \phi_{init\_states+n}$ at level k are used to form the columns of matrix PHI, with those corresponding to non-initial states read from $tree$ similarly to psi. The arrays representing the initial states at level k, on the other hand, are calculated by upsampling or downsampling the initial states/images the appropriate number of times, and are stored in variable f before being placed into matrix PHI. The coefficient vector r is then calculated by using the "backslash"-operator, whereafter the coefficients are quantized by storing them with reduced precision (reduced number of bits).

The increase in the size of the output file that would result from entering r into W is calculated next. Variables $s1$ and $s2$ are used to represent how many bits the file would comprise of before and after, respectively, not taking into account $16n$. The potential new statistics are placed in $NewSymTable$, from which $s2$ is calculated. The increase s in the filesize is then $s = s2 - s1$, and $cost1$ is calculated. $NewSymTable$ is retained in case the decision is made to accept the linear combination approximation.

The quadrant psi is now added as a new state: Its starting position in $tree$ is added to $state$ and its final distribution value is added to FF. This causes each component of W to gain an extra row and column (filled with zeros), and a transition with weight 1 to be added to component $a+1$ of W at row ii and column $init\_states+n$. These extra rows and columns cause an increase in *SymCount*, but since we might reject this new state, the old value of *SymCount* is saved in *OldSymCount*. Furthermore, the transition with weight 1 causes the number of occurences of ones to increase by one and the number of occurences of zeros to decrease by one. The increase in the occurences of zeros is therefore $4(n0+init\_states)+4n-1$. *SymTable* is updated with this information.

Variable $s2$ once again designates the new entropy, and the increase s is now $s=s2+16-s1$. A recursive call is then made to $make\_wfa$, and $cost2$ is calculated. If $cost2 \leq cost1$ we accept the new state and all the states created during the recursive call; thus variables *SymTable*, *SymCount*, n, FF, W and *state* are left as they are. Otherwise, we accept the linear combination approximation and reject all newly created states; the variables *SymCount*, n, FF, W and *state* therefore refer back to the way they were before the recursive call was made. The coefficients in r are then entered into W and *SymTable* is assigned the information in $NewSymTable$.

6.6.3 The Decoder

The WFA decoding algorithm (Algorithm 6.2) is implemented in the m-file *wfa_decoder.m*. It takes as input the data structures FF and W that were output by the encoder, as well as the resolution level n to decode at. It returns as output the decoded image F. The decoder has the same knowledge of the initial basis as the encoder; thus variables *init_states* and *init_res* are assigned the same values as in the encoder, as are the initial states themselves.

The m-file proceeds more or less the same way as the pseudocode in Algorithm 6.2, filling in the values of the quadtrees of each state. A cell-array named *level* is used to store the array of quadtrees, and the values of the initial states at each resolution level are filled in the cell-array by appropriately upsampling and downsampling. The values of the last level, level n, for the state corresponding to the entire image, state $init\_states+1$, are not written into the cell-array, but rather directly into an array named *leeves*. This array therefore contains the leaves associated with the entire image. Finally, these values have to be entered into the output image F, in addressing order. To do this, the decoder contains an embedded recursive function (procedure — it returns nothing) named *pixel*, which basically performs the opposite task of auxilliary m-file *get_leaves.m*.

Each invocation of *pixel* attempts to fill in the pixel values of a sub-square of F, and therefore takes as input the size (nn refers to a size of $2^{nn} \times 2^{nn}$), starting row position and starting column position (relative to F) of the sub-square. If the sub-square has dimensions 2×2 (if $nn = 1$), the applicable pixel values are written into it in addressing order, otherwise recursive calls are made to *pixel* to process its quadrants in addressing order. Variable *pos* keeps track of the positions in *leeves* at which the next set of leaves should be copied over. Since each invocation of *pixel* needs to have access to F, *pos* and *leeves*, they are declared as global variables.

Chapter 7

Results and Conclusions

The compression results obtained by JPEG, fractal image compression with HV partitioning and WFA image compression were investigated for six 256×256 images. These images, shown in Appendix B, are *Boat*, *Cameraman*, *Plane*, *Peppers*, *Bridge* and *Lenna*. Experiments were not conducted for any 512×512 images, because the Matlab implementations of the WFA encoder and decoder unfortunately have prohibitively long execution times for such images. An implementation of the wavelet compression technique using the EZW algorithm could also unfortunately not be located, and this technique is therefore not included in this comparison.

It is known from the historical background of JPEG [1] that the quantized DCT consistently gives better results than vector quantization (as well as the other techniques it competed against), which led to it being adopted as the JPEG standard. Therefore, vector quantization has also not been included in these comparisons; it was, however, included in this thesis to give as wide as possible a perspective on image compression, as was wavelet image compression. A VQ codebook was also experimented with as an initial basis for WFA image compression, further motivating the inclusion of the VQ technique in this thesis.

7.1 Quality Measure

When measuring the quality of a compressed image (relative to the original), the *peak signal-to-noise ratio* (PSNR) is conventionally used. Given an image $X = (x_1, x_2, \ldots, x_n)$ and its approximation $Y = (y_1, y_2, \ldots, y_n)$, the *mean squared error* (mse), which is also sometimes used, is given by

$$\text{mse} = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2.$$

From this, the PSNR for an 8-bit grayscale image is defined by

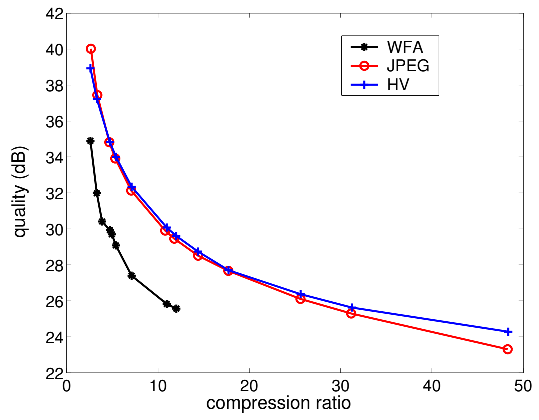
$$\text{PSNR} = 10\log_{10}\left(\frac{255^2}{\text{mse}}\right),$$

where 255 is the maximum value an 8-bit pixel can assume. The units for the PSNR are *decibels* (dB).

