Scalable video using Wavelets

Henrik Andrén
Kim Taavo

February 21, 2002

Abstract

This Master thesis mainly attempts to become a kind of manual for the interested reader to try out wavelets, and different computationally effective ways of doing transforms. Secondarily it is an introduction on how you can make an pretty decent video encoder, as well as some test results and comparisons to JPEG and MPEG equivalents. The wavelets we have been using are included in the appendix.

Sammanfattning

Det här examensarbetet handlar om wavelets och hur de kan användas för videokomprimering. Den första delen handlar om hur wavelets är uppbyggda, och är mestadels ämnat som en introduktion till hur man kan experimentera med wavelets och de olika formerna av transformer. Senare kommer man in på hur man kan tillverka en relativt effektiv videokodare med hjälp av wavelets. De wavelets som vi använde finns att hitta i appendix.

Preface

This thesis was worked on and written at Telia Research in Luleå, as a part of graduation to Master of Science. It is a development from an idea of service transparency that we later developed into video coding on our own.

We would like to express our thanks to our supervisor Jonas Månsson at Telia Research AB, as well as Thomas Gunnarson at the department of mathematics and Lenka Carr Motyčková at the division of computer science and networking at Luleå Tekniska Universitet. Our thanks also goes to all of the coworkers at Telia Research, who have been nothing but kind and welcoming to us, and especially Tommy Wall, who's humor and insights have been uplifting and interesting. Maybe at times even too much.

This document is entirely written in LaTeX, over night at times.

Contents

1 Philosophy **4**
- 1.1 Compression . . . 4
- 1.2 Service transparency . . . 5

2 Theory **5**
- 2.1 Wavelet Theory . . . 5
  - 2.1.1 Orthogonality . . . 6
  - 2.1.2 Wavelets . . . 8
  - 2.1.3 Multiresolution analysis . . . 12
- 2.2 Filter Banks . . . 14
  - 2.2.1 The Z-transform . . . 14
- 2.3 From wavelets to filters . . . 15
  - 2.3.1 Signal analysis and filters . . . 18
  - 2.3.2 Using filter banks . . . 19
  - 2.3.3 Fringe effects . . . 20
  - 2.3.4 Inverse filter . . . 22
  - 2.3.5 Integer filters . . . 23
  - 2.3.6 Reflexions about the Haar transform . . . 24
- 2.4 Lifting . . . 24
  - 2.4.1 Inverse lifting . . . 26
  - 2.4.2 Integer lifting . . . 27
- 2.5 Two dimensional transform . . . 28
- 2.6 Inseparable two dimensional transform . . . 30
  - 2.6.1 Two dimensional lifting . . . 31
  - 2.6.2 Neville 2-d filter . . . 33
  - 2.6.3 Inverse two dimensional lifting . . . 34
  - 2.6.4 Fringe effects in two dimensions . . . 34
- 2.7 Multi dimensional transforms . . . 35
  - 2.7.1 Uneven transforms . . . 35
  - 2.7.2 Interpolation . . . 37

3 Video coding **38**
- 3.1 Compression . . . 38
- 3.2 Computer images and video . . . 39
- 3.3 Video and wavelets . . . 40
- 3.4 Video codec . . . 42
- 3.5 Color space transform . . . 42
- 3.6 Discrete wavelet transform . . . 43
- 3.7 Quantization . . . 47
  - 3.7.1 SPIHT . . . 49
- 3.8 Entropy encoding . . . 54
- 3.9 Fixed point arithmetic . . . 54

3.10 Peak Signal to Noise Ratio 56

4 Implementation **56**

5 Results **59**
5.1 Notes 61

6 Conclusions **61**

7 Further development **62**
7.1 Resolution 62
7.2 Uneven scaling 62
7.3 Timing 63
7.4 Optionality 63

8 History **63**
8.1 Fourier transform 64
8.2 Early wavelets 65
8.3 The birth of wavelets 65
8.4 Modern wavelets 66
8.5 Future 66

A Wavelets **69**
A.1 Haar filterbanks 69
A.2 Haar lifting 69
A.3 Neville lifting 70

B Matlab code **71**
B.1 Wavelet transform 71
B.1.1 Function to wavelet space transform 71
B.2 Different wavelets in matlab 72
B.3 Inverse wavelet transform 73
B.3.1 Wavelet to function transform 74

C Pseudo code **76**
C.1 SPIHT 76
C.2 Wavelet sort algorithm 78

List of Figures

1 Polynomial sin approximation 8
2 Haar scaling function 9
3 Haar wavelet 10
4 Function approximation 11
5 Transforming filter bank 20
6 Multiple filter bank 22
7 Inverse filterbank 23
8 Lifting transform 25
9 Inverse lifting 27
10 Subband coding 29
11 Quinox lattice 31
12 Multi dimensional lifting 32
13 Multi dimensional lifting 34
14 Uneven transform 36
15 Interpolation example 37
16 2D coefficient grouping 41
17 3D coefficient grouping 41
18 Video codec 42
19 1D DWT - First level 44
20 1D DWT - All levels 44
21 2D DWT - All levels 45
22 Quantized signal 48
23 Scan orders 48
24 Lena sub band 50
25 parent child/tree 50
26 Numbered order 52
27 Transformed wavelet data 53
28 Tree ordered wavelet data 53
29 PSNR values of different compressed pictures of Lena 57
30 Wavelet video packet 58
31 Screen shot of GUI 59
32 Wavelet vs. JPEG 60
33 Wavelet vs. MPEG-1 61
34 Pyramid function 65

List of Tables

1 Interlace scheme 39
2 Digital video standards 40
3 Color spaces 42
4 Fixed point arithmetic 55

1 Philosophy

The idea of image compression is to limit the information necessary to describe the picture, either in full detail, or just the most important parts. This is easy enough for the human eye. If we want to identify a person we know exactly what to look at, cheekbones, eyebrows, lines around the eyes and so on. The trouble is teaching a computer this. Some experiments have been based on identifying the lines in a picture, but wavelet transform uses another technique. Wavelet transform is a bit of a compromise between Fourier transform and spline approximation, but more about this in chapter 2.1.2. What we basically do is try and discern certain elements in the picture, the ones that differ sharply from the surroundings. In the example of ice hockey we have brightly colored players, a white ice and a puck, a small cylindrical object skidding on the ice. With wavelets we can then first take an overview of the white ice, describing it with a few functions spread out over a larger area than a single pixel. The players though, who are a sharp contrast to the ice are covered with other wavelets, and finally the puck, a black small object, are covered by wavelets of the finest degree. Then we take the most noticeable wavelets and just send that information, meaning that the puck, the players and the ice will be kept. Leaving out finer details, like cuts in the ice and similar apperances. The sharp contrasts stay sharp, but the smaller details or smooth surfaces blend out more.

1.1 Compression

When it comes to image compression we basically have two different techniques, lossless and lossy. Lossless compression means that we can guarantee that no information is lost, all the colors appear as they were in the original image, and no data is lost. This can be done thanks to the redundancy in the information stream. The most trivial example is if you have a lot of the same colors following one another, you simply write a number of how many, and then the color. So if we have a picture of for example a red sports car, the surface will have big parts that is the same shade of red. Meaning we can compress this information greatly. Generally we use more complex methods like entropy coding, for example Huffman coding. You try and identify details in the picture that appear over and over, and represent them with a smaller number instead. Lossless coding is mostly used for data, such as written text, where certain words appear multiple times, and you can replace them by a single number or two. Now, lossy coding resides on the idea that a picture does not need to be perfectly recreated. The human visual system can generally cope with quite big perceptual losses, it is used to it every day. Lossy compression is usually achieved by looking at the overall trends of the image, and then just reporting the differences to this trend. A common way of doing this is with some sort of mathematical

transformation, for example Fourier transform, like in the JPEG standard for example. Naturally lossy compression can be virtually lossless, since we can decide the degree of precision at the cost of having to use more data. But generally the benefit of having to use so little data makes the poor quality worthwhile.

1.2 Service transparency

One of todays most common uses of image compression is to send images over the Internet. Since the bandwidth is a very limiting factor to using high resolution video streams, this field has enormous research potential. One of the troubles are that different clients have very different bandwidths. It is almost impossible to compare the connection speed of a modem with a broadband connection. Generally this is solved with the user being able to choose between three different types of resolution, or some other band controlling settings, like quality. Sometimes the server finds out the connection speed automatically and supplies you with the best alternative as far as it can perceive, but some users prefer to be able to decide this on their own. What we have been trying to achieve is something that is truly service transparent, that means that no matter what your connection speed or terminal type is, you will get the best possible video transfer. Of course this is an ambitious goal. Ordinary MPEG video standard has trouble achieving this, since the code contains information such as motion compensation. The code has to be fully embedded, that means that control information has to be transmitted with the image information, and not before, or even worse, after. It is also a good thing if it can be cut off at any time, without loss of necessary information. Other good features is the scalability of the transform. MPEG, which is based on Fourier transform, is interpolating between the pixels. This means that even if we scale it up, it will not become as blocky as some other methods. It is also good to have some way to scale the video down, and still have certain anti alias effects. Other things to take in consideration are memory and processor consumption of some algorithms. Now wavelets have proven to be contained within all those criteria, it has very good compression and interpolating abilities, an in place transformation and relatively low computational complexity. This is what drove us to try out wavelets at this borderline case. Service transparency is just full of compromises, since it has to spread over such a wide base of end users.

2 Theory

2.1 Wavelet Theory

Wavelet transform is an orthogonal transform, like Fourier and many other transforms used in image compression. An orthogonal transform is not only

one to one, but the relation is also very simple, due to the spectral theorem. We will start by going through those concepts with the reader who might need to brush up on them. If you already feel confident in this field, feel free to skip to chapter 2.1.2.