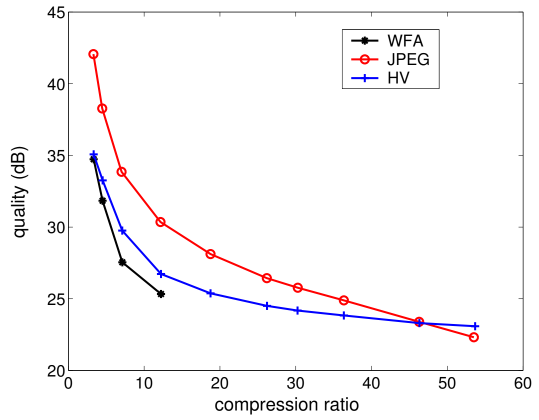
We see that the PSNR is inversely proportional to the error made in an approximation; thus the higher the PSNR, the better the image quality will be, and the lower the PSNR, the poorer the image quality will be.

7.2 Results

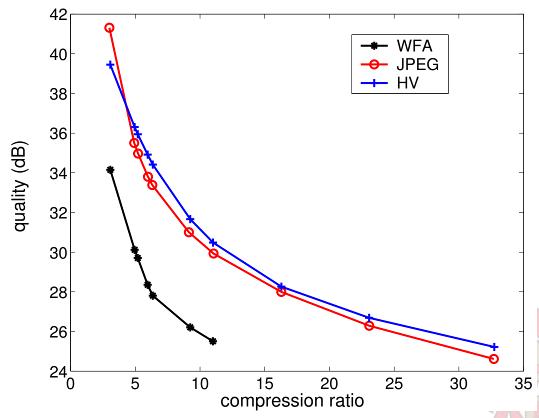
The results of the experiments conducted on the aforementioned six images are presented in Figure 7.1 below. Figures (a) to (f) show how the quality (in dB) decreases with increased compression ratio for each of the three compression techniques in question when applied to each of these images. Data points are shown interpolated by straight lines, with the curves for JPEG coloured red, those for WFA image compression coloured black and those for HV fractal compression coloured blue. (Discussion of results to follow Figure 7.5.)