2.1.1 Orthogonality

The concept of orthogonality was first developed for geometry, where you said that two lines crossing each other perpendicularly was orthogonal to each other. This was then transfered to vectors, where orthogonality became a very useful tool, especially when defining coordinates. This concept was adapted by more general spaces as well, consisting of both vectors or functions or anything you can define a scalar product over. To be able to see if we have orthogonality, we need an inner product. The most common inner product used is the $\mathbb{L}^2$ norm, that is

$$\langle f, g \rangle = \int_{-\infty}^{\infty} f(t)g(t)\,\mathrm{d}t.$$

From this we can derive the norm

$$\mathbb{L}^2(f) = \left(\int_{-\infty}^{\infty} f(t)^2\,\mathrm{d}t \right)^{\frac{1}{2}} = \sqrt[2]{\langle f, f \rangle}.^1$$

The concept of orthogonality comes in play when

$$\int_{-\infty}^{\infty} f(t)g(t)\,\mathrm{d}t = \langle f, g \rangle = 0,\ f \neq g.$$

That is when we say that f and g are orthogonal. The concept of orthonormality means that f and g also are scaled so that

$$\sqrt{\int_{-\infty}^{\infty} f(t)^2\,\mathrm{d}t} = \sqrt{\langle f, f \rangle} = F,$$

so if we scale $f(t)$ with $c = \frac{1}{F}$ we get

$$\int_{-\infty}^{\infty} c^2 f^2(t)\,\mathrm{d}t = \langle f, f \rangle = c^2 F^2 = \frac{F^2}{F^2} = 1,$$

[1] Since we assume $\int f^2 \mathrm{d}t < \infty$.

and has thus been rendered orthonormal. The same naturally goes for g. The extra work we spent normalizing the functions is well spent, as we will see later. We can namely described a function space with a few basis functions. For example all polynomials of degree 2 and lower, $\mathbb{P}_2$ on the interval between $[-1, 1]$, can be describe with linear combinations of the functions $\left\{1, x, x^2 - \frac{1}{3}\right\}$. If we take the function

$$f = 7 + 3x(2 + x) \, , x \in [-1, 1]$$

it can be described as

$$f = 8 \cdot 1 + 6 \cdot x + 3(x^2 - \frac{1}{3}) = 8 + 6x + 3x^2 - 1 = 7 + 3x(2 + x) \, , x \in [-1, 1].$$

But if we make it clear that we are working in a function space that consists of $\left\{1, x, x^2 - \frac{1}{3}\right\}$ it is easier to say that

$$f = \{8, 6, 3\}.$$

Now this happen to be an orthogonal basis for $\mathbb{P}_2$, but it is not orthonormal. Finding out the coefficients is easy enough, but let us assume for a moment that we have to do this in the correct manner. We then have something called the spectral theorem (1), which states that when we have an orthonormal basis, the coefficients can be calculated by

$$f = \left\{ \frac{\langle f, 1\rangle}{\langle 1, 1\rangle}, \frac{\langle f, x\rangle}{\langle x, x\rangle}, \frac{\langle f, x^2 - \frac{1}{3}\rangle}{\langle x^2 - \frac{1}{3}, x^2 - \frac{1}{3}\rangle} \right\}. \tag{1}$$

But if we had normalized our functions, as we explained above, equation (1) is reduced to

$$f = \{\langle f, p_0\rangle, \langle f, p_1\rangle, \langle f, p_2\rangle\} \, ,$$

with $\{p_0, p_1, p_2\}$ being the orthonormalization $\left\{ \sqrt[2]{\frac{1}{2}}, \sqrt[2]{\frac{3}{2}}x, \sqrt[2]{\frac{45}{8}}(x^2 - \frac{1}{3}) \right\}$. The interesting thing is that even if we have a function that can not be described as a linear combination of $\{p_0, p_1, p_2\}$, we can use the spectral theorem to get the best possible approximation. If we for example wish to represent $\sin\left(\frac{\pi t}{2}\right)$ in $\mathbb{P}_2$ it would look like

$$\sin(\pi t/2)_{\mathbb{P}_2} = p_0\langle p_0, \sin(\pi t/2)\rangle + p_1\langle p_1, \sin(\pi t/2)\rangle + p_2\langle p_2, \sin(\pi t/2)\rangle =$$

$$\sqrt[2]{1/2} \int_{-1}^{1} \sqrt[2]{1/2} \sin^2(\pi t/2) \, \mathrm{d}t + \sqrt[2]{3/2}x \int_{-1}^{1} \sqrt[2]{3/2}x \sin^2(\pi t/2) \, \mathrm{d}t +$$

$$+ \sqrt[2]{45/8}\left(x^2 - 1/3\right) \int_{-1}^{1} \sqrt[2]{45/8}\left(x^2 - 1/3\right) \sin^2(\pi t/2) \, \mathrm{d}t =$$

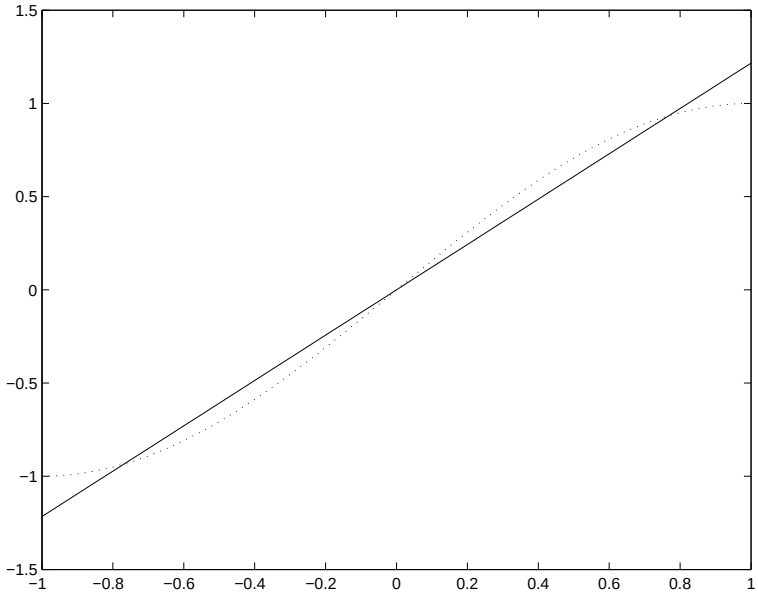


Figure 1: This is a second degree polynomial approximation of a sine function around zero.

$$= 0 \cdot 1 + 12x/\pi^2 + 0 \cdot \left(x^2 - 1/3\right).$$

As we see in figure (1), the approximation is pretty good. Do not let the square roots intimidate you, orthonormality is really useful, especially since we do not have to take $|p_n|$ into consideration. The general Spectral Theorem looks like

Theorem 1 (Spectral) *If f is a function belonging or not belonging to $\mathbb{S}$, where $\mathbb{S}$ consists of the finite of infinite orthonormal basis $\{s_0, s_1, \ldots, s_n\}$, and have a scalar product $\langle \cdot, \cdot \rangle$ and a norm $|\cdot|$. Then f can be represented as*

$$f = s_0\langle f, s_0\rangle + s_1\langle f, s_1\rangle + \ldots + s_n\langle f, s_n\rangle + f_o,$$

where $|f_o|$ is as small as possible.

It is really the spectral theorem that you use in the Fourier transform. Since all the sine and cosine functions are orthogonal to one another, they create an orthonormal basis for at least all continuous functions in $\mathbb{L}^{2\dagger}$ and a very good approximation for discontinuous ones.

2.1.2 Wavelets

A logical question now is, are trigonometric functions the only orthogonal bases in $\mathbb{L}^2$, and the answer is of course no. We can make an orthonormal

$^\dagger\mathbb{L}^2$ is the room of all square integrable, $\int f^2 \mathrm{d}t < \infty$, functions.

base of any functions we like, for example all polynomials $\{1, x, x^2, \dots, x^n\}$. But to describe any continuous function we need to continue $\lim_{n\to\infty}$, and those bases are not orthogonal. Of course they can be orthogonalized to each other with for example Gram Schmidt's method, but that is really not feasible on an infinite amount of polynomials. On the other hand, there are indeed non elementary functions that are orthonormal on $\mathbb{L}^2$, for example Wavelets. But let us start with a little example, the Haar wavelet. It is an easy wavelet to understand and use, and was developed some hundred years ago. A wavelet really consists of two parts, a scaling function, or a father wavelet. In the Haar case it looks like

$$\mathrm{S}_0^0 = \begin{cases} 1 & : & 0 \leq x < 1 \\ 0 & : & x \notin [0,1) \end{cases} .$$

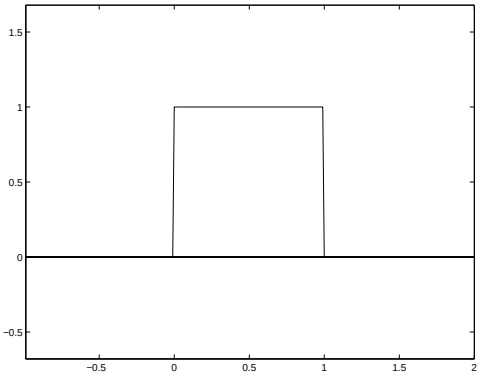


Figure 2: This is the Haar scaling function, as presented mathematically and visually.

This function can be used to approximate a function of zero polynomials that are piecewise one by one of the length one. What the scalar product

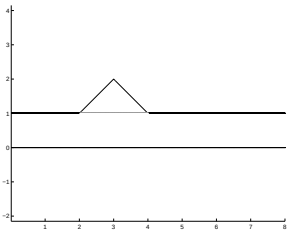
$$\langle f, \mathrm{S}_0^0 \rangle = \mathrm{mean}(f(x)),\ x \in [0,1)$$

really represent is the average value of the function we wish to approximate over the interval $[0,1)$. Now, very few functions just stretch over $[0,1)$, so we can either contract the function down to the desired length, or we can use many different scaling functions, translated to cover our original function with piece vice zero polynomials like Haar. The translations will appear as

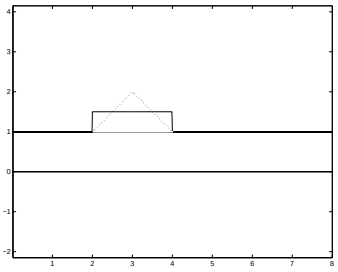
$$\mathrm{S}_0^0(x-n) = \begin{cases} 1 & : & n \leq x < n+1 \\ 0 & : & x \notin [n, n+1) \end{cases} \quad n \in \mathbb{Z}.$$

Now let us approximate the function

$$f(x) = \begin{cases} 1 & : & 0 \leq x < 2 \\ x-1 & : & 2 \leq x < 3 \\ 5-x & : & 3 \leq x < 4 \\ 1 & : & 4 \leq x < 8 \\ 0 & : & x \notin [0,8) \end{cases} .$$



$$
\begin{aligned}
c_1 &= \langle f, \mathrm{S}_0^0 \rangle = 1 \quad : \quad x \in [0,1), \\
c_2 &= \langle f, \mathrm{S}_1^0 \rangle = 1 \quad : \quad x \in [1,2), \\
c_3 &= \langle f, \mathrm{S}_2^0 \rangle = 1.5 \quad : \quad x \in [2,3), \\
c_4 &= \langle f, \mathrm{S}_3^0 \rangle = 1.5 \quad : \quad x \in [3,4), \\
c_5 &= \langle f, \mathrm{S}_4^0 \rangle = 1 \quad : \quad x \in [4,5), \\
c_6 &= \langle f, \mathrm{S}_5^0 \rangle = 1 \quad : \quad x \in [5,6), \\
c_7 &= \langle f, \mathrm{S}_6^0 \rangle = 1 \quad : \quad x \in [6,7), \\
c_8 &= \langle f, \mathrm{S}_7^0 \rangle = 1 \quad : \quad x \in [7,8).
\end{aligned}
$$



We can do this with eight coefficients

The trouble here is that the flat area of $x = [4, 8)$ has to be represented with four coefficients. If we were to describe this function to someone we would not say that "It starts at one, then continues as one, then it goes up to two, then it goes down to one, stays at one, stays at one ... ". No, you would most probably say something like "It is continuous at one, except a bump between two and four.". This is common sense, you state the overall appearance first, and then you start describing the details. Therefore we have the wavelets[2]. The wavelet is slightly different from the scaling function, in the Haar case it appears as

$$
\mathrm{H}_0^0 = \left\{ \begin{array}{rcl} 1 & : & 0 \leq x < \frac{1}{2} \\ -1 & : & \frac{1}{2} \leq x < 1 \\ 0 & : & x \notin [0,1) \end{array} \right\} = \sqrt{2} \left(\frac{\mathrm{S}_0^1}{\sqrt{2}} - \frac{\mathrm{S}_1^1}{\sqrt{2}} \right). \tag{2}
$$

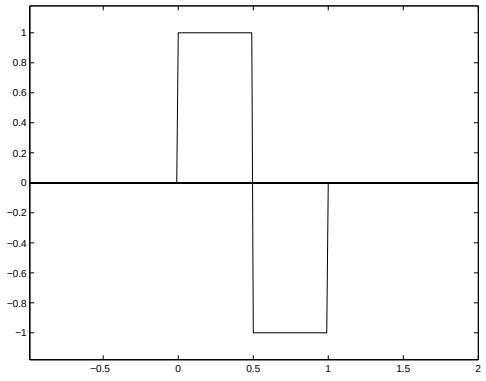


Figure 3: This is the primal Haar wavelet, known as the mother wavelet.