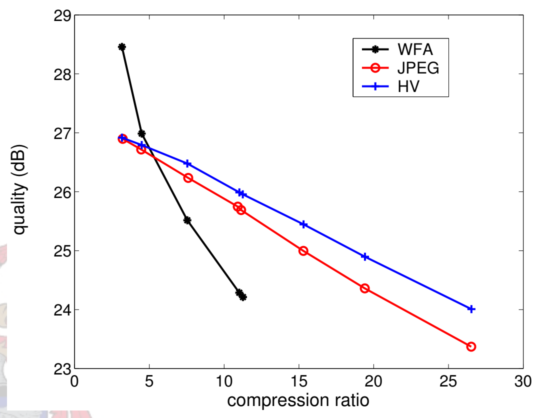
a) Boat



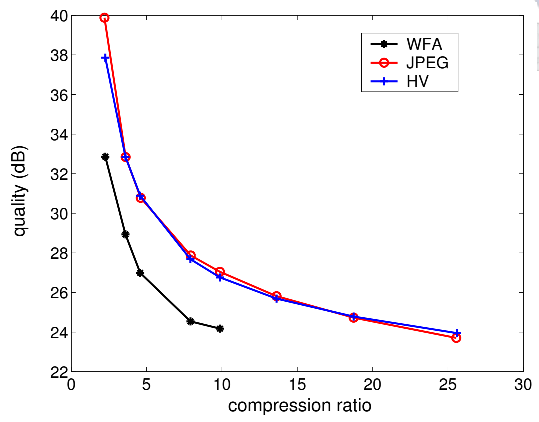
b) Cameraman



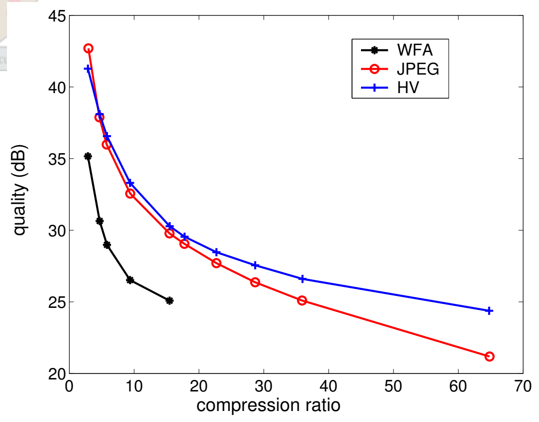
c) Plane



d) Peppers



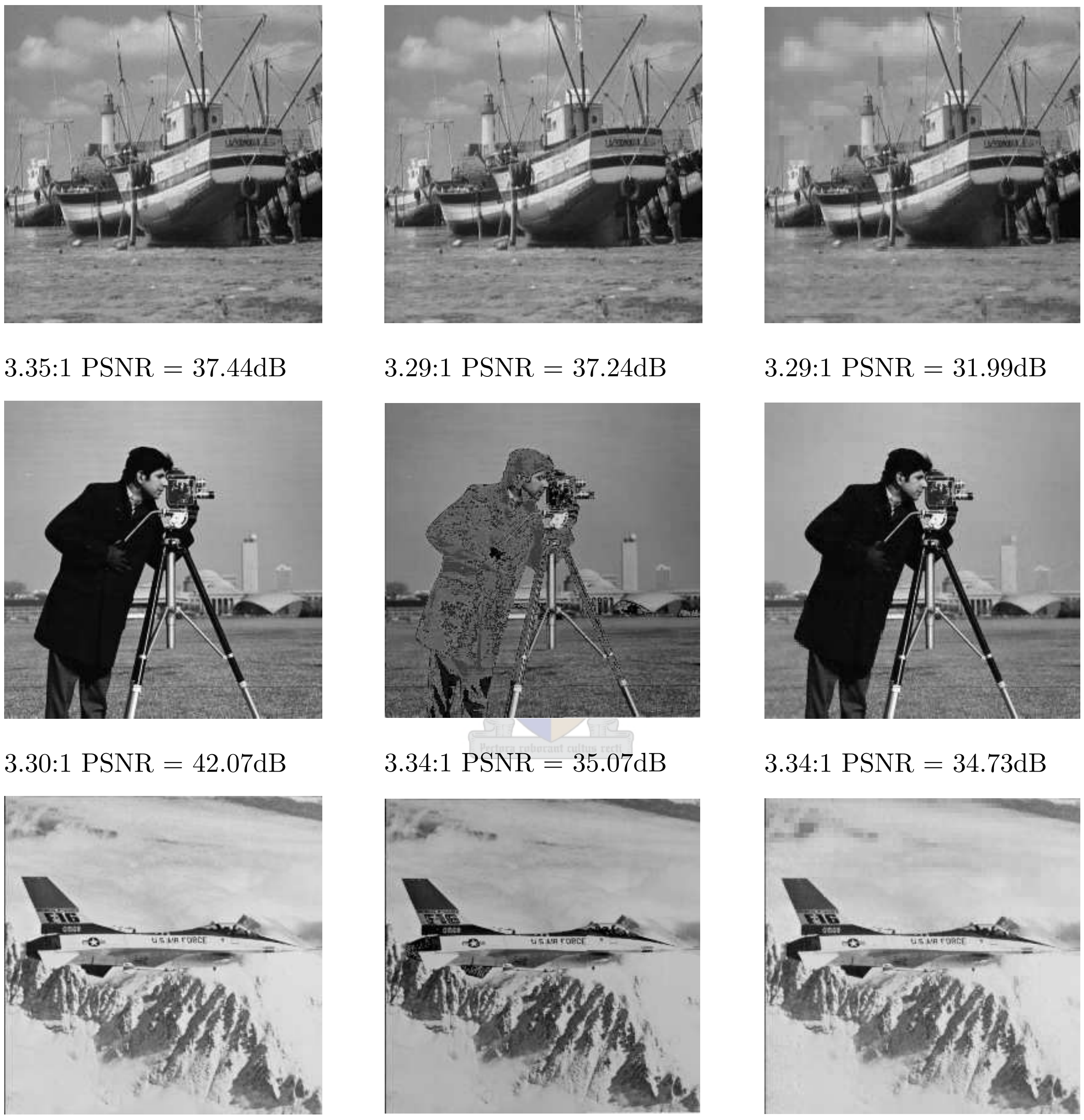
e) Bridge



f) Lenna

Figure 7.1: Graphical representation of compression results

In Figures 7.2 to 7.5 some decoded images are also compared; for each of the six test images, a high fidelity encoding generated by each technique is shown in Figures 7.2 and 7.3, while a low fidelity encoding generated by JPEG and HV fractal compression each is shown in Figures 7.4 and 7.5 for each of the test images.

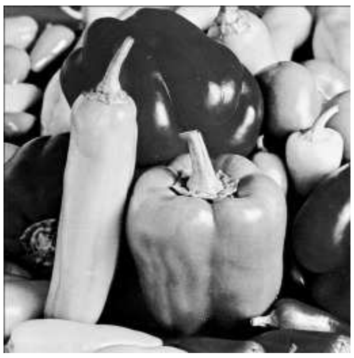


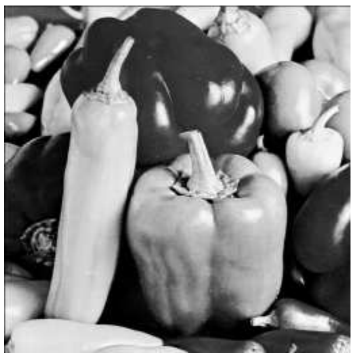
3.35:1 PSNR = 37.44dB | 3.29:1 PSNR = 37.24dB | 3.29:1 PSNR = 31.99dB

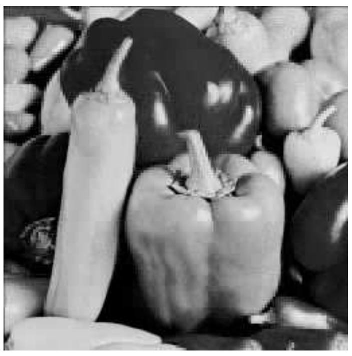
3.30:1 PSNR = 42.07dB | 3.34:1 PSNR = 35.07dB | 3.34:1 PSNR = 34.73dB

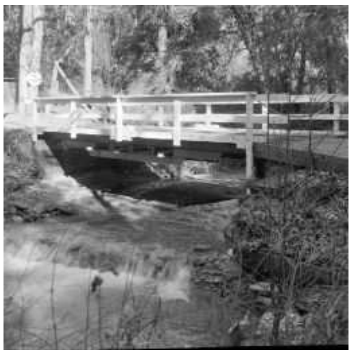
3.00:1 PSNR = 41.31dB | 3.07:1 PSNR = 39.45dB | 3.07:1 PSNR = 34.15dB

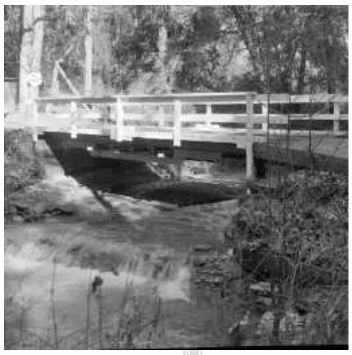
Figure 7.2: Decoded high fidelity images; JPEG left, HV middle, WFA right