The last part here shows that the wavelet really consists of two instances of the scaling function, slightly altered. The new notation now implies that the scaling functions are not only slided over the $\mathbb{R}$ axis, it is also dilated.

[2]Not to be mixed up with the father wavelet, who was the scaling function. The wavelets are also sometimes referred to as mother wavelets, this comes clear later when we see how all following wavelets really spawn from one original form, the mother wavelet.

Meaning that it is compressed in the direction of the $\mathbb{R}$ axis, but still has to retain it's overall integral of one. This gives us the dilation equation

$$\mathrm{S}_{\mathrm{n}}^{\mathrm{d}} = \mathrm{S}_{2n}^{d+1}(2x - 2n) + \mathrm{S}_{2n-1}^{d+1}(2x - 2n + 1),\ d, n \in \mathbb{Z}, \tag{3}$$

which we can use recursively to increase the resolution of our piecewise constant approximation. The implication of this is that we can write $\mathrm{S}_{\mathrm{n}}^{\mathrm{d}}$ in an arbitrary resolution. Leading to the fact that we can represent any function in $\mathbb{L}^2$ with a maximum error

$$\varepsilon(d) = \int_{\infty}^{-\infty} \left(f - \sum_{n=-\infty}^{\infty} \mathrm{S}_n^d \right)^2 \mathrm{d}t, \tag{4}$$

where $\lim_{d\to\infty} \varepsilon(d) = 0$. But what do we gain from representing a function with wavelets and scaling functions, if the scaling functions alone can represent any $\mathbb{L}^2$ function arbitrary well? If we take the function before, and represent it with a scaling function cowering up half the support of f. That is, using $\mathrm{S}_{\mathrm{n}}^2$ instead of S_n^0 with $n = \{0, 1\}$ we get the coefficients

$$s_1^2 = \langle f, \mathrm{S}_0^2 \rangle = 1.25,\ x \in [0, 4),$$

$$s_2^2 = \langle f, \mathrm{S}_1^2 \rangle = 1,\ x \in [4, 8).$$

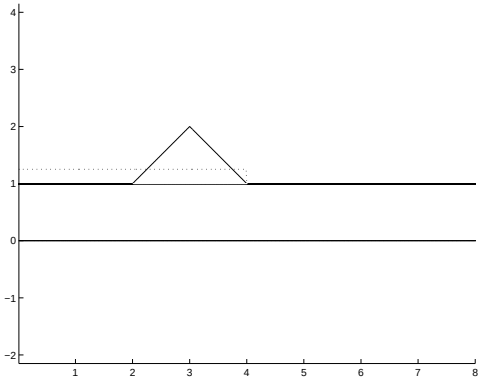


Figure 4: This is a $\mathcal{V}^2$ Haar approximation of our previous function.

Now this is not the same representation as we had above, but if we add

$$d_{1,0}^2 = \langle f, \mathrm{H}_0^2 \rangle = -0.25,\ x \in [0, 4).$$

we have

$$s_1^2\mathrm{S}_0^2 + s_2^2\mathrm{S}_1^2 + d_1^2\mathrm{H}_0^1 =$$

$$= c_1\mathrm{S}_0^0 + c_2\mathrm{S}_1^0 + c_3\mathrm{S}_2^0 + c_4\mathrm{S}_3^0 + c_5\mathrm{S}_4^0 + c_6\mathrm{S}_5^0 + c_7\mathrm{S}_6^0 + c_8\mathrm{S}_7^0.$$

Here we can see that by cleverly choosing the right scaling function, and using wavelets to increase the resolution, we will get by on a lot fewer coefficients. Naturally we have to express it with all scalar products, exactly as

in the spectral theorem (see page 8), but many more of the coefficients will be zero. We also see that if we would have chosen to represent the function with S_0^3 we would have ended up with the same amount of coefficients, but this is not always the case. Sometimes it is desirable to not start from the lowest resolution. But more about that later.

2.1.3 Multiresolution analysis

This is the theory behind what we have just witnessed. To make all this work we have certain criteria that the Wavelets have to fulfill. The most important one is firstly that all scaling functions ϕ_n^d of a certain resolution d has to be orthogonal to one another. Moreover, all following wavelet functions ψ_n^p of resolution $p \geq d$ has to be orthogonal, both to themselves as well as all scaling functions of resolution d. Mathematically this can be expressed as

$$\begin{aligned}
\langle \phi_n^d, \phi_m^d \rangle &= \begin{cases} 1 & : \quad n = m \\ 0 & : \quad \text{otherwise} \end{cases}, \\
\langle \psi_n^p, \psi_n^h \rangle &= \begin{cases} 1 & : \quad p = h,\ n = m,\ p, h \geq d \\ 0 & : \quad p, h \geq d,\ \text{otherwise} \end{cases}, \\
\langle \psi_n^p, \phi_m^d \rangle &= \quad 0 \quad : \quad p, h \geq d\ .
\end{aligned}$$

Now you can see that the notation has changed, it is no longer S_n^d but ϕ_n^d for the scaling function, and instead of H_n^d we write ψ_n^d. This change of notation is made to make sure we are now talking about general scaling functions and wavelets, not the Haar case explicitly. As we observed before, the Haar wavelet was composed by two scaling functions of a higher resolution. This is how all wavelets are made, the wavelet consists of a linear combinations of scaling functions from $\text{Span}(\phi_n^{d+1})$, in a fashion to make the orthogonality criteria above hold true. The same goes for the wavelet. Generally the scaling function of a higher resolution is written, as in the Haar case in equation (3) as

$$\phi_i^d = \sum_n g(n) \sqrt{2} \phi_{2i+n}^{d+1} (2x)\,.$$

In the Haar function $g = \left\{ \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \right\}$, giving us equation(3). One of the most important requirement of $g(n)$ is that it is normalized, as

$$\sum_n |g(n)|^2 = 1,$$

but many other apply. We do not plan to take it up here though, but it is very explicitly written in [Dau92]. In the same way as ϕ_i^d is constructed out of ϕ_{2i+n}^{d+1} we can construct

$$\psi_i^d = \sum_n h(n) \sqrt{2} \phi_{2i+n}^{d+1} (2x)\,.$$

Here $h(n)$ is actually $h(n) = (-1)^n g(-n+1)$. For example in equation (2) $h(n) = \left\{\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\right\}$. Here we see again that higher resolution wavelets and scaling functions, have twice the resolutions of old scaling functions. If we take $\text{Span}(\phi_n^d) = \mathcal{V}^d$ we can immediately see that $\mathcal{V}^d \in \text{Span}(\phi_n^{d+1}) = \mathcal{V}^{d+1}$. In a similar manner we call the function space $\text{Span}(\psi_n^d) = \mathcal{W}^d$, such that we can write

$$\mathcal{V}^{d+1} = \mathcal{V}^d \oplus \mathcal{W}^d. \tag{5}$$

Now let us leave this for a little moment, and instead consider how we can achieve arbitrary good approximations with wavelets, as in equation (4). We will not try and present any proof of this, rather some arguments to clarify that such is the case. If we take the scaling function, and dilate it far enough,

$$\lim_{d\to\infty} \phi_n^d = \lim_{d\to\infty} \sqrt{2}^d \phi\left(2^d\left(x - \frac{n}{2^d}\right)\right) = \lim_{d\to\infty} \delta\left(x - \frac{n}{2^d}\right) =$$

$$= \left\{\lim_{d\to\infty} \frac{n}{2^d} \to t,\ n \in \mathbb{Z},\ t \in \mathbb{R}\right\} = \delta(x-t), \quad x, t \in \mathbb{R}$$

we see that we end up with $\delta(x-t)$. This does not really tell us anything more than that the resolution can be as fine as we like. The implications of this is though that

$$\overline{\text{Span}\left(\mathcal{V}^\infty\right)} \in \mathbb{L}^2,$$

where $\overline{\text{Span}(f)}$ means that we take the closure of $\text{Span}(f)$. That is, we continue every converging series of elements in the room, or in short, we take the elements and their closest neighbors. Now this means that we can represent $\mathbb{L}^2$ with scaling functions. But the thing that really adds the edge to this transform are the wavelets. If you remember from equation (5), we could write

$$\mathcal{V}^{d+1} = \mathcal{V}^d \oplus \mathcal{W}^d$$

and naturally we can show by induction that

$$\mathcal{V}^p = \mathcal{V}^d \oplus \mathcal{W}^d \oplus \ldots \oplus \mathcal{W}^{p-1},\ p \geq d$$

also hold true. Letting $\lim_{p\to\infty}$ we then arrive at

$$\mathbb{L}^2 = \mathcal{V}^\infty = \mathcal{V}^d \oplus \mathcal{W}^d \oplus \ldots \oplus \mathcal{W}^\infty$$

and thus by adding the wavelet spaces of higher resolution, we arrive at $\mathbb{L}^2$ and our goal of arbitrary closeness[3].

[3] See equation (4).

2.2 Filter Banks

All this theory is useful for continuous functions. But in computers we never have continuous signals, everything is discrete, both in amplitude and over time, $f(x)$ $f, x \in \mathbb{Z}$. Now most wavelets and scaling functions have no analytical representation, so we could probably use the numerical approximations to do integrals over and over again. But then you need new functions for each dilation, making a lot of calculations, and naturally the computers will sneak a lot of errors into the calculations. But there is another way, called filter banks. This is a well known concept in signal analysis, and the theory can be directly used in wavelet transform over discrete functions. The first chapter will clarify some details in the Z-transform, and is just there for you who are not really fluent it it. If you already know what Z-transform is you can safely skip that chapter and go directly to section 2.3.

2.2.1 The Z-transform

The Z-transform takes a sequence $s(n)$ of numbers and transforms it into a polynomial of z, like

$$\mathcal{Z}(s(n)) = \sum_{n=-\infty}^{\infty} s(n)z^{-1} = S(n).$$

This is mainly used in convolutions where

$$\mathcal{Z}(s * t) = \mathcal{Z}\left(\sum_{k=-\infty}^{\infty} s(n-k)t(k)\right) = \mathcal{Z}(s(n))\mathcal{Z}(t(n)) = S(z)T(z).$$

This is an operation that means that every point in $s(n)$ is multiplied with the points in $t(k)$, spread out over $n-k$ and then added together to form a new sequence $p = s * t$. For example, let us consider the sequences of

$$s(n) = \{1, 2, 3, 4, 3, 2, 1\},\ t(n) = \{2, 3, 1\}.$$

Here we already see one of the drawbacks of not using Z-transform. There is no way we can discern if t's zero element is the number 2, 3 or maybe even the number 1, and this effects the later convolution. Let us assume the zeroth element is the middle of t and do a proper convolution, this would

look like

s		1	2	3	4	3	2	1	
$s(1)\cdot t$	2	3	1						
$s(2)\cdot t$		4	6	2					
$s(3)\cdot t$			6	9	3				
$s(4)\cdot t$				8	12	4			
$s(5)\cdot t$					6	9	3		
$s(6)\cdot t$						4	6	2	
$s(7)\cdot t$							2	3	1
Σ	2	7	13	19	21	17	11	5	1

.

This is then equal to the simple polynomial of

$$S(z) = 1 + 2z^{-1} + 3z^{-2} + 4z^{-3} + 3z^{-4} + 2z^{-5} + 1z^{-6},\ T(z) = 2z + 3 + z^{-1},$$

being multiplied as

$$S(z)T(z) = (1 + 2z^{-1} + 3z^{-2} + 4z^{-3} + 3z^{-4} + 2z^{-5} + 1z^{-6})(2z + 3 + z^{-1}) =$$

$$= 2z + 4 + 6z^{-1} + 8z^{-2} + 6z^{-3} + 4z^{-4} + 2z^{-5}3 + 6z^{-1} + 9z^{-2} + 12z^{-3} +$$

$$+ 9z^{-4} + 6z^{-5} + 3z^{-6} + z^{-1} + 2z^{-2} + 3z^{-3} + 4z^{-4} + 3z^{-5} + 2z^{-6} + z^{-7} =$$

$$= 2z + 7 + 13z^{-1} + 19z^{-2} + +21z^{-3} + 17z^{-4} + 15z^{-5} + z^{-6} + z^{-7} = \mathcal{Z}(s * z).$$

The calculations may look more cumbersome, but the notation is really easier to handle, especially in algebraic equations.

2.3 From wavelets to filters

To go from ordinary continuous functions to filter banks we need a little theory, explaining why this actually works, and is a natural ability within the wavelet transform. If we look at the basic dilation equation

$$\phi(x) = \sum_{n=-\infty}^{\infty} g(n)\sqrt{2}\phi(2x - n),$$

which we learned in the former chapter. The g coefficients represent the construction vector for the scaling function $\phi \in \mathcal{V}^n$ from the scaling functions of finer resolution $\phi(2x) \in \mathcal{V}^{n+1}$. Now let us unleash some math over all this, and see what we can get out of it. We start with substituting the time variable $x = 2^d t - k$ like in the dilation equation, and get

$$\phi(2^d t - k) = \sum_{n=-\infty}^{\infty} g(n)\sqrt{2}\phi(2(2^d t - k) - n) = \sum_{n=-\infty}^{\infty} g(n)\sqrt{2}\phi(2^{d+1}t - 2k - n).$$

Now let us substitute $2k+n$ with m to make everything more readable, giving us

$$\phi(2^d t - k) = \sum_{m=-\infty}^{\infty} g(m-2k)\sqrt{2}\phi(2^{d+1}t - m), \tag{6}$$

and in the wavelet case

$$\psi(2^d t - k) = \sum_{m=-\infty}^{\infty} h(m-2k)\sqrt{2}\phi(2^{d+1}t - m). \tag{7}$$

If we now take a function $f^{d+1}(t) \in \mathcal{V}^{d+1}$ it will be represented as

$$f^{d+1} = \sum_{m=-\infty}^{\infty} s^{d+1}(m) 2^{\frac{d+1}{2}} \phi(2^{d+1}t - m),$$

or with s^d and d^d from a lower resolution j as

$$f^{d+1} = \sum_{m=-\infty}^{\infty} s^d(m) 2^{\frac{d}{2}} \phi(2^d t - m) + \sum_{m=-\infty}^{\infty} d^d(m) 2^{\frac{d}{2}} \psi(2^d t - m).$$

By the spectral theorem, we can express

$$s^d(k) = \langle f(t), \phi^d(t) \rangle = \int f(t) 2^{\frac{d}{2}} \phi(2^d t - k) dt \tag{8}$$

and

$$d^d(k) = \langle f(t), \psi^d(t) \rangle = \int f(t) 2^{\frac{d}{2}} \psi(2^d t - k) dt. \tag{9}$$

Then, by inserting equations (6) and (7) into equations (8) and (9) and exchanging the order of integration and summation, we get

$$s^d(k) = \sum_m g(m-2k) \int f(t) 2^{\frac{d+1}{2}} \phi(2^{d+1}t - m) \mathrm{d}t \tag{10}$$

and

$$d^d(k) = \sum_m h(m-2k) \int f(t) 2^{\frac{d+1}{2}} \phi(2^{d+1}t - m) \mathrm{d}t. \tag{11}$$

Taking a closer look at the integral, and equation (10) we see that the integral really is $\langle f(t), \phi^{d+1,n}(t) \rangle$, that is, $s^{d+1}(n)$. This gives us the important equations

$$s^d(k) = \sum_m g(m-2k) s^{d+1}(m),$$

$$d^d(k) = \sum_m h(m-2k)s^{d+1}(m).$$

This is then equivalent with us taking every second element in the convolution $g(-n)*s^{d+1}(n)$. This is known as downsampling, $s(n) = p(2n) = \downarrow^2(p(m))$. We downsample $h(-n)$ and $s^{d+1}(n)$ in the same manner. Here we see how we can calculate the lower level coefficients from the higher level scaling coefficients. So all we need is the highest level coefficients, and how do we get hold of them? The truth is, we already have them. Remember from chapter 5 that as we dilate the scaling function, we eventually reach a Dirac function. So a discrete time signal is really the highest level scaling coefficients, and all we need to do to get a lower level set of coefficients, useful for our transforms, is running it through the two convolutions above. In a similar manner we can calculate the coefficients of the highest level, that is the original function, with the lower level scaling and wavelet coefficients. If we express our function as

$$f^{d+1}(t) = \sum_n s^{d+1}(n)2^{\frac{d+1}{2}}\phi(2^{d+1}t-n) =$$

$$\sum_n s^d 2^{\frac{d}{2}}\phi(2^d t-n) + \sum_n d^d 2^{\frac{d}{2}}\psi(2^d t-n).$$

If we then substitute equations (6) and (7) into this we get

$$f^{d+1}(t) = \sum_n \sum_m s^d 2^{\frac{d+1}{2}} g(m-2n)\phi(2^{d+1}t-m) +$$

$$+ \sum_n \sum_m d^d 2^{\frac{d+1}{2}} h(m-2n)\phi(2^{d+1}t-m).$$

Which can be written as

$$f^{d+1}(t) = \sum_n s^{d+1}2^{\frac{d+1}{2}}\phi(2^{d+1}t-n) =$$

$$= \sum_n \sum_m (g(m-2n)s^d + h(m-2n)d^d)2^{\frac{d+1}{2}}\phi(2^{d+1}t-n).$$

The only way this can be correct is if

$$s^{d+1}(n) = \sum_m (g(m-2n)s^d(n) + h(m-2n)d^d(n)),$$

that is, the $s(n)$ of a higher resolution equals the sum of both series s and d, first upsampled by 2 and then convolved with h and g respectively. Upsampling is the opposite of downsampling, where we put in a zero between every element in the series, as $p(m) = \uparrow_2(s(n))$ means that $p(2n) = s(n)$ and $p(2n+1) = 0$. This is exactly as in equation (14). The series of g and h

will also hereby be known as lowpass and highpass filters. Since they are constructed in a way to only keep signals of low respectively high frequencies.

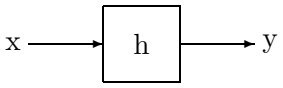
$$\boxed{s^d(n) = \downarrow^2(g^* * s^{d+1})} \tag{12}$$

$$\boxed{d^d(n) = \downarrow^2(h^* * s^{d+1})} \tag{13}$$

$$\boxed{s^{d+1}(n) = g * \uparrow_2(s^d) + h * \uparrow_2(d^d)} \tag{14}$$

2.3.1 Signal analysis and filters

Now we can use ordinary signal analysis to express all of this in a very elegant manner. The convolutions can first and foremost be expressed as filter equations, or more commonly, just filters. A filter is an operator working on a signal, transforming a one or multi dimensional input signal, here known as $x(t)$ into an output signal $y(t)$.



A filter h is commonly expressed by it's impulse response[4], this uniquely defines our filter, and our filter coefficients. For example, a Haar highpass filter gives us the impulse response of $\left\{\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\right\}$. This can also be expressed as, if

$$x_0 = 1,\ x_n = 0\ \{n|\, n \neq 0,\ n \in \mathbb{Z}\},$$

then

$$y_n = \sum_{k=0}^{1} h(k)x(n-k)$$

which leads to

$$y_0 = \frac{1}{\sqrt{2}},\ y_1 = -\frac{1}{\sqrt{2}},$$

giving us

$$h(z) = \frac{1}{\sqrt{2}} - \frac{z^{-1}}{\sqrt{2}}.$$

[4]What comes out when we put in a single one on position zero,

Before we dig ourselves down in details about how this can be used for our discrete time wavelet transform, We think we should go though some theory and examples about filters. If the reader already feel confident in this she can skip forward to the next chapter. For the theory above to work we need to have causal filters, that is

$$h_n = 0, \ \forall n < 0.$$

But this is no real problem, because all finite response filters can be made causal with just a long enough time shift. A finite response filter is a filter with only a finite number of filter coefficients set non zero, meaning that our impulse response have a start and a finish. This is the filter equivalent to our compactly supported wavelets. You will sometimes see that this part is sloppily skipped in some literature, but it works the same way except for the Z-transform, where you have to add a time shift before and after, to make your filter and function both causal. For example

$$\begin{aligned} h(z) &= z^2 - z^1 + 1 - z^{-1} + z^{-2}, \\ z^{-2}h(z) &= 1 - z^{-1} + z^{-2} - z^{-3} + z^{-4}, \end{aligned}$$

assures that h will be causal. We then of course have to shift our answer back down two steps to not end up with a delayed signal. This step is generally skipped though. One last thing that has to be done with filters are time reversal, meaning that we switch the order of the filter coefficients, like

$$h_n^* = h_{-n},$$

or in Z-transform

$$h^*(z) = h(z^{-1}).$$

Here we assign the time reversed filters with h^*. Our highpass filter from before would for example time reverse into

$$h_{-1}^* = \frac{-1}{\sqrt{2}}, \ h_0^* = \frac{1}{\sqrt{2}}.$$

Now we have the theoretical knowledge of filters that we need to explain how they can be used in filter banks, to start explaining how they can be used for discrete time wavelet transform.