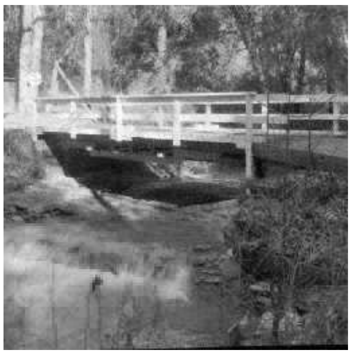

3.22:1 PSNR = 26.90dB

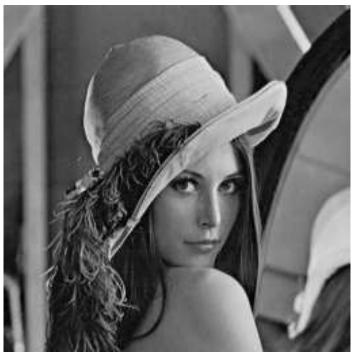

3.17:1 PSNR = 26.92dB

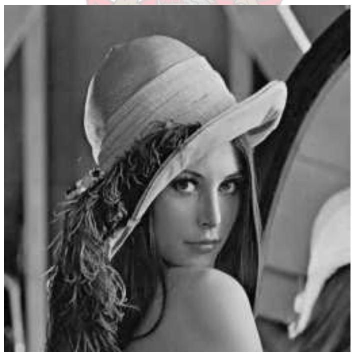

3.17:1 PSNR = 28.46dB


3.61:1 PSNR = 32.84dB


3.60:1 PSNR = 32.85dB


3.60:1 PSNR = 28.93dB


4.65:1 PSNR = 37.88dB


4.69:1 PSNR = 38.10dB

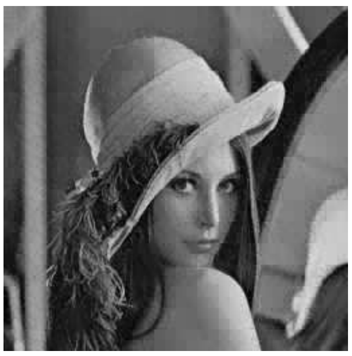
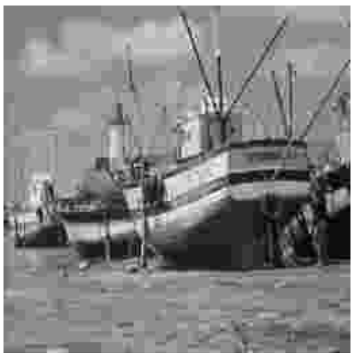
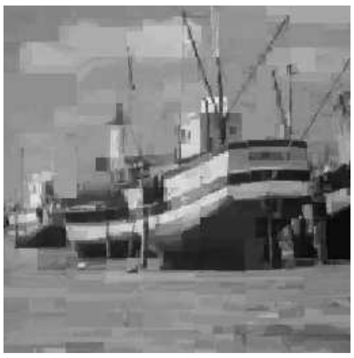
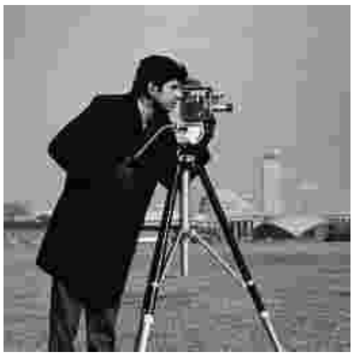
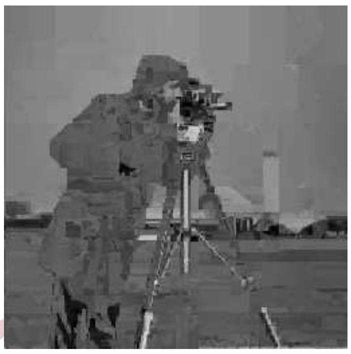

4.69:1 PSNR = 30.64dB

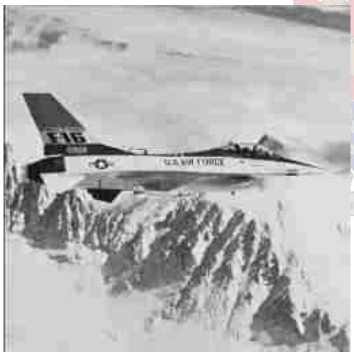
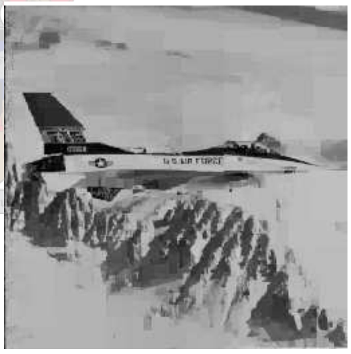
Figure 7.3: Decoded high fidelity images (continued)

31.16:1 PSNR = 25.30dB 31.22:1 PSNR = 25.63dB

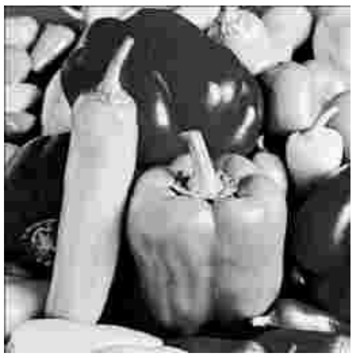



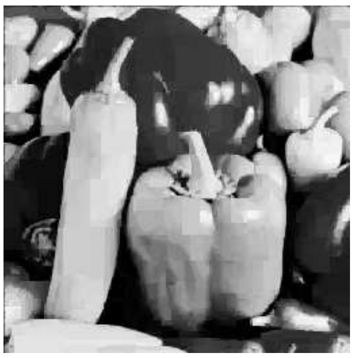
30.27:1 PSNR = 25.77dB 30.24:1 PSNR = 24.18dB

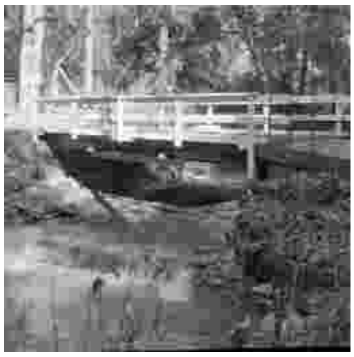



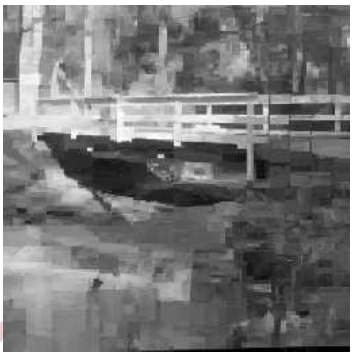
16.28:1 PSNR = 28.00dB 16.29:1 PSNR = 28.26dB

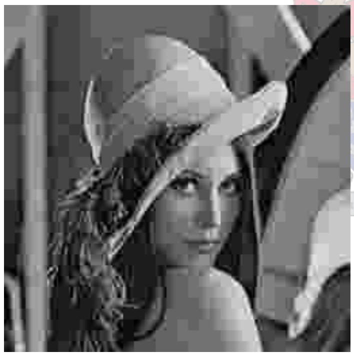
Figure 7.4: Decoded low fidelity images; JPEG left, HV right


19.41:1 PSNR = 24.36dB


19.42:1 PSNR = 24.90dB


25.55:1 PSNR = 23.70dB


25.58:1 PSNR = 23.95dB


28.67:1 PSNR = 26.37dB

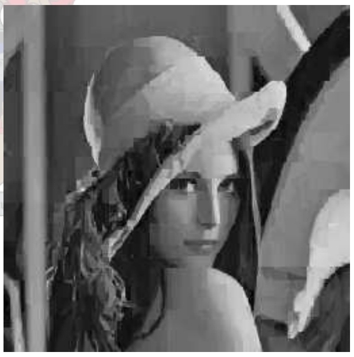

28.67:1 PSNR = 27.55dB

Figure 7.5: Decoded low fidelity images (continued)

The results for WFA image compression were acquired using the cosine initial basis as used in *wfa_encoder.m* and *wfa_decoder.m* (Section 6.6.2). Of all the initial bases experimented with, that one gave the best results; both in terms of "approximating power" and distribution of the coefficients of the linear combinations. For the amplitude of 400 and

256 initial states, the coefficients consistently lie in the interval $(-1,\ 1]$. Those that are not zero or one are of the form

$$c = \pm 0.b_1 b_2 b_3 \ldots,$$

where $b_1, b_2, b_3, \ldots$ are binary digits. When quantizing to n bits, they are reduced to

$$c = \pm 0.b_1 b_2 \ldots b_n.$$

(This form of quantization has no effect on those coefficients that are zero or one.) It was found that this distribution was more compressible, i.e. had a lower entropy, than those resulting from different initial bases. More than 256 initial states and/or amplitudes greater than 400 did not yield any better results, whereas fewer initial states and/or smaller amplitudes yielded poorer results. The values used for parameter G differed greatly, and finding a value of G giving an encoding of a certain quality (or filesize) was a process of trial and error. Quantization was used as follows. For low compression (high fidelity) encodings, coefficients were quantized to $k+4$ bits, where k refers to the resolution level at which the particular approximation took place. This type of quantization was used by Kari and Fränti in [13]. For medium fidelity encodings, the quantization parameter was $k+3$; for low fidelity encodings, $k+2$ was used in some cases, and in others coefficients were uniformly quantized to 8 bits, depending on which gave the best results. As stated before, a VQ codebook was also experimented with as an initial basis. This codebook was generated with the DCT-based technique explained in Section 3.2.3., and consisted of 64 8×8 elements, which were then converted to arrays of leaves with 64 entries. Another initial basis experimented with consisted of six elements which were generated by sampling the functions (of x and y) 255, $255x$, $255y$, $255xy$, $255x^2$ and $255y^2$ on the unit square. Both of these initial bases were found to be inferior.

The results for JPEG were obtained with the Linux program Gimp with settings chosen to give optimal results:

1. No restart markers

2. Progressive coding (Baseline disabled)

3. Speed versus quality trade-off set in favour of quality

The HV fractal encoder/decoder used was written by Markus Fick, and can be obtained from `www.stud.uni-siegen.de/markus.fick`. It has a parameter p ranging between 0 and 9 controlling the speed versus quality trade-off. For all experiments $p = 3$ was chosen, since it gave encoding times comparable to those of *wfa_encoder.m*. In both cases (for both programs) higher compression ratios require shorter encoding times, while lower compression ratios require longer encoding times. Even if we were to explore the HV fractal encoder's full potential, by setting $p = 9$, one would typically do so for high compression ratios — for low compression ratios the encoding times for $p = 9$ might easily be in terms of days! We therefore chose $p = 3$ in order to be consistent.

The filesizes for HV fractal compression and JPEG were the actual filesizes obtained. Those for WFA compression correspond to the estimates given by *wfa_encoder.m* and would in practice be slightly larger (overhead for arithmetic coding, etc). Compression ratios were calculated by dividing 65536 bytes ($= 256 \times 256$ pixels $\times$ 8 bits/pixel) by the actual filesizes (in bytes) in the case of JPEG and HV fractal compression, and the estimated filesizes in the case of WFA compression.

7.3 Conclusions

Even with the slight "unfair advantage" in terms of the reported filesizes mentioned above, WFA image compression consistently gave inferior results. Because of this, the quality of the compressed images falls below anything useful, which is more or less 25dB, at what are regarded as medium compression ratios. Experiments at higher compression ratios have therefore not been conducted. A notable exception is the high fidelity performance given for image *Peppers*, seen in Figure 7.1(d). The quality degredation soon falls far below those of JPEG and HV fractal compression, however.

Results are, on the other hand, listed for HV fractal compression and JPEG for high compression ratios, with Figure 7.1 thus showing how they compare for compression ratios ranging from low to high. Let us discuss this.

For all images JPEG performs better at lower compression ratios (high fidelity) than HV fractal compression. However, HV compression sooner or later “catches up” and eventually outperforms JPEG at high compression ratios (low fidelity). For most images these two techniques give comparable results throughout medium compression ratios (*Boat*, *Plane*, *Bridge* and *Lena*), while for *Peppers* HV compression starts outperforming JPEG very early on. For image *Cameraman*, HV compression gives very poor results at low and medium compression ratios before it starts to give better results than JPEG at very high compression ratios. Looking at the decoded HV fractal encodings of *Cameraman* in Figures 7.2 and 7.4, it appears that the implemented encoding and/or decoding had an anomalous reaction to this image, which might account for the poor numerical results.

It is suspected that, for parameter $p = 9$, this implementation of HV fractal compression would in most cases consistently yield results superior to JPEG, and in the case of high compression ratios, vastly superior results.

Based on results reported in [8] and [10] one can expect wavelet image compression techniques to slightly outperform JPEG for medium and high compression ratios and considerably in the case of high fidelity encodings (low compression ratios). Furthermore, the encoding (and decoding) times of these techniques appear to be comparable to those of JPEG.

Thus, it would appear that, when short encoding times as well as high to medium fidelity are of importance, wavelet image compression would be one's first choice, with JPEG a close second. If, on the other hand, one is willing to tolerate longer encoding times and seeking the best low fidelity encodings, HV fractal encoding would be recommended. Until WFA compression can be improved, it will remain a technique which is of no practical importance, but only interesting from a theoretical point of view.