2.3.2 Using filter banks

As we saw before in equations (12) and (13) we showed how to calculate the wavelet and scaling coefficients recursively from the scaling coefficients of a higher level. As well as the fact that the values of the function, from start, can be considered as wavelet coefficients of the highest order, namely s^{max}. The wavelet coefficients of the highest level could the be calculated by

a convolution with the scaling coefficients and the coefficients of $h^*(n)$, then a downsample[5]. An important note is that downsampling destroys information, we only keep half of it. But since we do this on two different sequences, we first double the amount of information before the downsample, and then go back to scratch. In the same way we could calculate the coefficients of s^{max-1} by a convolution with $g^*(n)$ and sampling down[5]. This is much easier to express with filter banks, a more visual graph of the operations.

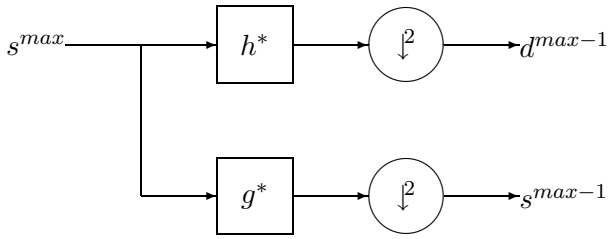


Figure 5: The first instance of a transforming filter bank.

This is a so called filter bank, used to split a signal up in lowpass $s^{max-1}(n)$ and highpass $d^{max-1}(n)$ coefficients. Splitting the signal up in two gives us twice the amount of samples as before, but the downsampling takes away half of those, giving us roughly the same amount as we started with. Generally a little more, since the filters give certain fringe effects, but more about taking care of those later. The most important details are the lowpass coefficients s. They, as stated before, explains the generic behavior of our function, whereas the scaling coefficients d give us the differences from the generic behavior. The energy of the function is also kept in the scaling coefficients, which means they act just like a function on their own. This function can then be divided up further into another filter bank, halving the amount of coefficients left with every iteration. Naturally this works best if we have a length that is a power of two, $|f| = 2^n, n \in \mathbb{Z}$, then the initiating function can be divided up without having to consider serious fringe effects. The out data s^1 and d^1 is generally stored after each other in a new function $f^1 = \{s^1, d^1\}$, the next iterations s^2 and d^2 are stored in $f^2 = \{s^2, d^2, d^1\}$ and so on. Finally giving us $f^n = \{s^n, d^n, d^{n-1}, \dots, d^2, d^1\}$. Of course it would be very nice if this new function could just overwrite our old one, being of equal length. But the filters will add non zero elements before and after the original function, due to the nature of convolutions.

2.3.3 Fringe effects

Now those effects can be dealt with, as well as if we wish do divide a function that is not of even length $|f| = 2^n, n \in \mathbb{Z}$. But let us start with an example

[5]Remember, $s(n) = p(2n) = \downarrow^2(p(m))$.

of all of the above. Let us take a simple function and run it through a Haar filter,

$$x(z) = 1 + 2z^{-1} + z^{-2} + 3z^{-3} = \{1, 2, 1, 3\},$$
$$g^* = \frac{z}{\sqrt{2}} + \frac{1}{\sqrt{2}},\ h^* = \frac{-z}{\sqrt{2}} + \frac{1}{\sqrt{2}}.$$

First we convolve x with g

x		, 1, 2, 1, 3,
$g^* * y_t$		$\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}},$, , ,
$g^* * y_{t+1}$		, $\frac{2}{\sqrt{2}}, \frac{2}{\sqrt{2}},$, ,
$g^* * y_{t+2}$		, , $\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}},$,
$g^* * y_{t+3}$	+	, , , $\frac{3}{\sqrt{2}}, \frac{3}{\sqrt{2}},$
y^g		$\frac{1}{\sqrt{2}}, \frac{3}{\sqrt{2}}, \frac{3}{\sqrt{2}}, \frac{4}{\sqrt{2}}, \frac{3}{\sqrt{2}},$

and then x with h

x		, 1, 2, 1, 3,
$h^* * y_t$		$\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}},$, , ,
$h^* * y_{t+1}$		, $\frac{-2}{\sqrt{2}}, \frac{2}{\sqrt{2}},$, ,
$h^* * y_{t+2}$		, , $\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}},$,
$h^* * y_{t+3}$	+	, , , $\frac{-3}{\sqrt{2}}, \frac{3}{\sqrt{2}},$
y^h		$\frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{-2}{\sqrt{2}}, \frac{3}{\sqrt{2}},$

Downsampling these results then gives us

$$y^{g_1} = \downarrow^2(y^g) = \left\{\frac{1}{\sqrt{2}}, \frac{3}{\sqrt{2}}, \frac{3}{\sqrt{2}}\right\},\ y^{h_1} = \downarrow^2(y^h) = \left\{\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{3}{\sqrt{2}}\right\}.$$

The trouble is that in this form we can not do the downsample before we do the convolution, but we will see later that there are methods to avoid this. But we see here that the length of $|y^{g_1}| = 3$ and $|y^{h_1}| = 3$, meaning that have added information. One way this can be avoided is to continue the function evenly, that is, reversing it and adding it to the edges, and if necessary, reversing it again and adding again, giving us

$$x = \{\dots, 2, 1, 1, 2, 1, 3, 3, 2, \dots\}.$$

This would give us the slightly longer

x		$\dots$, 1, 1, 2, 1, 3, 3,$\dots$
$h^* * y_t$		, $\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}},$, , , ,
$h^* * y_{t+1}$		, , $\frac{-2}{\sqrt{2}}, \frac{2}{\sqrt{2}},$, ,
$h^* * y_{t+2}$		, , , $\frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}},$,
$h^* * y_{t+3}$		, , , , $\frac{-3}{\sqrt{2}}, \frac{3}{\sqrt{2}},$
$h^* * y_{t+4}$	+	, , , , , $\frac{-3}{\sqrt{2}}, \frac{3}{\sqrt{2}},$
y^{h_1}		, $\frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, \frac{-2}{\sqrt{2}}, \frac{0}{\sqrt{2}}, \frac{3}{\sqrt{2}},$

where we simply trash the fringe coefficients at positions 0 and 5, then we downsample. Naturally we do it in the same manner with

$$y^{g_1} = \left\{ \frac{\cancel{1}}{\cancel{\sqrt{2}}}, \frac{3}{\sqrt{2}}, \frac{\cancel{3}}{\cancel{\sqrt{2}}}, \frac{4}{\sqrt{2}}, \frac{\cancel{6}}{\cancel{\sqrt{2}}}, \frac{\cancel{3}}{\cancel{\sqrt{2}}} \right\},$$

where the downsampled values are simply crossed over.

The reason we choose an even wrap around instead of an odd is that a odd wrap around would mean that high values at the end of a function, would also affect the values at the beginning. We are trying to minimize the amount of differences in our function, and it is at it's lowest if the function continues in the same manner.

We can now make a second iteration of our y^g function, giving us

$$y^{g_2} = \frac{7}{2},\ y^{h_2} = \frac{1}{2}.$$

Which can be described with filters like.

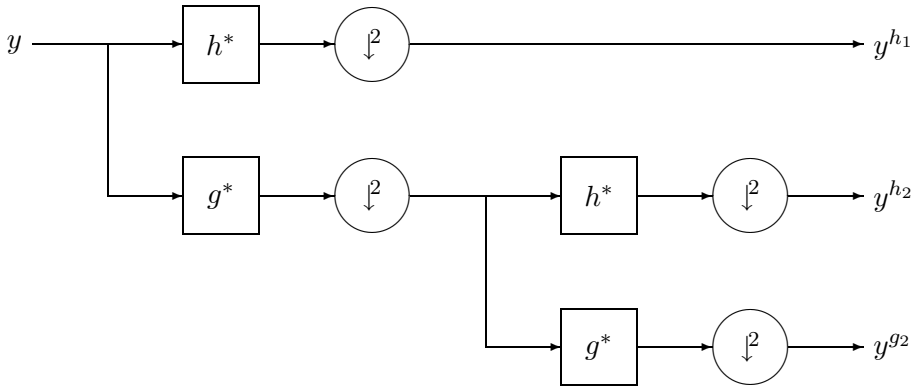


Figure 6: This is a graphical representation of the two first iterative steps in a filterbank, it can be added with lots more.

2.3.4 Inverse filter

Let us consider equation (14), we learned that it was equivalent with first an upsample[6], of both the s and d coefficients. Then we convolve them with the filters g and h respectively, note that those filters are not time reversed, like the first ones were. It is not important whether you time reverse the transformation or inverse transformation[7] filters, just as long as one set is

[6]The opposite of downsampling, where we put in a zero between every element in the series, as $p(m) = \uparrow_2(s(n))$ means that $p(2n) = s(n)$ and $p(2n+1) = 0$. This operation does not destroy any information, but does not add anything new either.

[7]Commonly known as synthesis.

and the other is not. Let us look at our example, with the synthesis filters

$$g = \frac{1}{\sqrt{2}} + \frac{z^{-1}}{\sqrt{2}},\ h = \frac{1}{\sqrt{2}} - \frac{z^{-1}}{\sqrt{2}}.$$

We start by operating on the sequence of $\{\uparrow_2 y^{g_2}\}$ and $\{\uparrow_2 y^{h_2}\}$,

$\uparrow_2 y^{g_2}$	$\frac{7}{2}, \quad 0$
$g * y_t$	$\frac{7}{2\sqrt{2}}, \frac{7}{2\sqrt{2}}$
$g * y_{t+1}$	$0, \quad 0$
y^g	$\frac{7}{2\sqrt{2}}, \frac{7}{2\sqrt{2}}$

,

$\uparrow_2 y^{h_2}$	$\frac{-1}{2}, \quad 0$
$h * y_t$	$\frac{-1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}$
$h * y_{t+1}$	$0, \quad 0$
y^h	$\frac{-1}{2\sqrt{2}}, \frac{1}{2\sqrt{2}}$

.

Now as we add $y^h + y^g = \left\{\frac{7-1}{2\sqrt{2}}, \frac{7+1}{2\sqrt{2}}\right\} = \left\{\frac{3}{\sqrt{2}}, \frac{4}{\sqrt{2}}\right\} = y^{g_1}$ we end up precisely with what we started. Then using this and y^{h_1}we can of course reach our goal of $y = \{1, 2, 1, 3\}$. Meaning we have achieved perfect reconstruction. Graphically representated as a filter bank this would be.

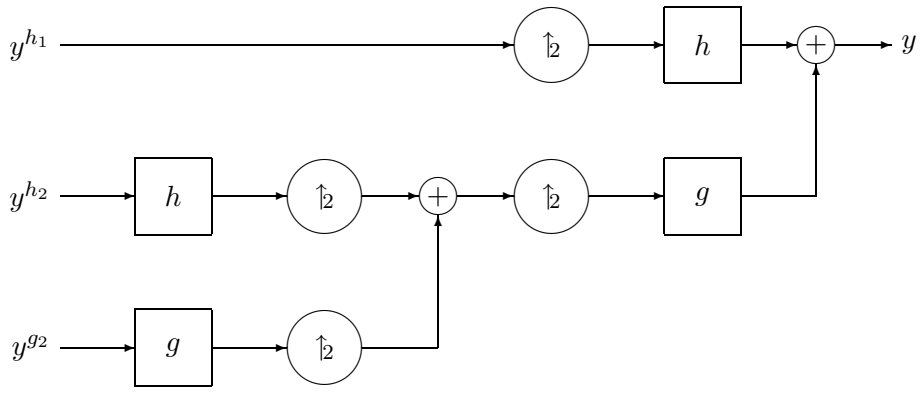


Figure 7: A graphical representation of an inverse, or synthesis filterbank.

Naturally we have to do the same even continuation of our function, this before we downsample, to get the zeroes at the proper places. The benefits are only obvious if you have longer and most importantly, more consisted sequences. Then we will get a lot more zeroes, which we then have to develop a smart method of not having to keep.

2.3.5 Integer filters

Now, since we are operating in a discrete space, we do not wish to have to operate with real numbers. Therefore the Haar filter of $h = \left\{\frac{1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}\right\}$ is very impractical. But there is a way to surpass this. If we put in an operation of

$\sqrt{2}$, as well as of course it's inverse $\frac{1}{\sqrt{2}}$, this is just a multiplication with a constant, and is linear to both convolution and the up and down samples. Meaning that we can put it in direct correlation to our convolutions. Leading to the new filters

$$g^* = 1 + z^{-1},\ h^* = 1 - z^{-1},\ g = \frac{1}{2} + \frac{z^{-1}}{2},\ h = \frac{1}{2} - \frac{z^{-1}}{2},$$

giving us the same net result. As can be seen, those are not really orthogonal counterparts, but rather referred to as biorthogonal. The transform and synthesis filters are not the same, only time reversed, but different sets of filters. But real orthogonality is not really required in wavelet transform, which is a definite feature, giving us the flexibility and freedom to, like in this case being able to just use $\mathbb{Q}$. Furthermore with some clever tricks like fixed point iteration, this can be solved. But not all wavelets are this nice, so it can be very well spent energy to try and find one with coefficients in the range of $\frac{n}{2^p},\ n, p \in \mathbb{Z}$ for computational purposes.

2.3.6 Reflexions about the Haar transform

The Haar transform is really quite worthless, except as a practical example of what wavelet transform really is all about. The lowpass filter of the Haar really just gives us the average value of the function, multiplied by $\sqrt{2}$, and the highpass filter gives us the differences of the two input values to the mean value, multiplied by $\sqrt{2}$ as well[8]. So if the function is very flat, it will lead to a lot of zeroes in the d_n^d coefficients. What we really are doing is moving all zero degree polynomials to s_n^d. Normally, more interesting wavelets have the ability to take care of even higher polynomials, this ability is measured as the amount of vanishing moments.

2.4 Lifting

Now if we take a closer look at the filter banks, we see that on the synthesis side we first make a convolution, then we downsample. This means that we are actually just throwing away about half of our work. Same thing on the other side, we add a lot of zeroes, then we multiply with them, can not we avoid doing this in some way, and there is. This method is called lifting, and the process is very well described in [DS96]. The first idea is that we remake the filters to be able to have the sampling before the filtering, efficiently halving the number of iterations. Now the second idea is to, instead of using convolutions, using elements from the two streams s and d and adding them to each other. This actually works, and the interested reader can read the proofs of this in [DS96].

[8] If we use the discrete filters from above it is of course multiplied by 2.

The basic concept is that we first divide the function into odd and even instances. This is easily done in a computer, but mathematically described as a downsample for the coefficients that are to be s calling it s^0, and a shift with z^{-1} followed by a downsample. This creates the coefficients that are to become d now known as d^0, by taking every odd value in our function f. Now we convolve s^0 with our first filter p_1 and subtract this result from d^0, creating d^1. Then we convolve d^1 with u_1 and add this to s^0 naturally creating s^1. Then we operate with our next lifting pair on those, that is p_2 and u_2 in the following manner

$$\begin{aligned}
d^0 &= z^{-1} \downarrow^2(f(z)) \\
s^0 &= \downarrow^2(f(z)) \\
d^1 &= d^0 - p_1 * s^0 \\
s^1 &= s^0 + u_1 * d^1 \\
&\vdots \\
d^n &= d^{n-1} - p_n * s^{n-1} \\
s^n &= s^{n-1} + u_n * d^n \\
d &= d^n \\
s &= s^n.
\end{aligned}$$

This concludes one filter step. Of course it might look like more cumbersome calculations, but the u_n and p_n filters are generally quite small, in comparison with their corresponding lowpass and highpass filters. Naturally this is just the first step of decreasing the resolution, and we loop s back into the lifting, doing it over and over till we have reached our desired resolution. Graphically this can be represented as follows.

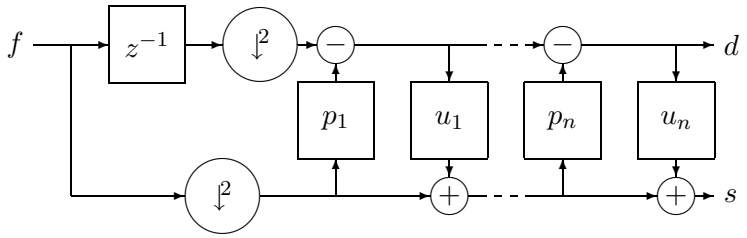


Figure 8: This is a simple sketch of the mechanics of the lifting transform.

One of the features is that this can be done in place. Since we are just subtracting s^i from d^i, we can do this with only one element s at a time, multiplying it with our filter p_i and adding it to the correct elements in d_i. We also do not need to separate our elements in function, we can simply write them on the same places as we took them. Mathematically this is nothing, but with computers it is a huge benefit. We do not have to allocate

extra memory, and the processes can be speeded up greatly. Since we only take elements from s and only subtract them from elements in d, since we are not subtracting from any of the s's we are working with, and not doing any multiplications with elements of d. Of course, in the next step the same holds true, but with the reversed roles of s and d. Generally you only have maybe one, or perhaps two lifting pairs.

If we write back to the sequence we feed our transform, where does it end up. Every even sample is, after writeback s. This means that for every following iteration of wavelet coding, we use every second sample. Because we want our scaling function to cover as much of the original function as possible. With writeback we will then end up with the coefficients widely spaced, since we only use every second element, then evey fourth, eighth and so on. Instead of the pattern with filterbanks, where they are placed after each other. This looks like

$$\begin{aligned} w_{lifting} &= \{s_1^n, d_1^0, d_1^1, d_2^0, d_1^2, d_3^0, \ldots, d_{n/4}^0, d_1^n, d_{n/4+1}^0, \ldots, d_{n/2}^1, d_{n/2}^0\}, \\ w_{filterbank} &= \{s_1^n, d_1^n, \{d_{1\ldots 2}^{n-1}\}, \ldots, \{d_{1\ldots n/2}^0\}\}. \end{aligned} \tag{15}$$

Here n is the amount of elements in our initial function f. As you see, none of the methods create any overlaps. But lifting contains convolutions, and should create fringe effects as well. That is correct, it does, and we treat them exactly like filters, just wrapping the function onto itself at the edges. Of course you can put the values back in any order you like, but we loose the fantastic ability of the writeback then, it also would require more memory shuffling, which increases the computer power required. Actually, it has been shown that filter banks have a computional complexity of $\mathcal{O}(n)$, in comparison with fast wavelet transform with a complexity of $\mathcal{O}(n \log(n))$. If we then consider that lifting uses a little more than half the amount of multiplications we have in filter banks, it is quite fast. Now of course, the fast fourier transform has a very optimized code in comparison with the code that exist for wavelet transform. But on the paper it looks promising.

2.4.1 Inverse lifting

The inverse lifting is even more intuitive than the inverse filter transform. What we just do is the same operations, in reversed order and with reversed sign, exactly like you always do. This looks like figure (9).

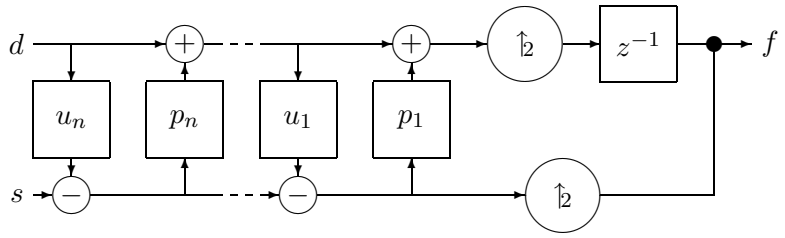


Figure 9: This is a model of the inverse lifting scheme, also known as synthesis.

As above it can be expressed as the equations

$$\begin{aligned} s^n &= s \\ d^n &= d \\ s^{n-1} &= s^n - u_n * d^n \\ d^{n-1} &= d^n + p_n * s^{n-1} \\ &\vdots \\ s^0 &= s^1 - u_1 * d^1 \\ d^0 &= d^1 + p_1 * s^0 \\ f &= \uparrow_2(s^0) + z \uparrow_2(d^0). \end{aligned} \tag{16}$$

As you can see we can even use the same filters here. Of course we loose the freedom we had with biorthogonality earlier, but since two dimensional filters have so much freedom in their designs, we can generally find nice orthogonal filters.

2.4.2 Integer lifting

Integers is always a trouble when it comes to using mathematical theories in a computer. Since the nature of a computer makes it much easier to store integers rather than real numbers, we generally need some sort of transform from $\mathbb{R}$ to $\mathbb{Z}$, generally this is just done via truncation. That is we round the real number towards zero. This should complicate matters, if it was not for the fact that lifting has another advantage, it is not affected by roundoff errors. This is because we add and subtract exactly the same function. if we look at the last instance of our lifting, namely using u_n on our s and d streams. If we recall that

$$\begin{aligned} d &= d^n \\ s &= s^n + u_n * d \\ s^n &= s - u_n * d = s^n + u_n * d - u_n * d = s^n, \end{aligned}$$

then it is easy to see that even if we add the roundoff error of $u_\varepsilon(x)$ whenever we use $u_n(x)$ we see that

$$\begin{aligned} d &= d^n \\ s &= s^n + u_n * d + u_\varepsilon(d) \\ s^n &= s - u_n * d - u_\varepsilon(d) = s^n + u_n * d + u_\varepsilon(d) - u_n * d - u_\varepsilon(d) = s^n + 0, \end{aligned}$$

and the erroneous coefficients cancel each other out. Of course it works the same way if we add p_n to this

$$\begin{aligned} d &= d^n + p_n * s^n + p_\varepsilon(s^n) \\ s &= s^n + u_n * d + u_\varepsilon(d) = s^n + u_n * (d^n + p_n * s^n + p_\varepsilon(s^n)) + u_\varepsilon(d) \\ s^n &= s - u_n * d - u_\varepsilon(d) = s^n + u_n * (d^n + p_n * s^n + p_\varepsilon(s^n)) + u_\varepsilon(d) - \\ &\quad - u_n * (d^n + p_n * s^n + p_\varepsilon(s^n)) - u_\varepsilon(d) = s^n + 0 \\ d^n &= d - p_n * s^n - p_\varepsilon(s) = d^n + p_n * s^n + p_\varepsilon(s^n) - p_n * s^n - p_\varepsilon(s) = \\ &= d^n + 0. \end{aligned}$$

Via induction it can be shown that this holds true for any amount of p and n convolutions. But everything is not well with the integers, small errors will creep in since we do not have a true transform. Sometimes our coefficients in p and u will be factors like $^1/_3$ or the like, meaning we don't really have the transform that we set out to have. Fixed point arithmetic is a way of avoiding this, or at least making the errors as small as possible. The method uses a limited amount of fractions, meaning we use the group

$$a = \sum_{t=-n}^{m-2n} (a_i)^t,$$

where m is the amount of bits we are using, generally 8, 16 or 32, meaning we multiply everything with 2^n. The 2^n factor comes from the fact that in multiplication and division we will be operating with twice as high precision, $\frac{a2^n \cdot b2^n}{2^n} = \frac{ab2^{2n}}{2^n} = ab2^n$ and $\frac{a2^n \cdot 2^n}{b2^n} = \frac{a}{b}2^n$. Now if we make sure to first round the values up during the transform, and then down during the synthesis. This means that the roundoff errors, which are always positive if we round down, and always negative when we round up, add together, minimizing their effects to each other. This gives us an error free transform. At least it makes the transform stable enough to minimize the errors to maybe a colour here or there, depending on the amount of fixed points we use.