Appendix A

M-files

```
function leeves = get_leaves(A)
n = length(A(1,:));
if n == 2
    leeves = [A(2,1) A(1,1) A(2,2) A(1,2)];
else
    leeves = [];
    leeves = [leeves get_leaves(A((n/2 + 1):end, 1:n/2))];
    leeves = [leeves get_leaves(A(1:n/2, 1:n/2))];
    leeves = [leeves get_leaves(A((n/2 + 1):end, (n/2 + 1):end))];
    leeves = [leeves get_leaves(A(1:n/2, (n/2 + 1):end))];
end


function f = downsample(x, n)
len = length(x);
f = x;
for k = 1:n
    x = f;
    f = [];
    for m = 1:length(x)/4
        f(m) = mean(x(4*(m-1)+1:4*(m-1)+4));
    end
end


function f = upsample(x, n)
    if n == 0
        f = x';
    elseif n < 0
        f = downsample(x, -n);
    else
        len = length(x);
        for m = 0:len - 1
            f((4^n)*m+1:(4^n)*m+4^n) = x(m+1);
        end
    end
```

```
function [W, FF, bits] = wfa_encoder(F)

% recursive algorithm with cosine initial basis

disp('running')

global W FF n depth G tree state SymTable SymCount init_states init_res;

init_states = 256;
init_res = 4;

W = [];
W(:,:,1) = zeros(1, init_states + 1); W(:,:,2) = zeros(1, init_states + 1);
W(:,:,3) = zeros(1, init_states + 1); W(:,:,4) = zeros(1, init_states + 1);

FF = [];
for kk = 1:init_states
    FF = [FF ; mean(400*cos((kk-1)*linspace(0, pi, 256))')];
end
FF = [FF ; mean(mean(F))];

SymTable = [0 ; 4*(init_states + 1)];
SymCount = 4*(init_states + 1);

G = 180;

depth = log2(length(F(1,:)));
tree{depth + 1} = get_leaves(F);
for level = depth:-1:1
    tree{level} = downsample(tree{level + 1}, 1);
end

n = 1;
state = [1; 1];
totalcost = make_wfa(1, depth, Inf);

% check for no zeros
if SymTable(2, 1) == 0
    bits = SymTable(2, 2:end)*log2(SymCount./SymTable(2, 2:end))';
else
    bits = SymTable(2,:)*log2(SymCount./SymTable(2,:))';
end
bits = bits + 16*n;


function cost = make_wfa(ii, k, maks)

    global W FF n depth G tree state SymTable SymCount init_states init_res;

    if maks <= 0 | k == 0
        cost = Inf; return
    end
```

```
cost = 0;

f = [];
for kk = 1:init_states
    f = [f upsample(400*cos((kk-1)*linspace(0, pi, 256))', k - init_res - 1)'];
end

for a = 0:3
    pos = state(:, ii);
    pos_psi(1) = pos(1) + 1;
    pos_psi(2) = 4*(pos(2) - 1) + 1 + a;
    psi = tree{pos_psi(1) + k - 1}(((4^(k-1))*(pos_psi(2)-1) + 1):((4^(k-1))* ...
     (pos_psi(2)-1) + 1 + 4^(k-1) - 1))';
    PHI = [];
    for m = 1:n
        pos = state(:, m);
        if pos(1) + k - 1 <= depth + 1
            PHI = [PHI ; tree{pos(1) + k - 1}(((4^(k-1))*(pos(2)-1) + ...
             1):((4^(k-1))*(pos(2)-1) + 1 + 4^(k-1) - 1))];
        else
            overshoot = pos(1) + k - 1 - (depth + 1);
            temp1 = tree{depth + 1}(((4^(k-1-overshoot))*(pos(2)-1) + 1): ...
             ((4^(k-1-overshoot))*(pos(2)-1) + 1 + 4^(k-1-overshoot) - 1));
            PHI = [PHI ; upsample(temp1, overshoot)];
        end
    end
    PHI = [f PHI'];
    r = PHI\psi;
    r = r';

    % quantize
    r = double(int32(r*(2^(k + 4))))/(2^(k + 4));

    s1 = SymTable(2,:)*log2(SymCount./SymTable(2,:))';
    NewSymTable = SymTable;
    NewSymTable(2, 1) = NewSymTable(2, 1) - length(r);
    file = unique(r);
    for p = 1:length(file)
        tablepos = find(NewSymTable(1,:) == file(p));
        occur = length(find(r == file(p)));
        if isempty(tablepos)
            NewSymTable = [NewSymTable [file(p) ; occur]];
        else
            NewSymTable(2, tablepos) = NewSymTable(2, tablepos) + occur;
        end
    end

    % check for no zeros
    if NewSymTable(2, 1) == 0
        s2 = NewSymTable(2, 2:end)*log2(SymCount./NewSymTable(2, 2:end))';
    else
        s2 = NewSymTable(2,:)*log2(SymCount./NewSymTable(2,:))';
        end
        s = s2 - s1;
        cost1 = G*s + norm(psi - PHI*r', 2)^2;

        n0 = n;
        n = n + 1;
        state = [state pos_psi'];
        FF = [FF ; tree{pos_psi(1)}(pos_psi(2))];
        W(1, n + init_states, 1) = 0;
        W(n, 1, 1) = 0;
        W(ii, n + init_states, a + 1) = 1;
        OldSymCount = SymCount;
        SymCount = SymCount + 4*(n0 + init_states) + 4*(n);
        tablepos = find(SymTable(1,:) == 1);
        if isempty(tablepos)
            SymTable = [SymTable [1 ; 1]];
        else
            SymTable(2, tablepos) = SymTable(2, tablepos) + 1;
        end
        SymTable(2,1) = SymTable(2,1) + 4*(n0 + init_states) + 4*(n) - 1;
        s2 = SymTable(2,:)*log2(SymCount./SymTable(2,:))';
        s = s2 - s1 + 16;

        cost2 = G*s + make_wfa(n, k - 1, min([maks - cost, cost1]) - G*s);

        if cost2 <= cost1
            cost = cost + cost2;
        else
            cost = cost + cost1;
            FF = FF(1:n0 + init_states);
            W = W(1:n0, 1:n0 + init_states, :);
            state = state(:, 1:n0);
            n = n0;
            SymCount = OldSymCount;
            SymTable = NewSymTable;
            W(ii, :, a + 1) = r;
        end
    end
    if cost > maks
        cost = Inf;
    end

function F = wfa_decoder(W, FF, n)

disp('running')

global F pos leeves;

init_states = 256;

init_res = 4;
```

```
total_states = length(FF);

level0 = FF';

% level1 :
level{1} = [];
for kk = 1:init_states
    level{1} = [level{1} downsample(400*cos((kk-1)*linspace(0, pi, 256))',  init_res - 1)];
end

for state = (init_states + 1):total_states
    for a = 0:3
        level{1}(4*(state-1) + 1 + a) = W(state - init_states, :, a + 1)*level0';
    end
end

% further levels :

for k = 2:(n-1)
    level{k} = zeros(1,(4^k)*total_states);
end

for k = 2:(n-1)
    assign = [];
    for kk = 1:init_states
        assign = [assign upsample(400*cos((kk-1)*linspace(0, pi, 256))', k - init_res)];
    end
    level{k}(1:init_states*(4^k)) = assign;
end

clear assign;
clear level0;

for k = 2:(n-1)
    k
    for state = (init_states + 1):total_states
        pos = 0;
        aaa = 0;
        for iterate = 1:4^(k-2)
            for a = 0:3
                aa = 0;
                for i = ((state-1)*(4^k) + 1 : 4^(k-1) : state*(4^k)) + a + pos
                    thingy = level{k-1}((1 : 4^(k-1) : end) + aaa);
                    level{k}(i) = W(state - init_states, :, aa + 1)*thingy';
                    aa = mod(aa + 1, 4);
                    if mod(aa, 4) == 0
                        aaa = aaa + 1;
                    end
                end
            end
            pos = 4*iterate;
        end
    end
end

k = n
leeves = zeros(1, 4^k);
state = init_states + 1;
pos = 0;
aaa = 0;
for iterate = 1:4^(k-2)
    for a = 0:3
        aa = 0;
        for i = (1 : 4^(k-1) : 4^k) + a + pos
            thingy = level{k-1}((1 : 4^(k-1) : end) + aaa);
            leeves(i) = W(state - init_states, :, aa + 1)*thingy';
            aa = mod(aa + 1, 4);
            if mod(aa, 4) == 0
                aaa = aaa + 1;
            end
        end
    end
    pos = 4*iterate;
end

clear level;
clear thingy;
pos = 1;
F = zeros(2^n, 2^n);
pixel(n, 1, 1);


function [] = pixel(nn, row, col)

global F pos leeves;

if nn == 1
    F(row + 1, col) = leeves(pos);
    F(row, col) = leeves(pos + 1);
    F(row + 1, col + 1) = leeves(pos + 2);
    F(row, col + 1) = leeves(pos + 3);
    pos = pos + 4;
else
    pixel(nn-1, row + 2^(nn-1), col);
    pixel(nn-1, row, col);
    pixel(nn-1, row + 2^(nn-1), col + 2^(nn-1));
    pixel(nn-1, row, col + 2^(nn-1));
end
```