2.5 Two dimensional transform

Since we have just been dealing with one dimensional transforms so far, it is not completely clear how to do a two or multi dimensional transform. There are basically two ways, the easy way and the hard way. Generally

the easy way is the fastest and simplest, but some factors in the hard way makes it worthwhile. To make it simple we see the picture first as rows of one dimensional signals, and transform those. Then we transform them in the other direction as well. Now a one dimensional transform leaves us with half the coefficients as s. A two dimensional transform transforms both the columns of scaling coefficients and of wavelets, but there are only the new scaling coefficients in the columns of scaling coefficients, generally referred to as LL, that are used in the next transform. This leaves us with only $^1/_4$ of the initial data, meaning further transformations are greatly sped up. The trouble is actually locating this data points. If you remember the order in equation (15), the filterbanks generally moved the s coefficients to the left and the d to the right, giving us a partition that appears as.

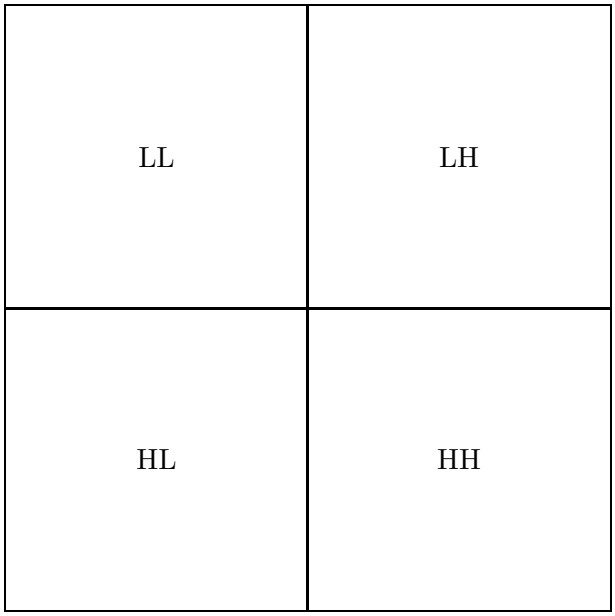


Figure 10: This is a sketch of the first level two dimensional filtering of a picture. Where we lay the coefficients with lowpass first and highpass later.

We then use the inverse transform to scale the little picture in the upper left corner, step by step, to the size of the original picture. As you see in the picture we have LL, LH, HL, HH bands. Those are the once that have ended up in s twice, d first then s, s first then d, and finally d both times. Being scaled as wavelets or scaling coefficients. Now with lifting this is a bit trickier to follow, since the different levels of coefficients will be spread all over the place. If we first look at the coefficients in an ordinary filterbank

scheme it will appear as.

$s^{3}_{1,1}$	$d^{3}_{1,1}$	$d^{2_x}_{1,1}$	$d^{2_x}_{2,1}$	$d^{1_x}_{1,1}$	$d^{1_x}_{2,1}$	$d^{1_x}_{3,1}$	$d^{1_x}_{4,1}$
$d^{3_y}_{1,1}$	$d^{3^x_y}_{1,1}$	$d^{2_x}_{1,2}$	$d^{2_x}_{2,2}$	$d^{3_x}_{1,2}$	$d^{3_x}_{2,2}$	$d^{3_x}_{3,2}$	$d^{3_x}_{4,2}$
$d^{2_y}_{1,1}$	$d^{2_y}_{2,1}$	$d^{2^x_y}_{1,1}$	$d^{2^x_y}_{2,1}$	$d^{3_x}_{1,3}$	$d^{3_x}_{2,3}$	$d^{3_x}_{3,3}$	$d^{3_x}_{4,3}$
$d^{2_y}_{2,1}$	$d^{2_y}_{2,2}$	$d^{2^x_y}_{1,2}$	$d^{2^x_y}_{2,2}$	$d^{3_x}_{1,4}$	$d^{3_x}_{2,4}$	$d^{3_x}_{3,4}$	$d^{3_x}_{4,4}$
$d^{1_y}_{1,1}$	$d^{1_y}_{2,1}$	$d^{1_y}_{3,1}$	$d^{1_y}_{4,1}$	$d^{1^x_y}_{1,1}$	$d^{1^x_y}_{2,1}$	$d^{1^x_y}_{3,1}$	$d^{1^x_y}_{4,1}$
$d^{1_y}_{1,2}$	$d^{1_y}_{2,2}$	$d^{1_y}_{3,2}$	$d^{1_y}_{4,2}$	$d^{1^x_y}_{1,2}$	$d^{1^x_y}_{2,2}$	$d^{1^x_y}_{3,2}$	$d^{1^x_y}_{4,2}$
$d^{1_y}_{1,3}$	$d^{1_y}_{2,3}$	$d^{1_y}_{3,3}$	$d^{1_y}_{4,3}$	$d^{1^x_y}_{1,3}$	$d^{1^x_y}_{2,3}$	$d^{1^x_y}_{3,3}$	$d^{1^x_y}_{4,3}$
$d^{1_y}_{1,4}$	$d^{1_y}_{2,4}$	$d^{1_y}_{3,4}$	$d^{1_y}_{4,4}$	$d^{1^x_y}_{1,4}$	$d^{1^x_y}_{2,4}$	$d^{1^x_y}_{3,4}$	$d^{1^x_y}_{4,4}$

.

Here the $d^{n_x}_{x,y}$ are the different coefficients we get in the LH sections, $d^{n^x_y}_{x,y}$ in HH and $d^{y}_{x,y}$ in HL. In the same manner we can write the lifting array as

$s^{3}_{1,1}$	$d^{1_x}_{1,1}$	$d^{2_x}_{1,1}$	$d^{1_x}_{2,1}$	$d^{3_x}_{1,1}$	$d^{1_x}_{3,1}$	$d^{2_x}_{2,1}$	$d^{1_x}_{4,1}$
$d^{1_y}_{1,1}$	$d^{1^x_y}_{1,1}$	$d^{1_y}_{2,1}$	$d^{1^x_y}_{2,1}$	$d^{1_y}_{3,1}$	$d^{1^x_y}_{3,1}$	$d^{1_y}_{4,4}$	$d^{1^x_y}_{4,1}$
$d^{2_y}_{1,1}$	$d^{1_x}_{1,2}$	$d^{2^x_y}_{1,1}$	$d^{1_x}_{2,2}$	$d^{2_y}_{2,1}$	$d^{1_x}_{3,2}$	$d^{2^x_y}_{2,1}$	$d^{1_x}_{4,2}$
$d^{1_y}_{1,2}$	$d^{1^x_y}_{1,2}$	$d^{1_y}_{2,2}$	$d^{1^x_y}_{2,2}$	$d^{1_y}_{3,2}$	$d^{1^x_y}_{3,2}$	$d^{1_y}_{4,4}$	$d^{1^x_y}_{4,2}$
$d^{3_y}_{1,1}$	$d^{1_x}_{1,3}$	$d^{2_x}_{1,2}$	$d^{1_x}_{2,3}$	$d^{3^x_y}_{1,1}$	$d^{1_x}_{3,3}$	$d^{2_x}_{2,2}$	$d^{1_x}_{4,3}$
$d^{1_y}_{1,3}$	$d^{1^x_y}_{1,3}$	$d^{1_y}_{2,3}$	$d^{1^x_y}_{2,3}$	$d^{1_y}_{3,3}$	$d^{1^x_y}_{3,3}$	$d^{1_y}_{4,4}$	$d^{1^x_y}_{4,3}$
$d^{2_y}_{1,1}$	$d^{1_x}_{1,4}$	$d^{2^x_y}_{1,2}$	$d^{1_x}_{2,4}$	$d^{2_y}_{2,2}$	$d^{1_x}_{3,4}$	$d^{2^x_y}_{2,2}$	$d^{1_x}_{4,4}$
$d^{1_y}_{1,4}$	$d^{1^x_y}_{1,4}$	$d^{1_y}_{2,4}$	$d^{1^x_y}_{2,4}$	$d^{1_y}_{3,4}$	$d^{1^x_y}_{3,4}$	$d^{1_y}_{4,4}$	$d^{1^x_y}_{4,4}$

.

In the latter array we can see that all elements of the $_x, _y$ and ^x_y families are arranged in squares with the center around their parents. This heritage actually plays a great role later in the SPIHT algorithm[9].

2.6 Inseparable two dimensional transform

The two dimensional transform presented above is kind of a trick, but a trick that works very well, the calculations are not too cumbersome. But there also do exist wavelet transforms that work truly in two or more dimensions. All we require is two dimensional filters with the same abilities as our one dimensional ones. First we need multi dimensional convolutions. If we were to convolve the function $f(x_1, x_2, \ldots, x_n)$ with the multi dimensional filter $g(x_1, x_2, \ldots, x_n)$ that would be

$$f(x_1, x_2, \ldots, x_n) * g = \sum_{k_1} \sum_{k_2} \cdots \sum_{k_n} g(k_1, \ldots, k_n) f(x_1 - k_1, \ldots, x_n - k_n).$$

[9]See chapter 3.7.1 on page 49.

But there is something much more important when designing our multi dimensional filters. If you remember we divided our function into even and odd parts. We can actually choose this division how we like. Only keeping every third, or even fourth sample, making different levels of wavelet coefficients out of the other. Now think of this as vectors. We get a one dimensional grid of equally spaced samples. Now we take a vector pointing at every scaling coefficient we want. Generally its $(2, 0)$, giving us every second sample. Then imagine if we increase it to two dimensions. Suddenly the amount of feasible combinations skyrocket. One of the simplest are the Quninox lattice.

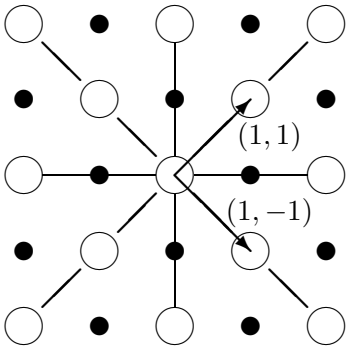


Figure 11: Here the big circles represent the Quinox lattice, the small black dots are the points outside of the lattice.