Appendix B

Test Images

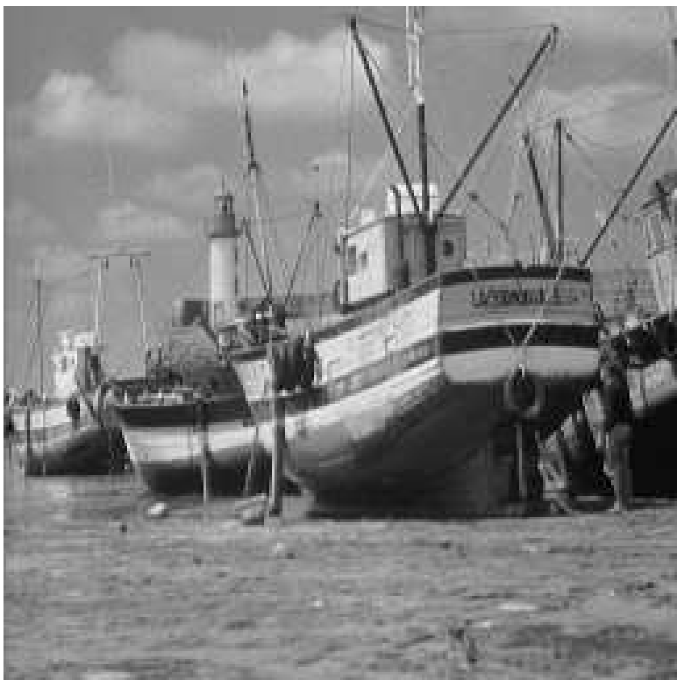


Figure B.1: Boat

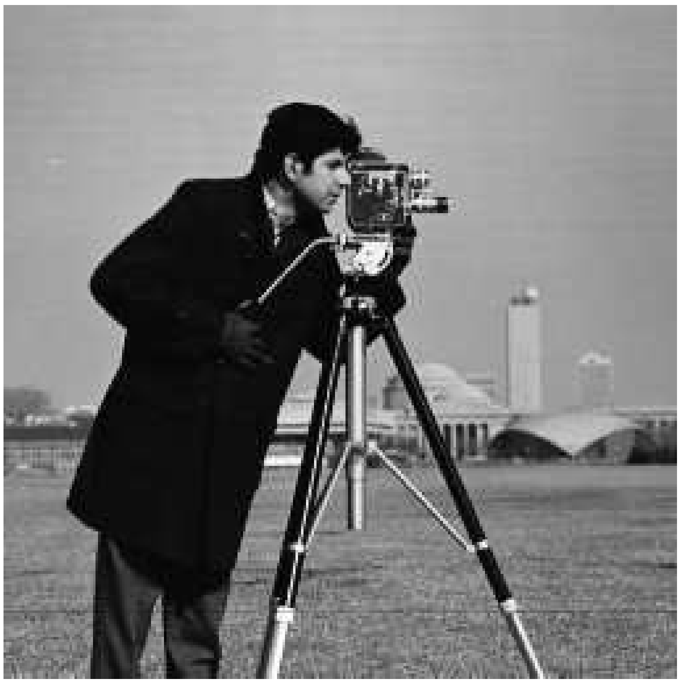


Figure B.2: Cameraman

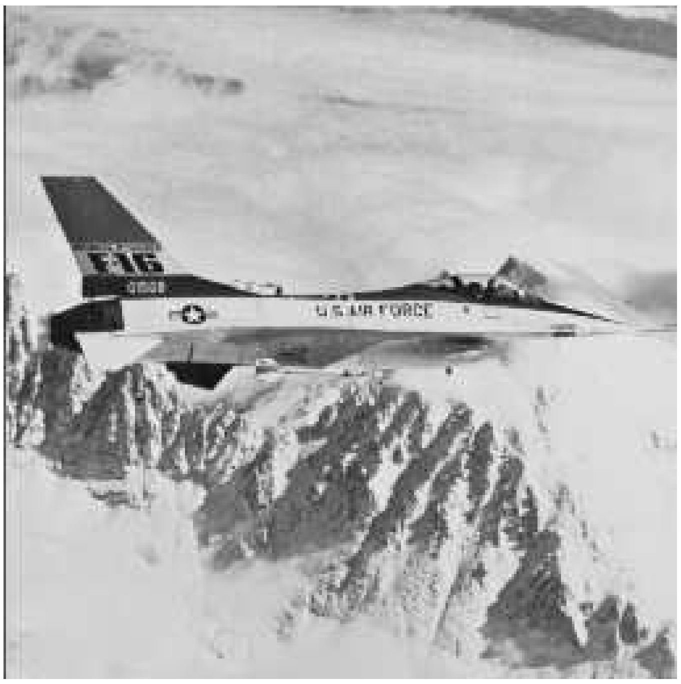


Figure B.3: Plane

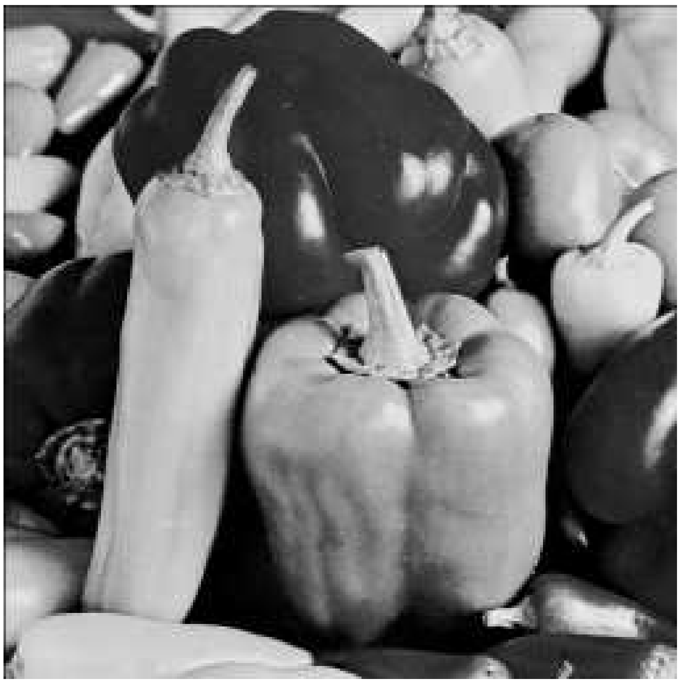


Figure B.4: Peppers

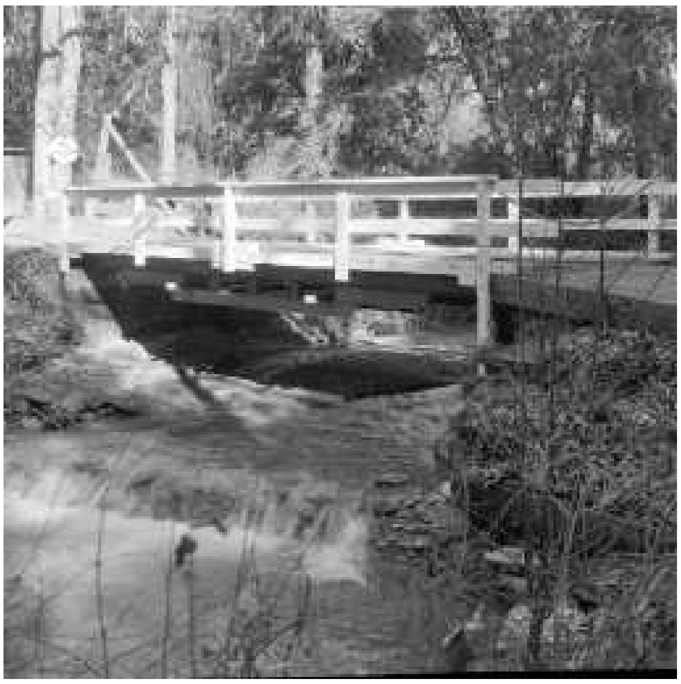


Figure B.5: Bridge

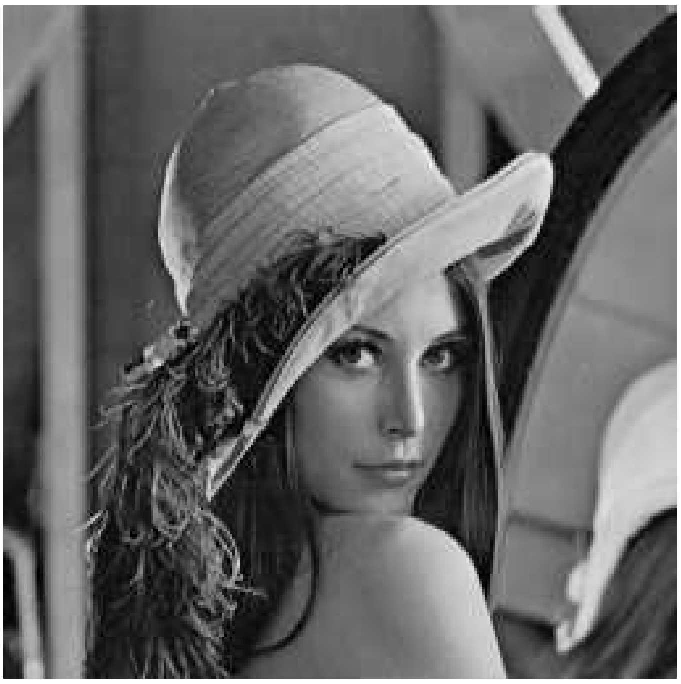


Figure B.6: Lenna

Bibliography

[1] W B Pennebaker and J Mitchell. *JPEG: Still Image Data Compression Standard.* Van Nostrand-Reinhold, New York, 1993.

[2] C E Leiserson T H Cormen and R L Rivest. *Introduction To Algorithms.* The MIT Press, Cambridge, 1990.

[3] Glen G Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28:135–149, 1984.

[4] Robert M Gray. Vector quantization. *IEEE, ASSP Magazine*, pages 4–28, 1984.

[5] Chaur-Heh Hsieh. DCT-based codebook design for vector quantization of images. *IEEE transactions on circuits and systems for video technology*, 2:401–405, 1992.

[6] Johan M de Villiers. *Subdivisie, golfies en latfunksies.* Department of Mathematics, University of Stellenbosch, 2002. Lecture notes.

[7] Barry Sherlock. *Wavelets and Filter Banks.* Department of Electrical and Electronic Engineering, University of Stellenbosch, 2002. Lecture notes.

[8] Jerome M Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41:3445–3459, 1993.

[9] C Valens. Embedded zerotree wavelet encoding, 1999. Tutorial, c.valens@mindless.com.

[10] Yuval Fisher. *Fractal Image Compression — Theory and Application.* Springer-Verlag, New York, 1994.

[11] M F Barnsley. *Fractals Everywhere.* Academic Press, San Diego, 1988.

[12] Carel Culik and Jarkko Kari. Image-data compression using edge-optimzing algorithm for WFA inference. *Information Processing and Management*, 30:829–838, 1994.

[13] Jarkko Kari and P Fränti. Arithmetic coding of weighted finite automata. *Theoretical informatics and Applications*, 28:343–360, 1994.

[14] Carel Culik and Jarkko Kari. Image compression using weighted finite automata. *Computer and Graphics*, 17:305–313, 1993.