Generally this is described as putting the vectors for the lattice in a matrix, as is common for different bases. In the Quinox case this is of course

$$Q = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

An interesting fact is that $|det(Q)| = 2$, which is exactly the amount of different points we get with this lattice. One for the scaling coefficients, and one for the only set of wavelet coefficients. But we are not using this scheme, but rather a standard Cartesian one

$$D = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}.$$

Naturally this gives us $|det(D)| = 4$, which leaves us with one set of scaling coefficients s and the well known d^{n_x}, $d^{n_y^x}$ and d^{n_y}. The different wavelet filters h^x, h_y^x and h^y are of course defined by the three vectors to describe the indescribable points in our lattice, of course $x = (1, 0)$, $y = (0, 1)$ and $\begin{smallmatrix} x \\ y \end{smallmatrix} = (1, 1)$.

2.6.1 Two dimensional lifting

Naturally we can turn wavelet filters into a lifting scheme as well. It works in the same way with lattices and all, so let us directly start with our Cartesian lattice. Now we do not split our sequence up in just two threads, but rather four, as the determinant already told us. The first one is the scaling

coefficients s^0, the other three are the different translations as above. See figure (12) below for how this works.

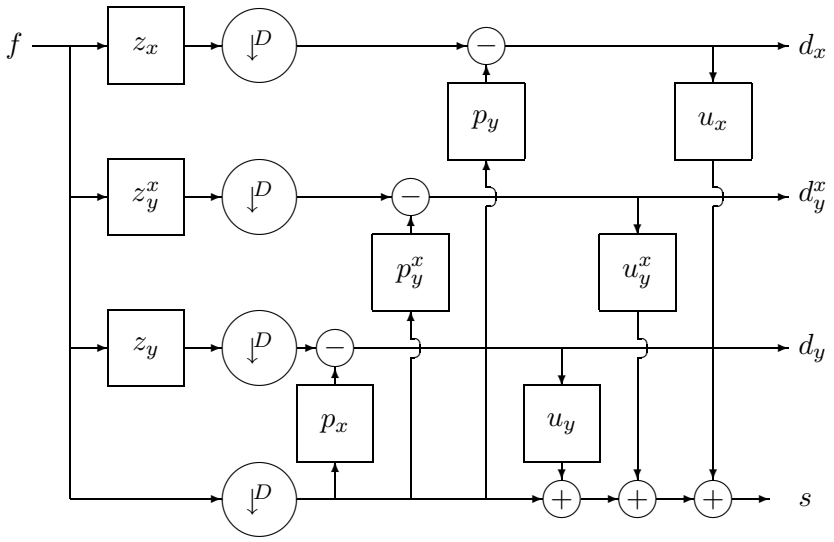


Figure 12: This is a graphical representation of two dimensional lifting with a Cartesian dilation matrix. Notice that the downsamples are noted with the dilation matrix. Meaning we just take the elements that fit in the lattice.

As you see we first subtract $p_1 * s^0$ from d^{0_x}, d^{0_y} and $d^{0_y^x}$, to create d^{1_x}, d^{1_y} and $d^{1_y^x}$. Then we add $u_1 * d^{1_x} + u_1 * d^{1_y} + u_1 * d^{1_y^x}$ to s^0 to get s^1. Then we end up with $^1/_4$:th of the original samples, and can do the same thing on those, finally ending up with the matrix we showed you previously, if we use writeback. This proves no problem, since we are not changing any of the d sequences when subtracting from s, and neither when we are working in the other direction, all streams work independently of the order of elements. The division looks like

$$\begin{bmatrix} s_{1,1} & d^{1_x}_{1,1} & s_{2,1} & d^{1_x}_{2,1} & \dots & s_{\frac{n}{2},1} & d^{1_x}_{\frac{n}{2},1} \\ d^{1_y}_{1,1} & d^{1_y^x}_{1,1} & d^{1_y}_{2,1} & d^{1_y^x}_{2,1} & \dots & d^{1_y}_{\frac{n}{2},1} & d^{1_y^x}_{\frac{n}{2},1} \\ s_{1,2} & d^{1_x}_{1,2} & s_{2,2} & d^{1_x}_{2,2} & \dots & s_{\frac{n}{2},2} & d^{1_x}_{\frac{n}{2},2} \\ d^{1_y}_{1,2} & d^{1_y^x}_{1,2} & d^{1_y}_{2,2} & d^{1_y^x}_{2,2} & \dots & d^{1_y}_{\frac{n}{2},2} & d^{1_y^x}_{\frac{n}{2},2} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ s_{1,\frac{n}{2}} & d^{1_x}_{1,\frac{n}{2}} & s_{2,\frac{n}{2}} & d^{1_x}_{2,\frac{n}{2}} & \dots & s_{\frac{n}{2},\frac{n}{2}} & d^{1_x}_{\frac{n}{2},\frac{n}{2}} \\ d^{1_y}_{1,\frac{n}{2}} & d^{1_y^x}_{1,\frac{n}{2}} & d^{1_y}_{2,\frac{n}{2}} & d^{1_y^x}_{2,\frac{n}{2}} & \dots & d^{1_y}_{\frac{n}{2},\frac{n}{2}} & d^{1_y^x}_{\frac{n}{2},\frac{n}{2}} \end{bmatrix},$$

and if we keep up the iterations we will finally end up with something looking like the previously lifted array.

What are the benefits of multidimensional lifting over one dimensional ditto. It is not less computionally effective, since the filters themselves have a tendency to grow enormously in every direction. This leads to a lot more of convolutions, and a lot more of overlapping. The benefits comes with us being able to use non separable filters, which only works in a set number of dimensions. Many such filters have very good uses in image coding, taking the different sensitivities of the human eye into consideration and similar optical effects. Another benefit is that we can work on parts of the array at a time, meaning we can work with huge arrays, or fit all our calculations in the cache.

2.6.2 Neville 2-d filter

One of the filters we have been looking closer in is the Neville 2D filter. It is a real two dimensional filter of variable size and amount of vanishing moments. It can be adapted to any downsampling matrix D, and it is easy to find filters with coefficients in the range of ${}^{i}/_{2^j},\ \{i,j\} \in \mathbb{Z}$. Sometimes we can even hold all the coefficients in ${}^{2^i}/_{2^j},\ \{i,j\} \in \mathbb{Z}$, meaning all computations can be handled with just bit shifts, but this requires a little more careful choice of the downsampling matrix. The interested reader can see how those filters are created in [KS97]. The filter we use is the Cartesian displacement filter with two vanishing moments. It look like

$$p_x = \begin{matrix} 0 & 0 & 0 \\ \frac{1}{2} & \underline{0} & \frac{1}{2} \\ 0 & 0 & 0 \end{matrix}, \qquad p_y^x = \begin{matrix} \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & \underline{0} & 0 \\ \frac{1}{4} & 0 & \frac{1}{4} \end{matrix}, \qquad p_x = \begin{matrix} 0 & \frac{1}{2} & 0 \\ 0 & \underline{0} & 0 \\ 0 & \frac{1}{2} & 0 \end{matrix},$$

where the underlined numbers have the same coordinates as the elements of s^{n-1} we are subtracting from. The multiples are of course what we multiply the corresponding p_{n-1} with. In the same manner we have

$$u_x = \begin{matrix} 0 & 0 & 0 \\ \frac{1}{4} & \underline{0} & \frac{1}{4} \\ 0 & 0 & 0 \end{matrix}, \qquad u_y^x = \begin{matrix} \frac{1}{8} & 0 & \frac{1}{8} \\ 0 & \underline{0} & 0 \\ \frac{1}{8} & 0 & \frac{1}{8} \end{matrix}, \qquad u_x = \begin{matrix} 0 & \frac{1}{4} & 0 \\ 0 & \underline{0} & 0 \\ 0 & \frac{1}{4} & 0 \end{matrix},$$

which are then subtracted from the corresponding elements of d^{n-1}, thus creating s^n and d^n. The observant reader will notice, that if we use the same lattice as explained above we can have an omni p filter that looks like

$$u = \begin{matrix} \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{4} & \underline{0} & \frac{1}{4} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \end{matrix}.$$

Here we can see that every element in s depends on the surrounding elements in d, the same way the elements in d is affected by the closest elements of s, assuring that our in place calculations work.

2.6.3 Inverse two dimensional lifting

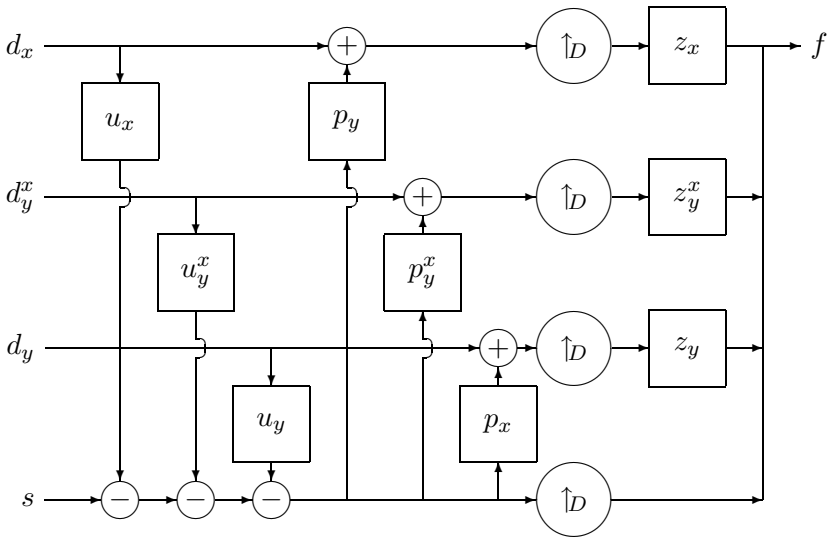


Figure 13: This is a graphical representation of two dimensional lifting with a Cartesian dilation matrix. Notice that the upsamples are noted with the dilation matrix, meaning we just take the elements that fit in the lattice.

This is really nothing more than the one dimensional lifting. We just apply our filters in the reverse order and sign, as above, to our matrix of numbers, see figure (13). Assuring perfect reconstruction or at least round off error annihilation, if we do not have infinite precision.

2.6.4 Fringe effects in two dimensions

Now the fringe effects is a whole different chapter, but not that complicated really. We simply wrap our filters around, counting the elements twice. For example, the first element in the upper left corner would have a p matrix of

$$u = \begin{matrix} \underline{0} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{matrix},$$

and the last element in the lower right corner would have a u matrix that looks like

$$p_y^x = \begin{matrix} 1 & 0 \\ 0 & \underline{0} \end{matrix},$$

since all but one the important elements end up outside of the s, d array we have to count it four times. The same goes for all borders. If we happen to have a bigger filter, we just have to wrap it around earlier, creating interesting side effects.

2.7 Multi dimensional transforms

Going from two dimensions to many dimensions are no more complicated than to go from one to two. For example an animated sequence of pictures is really a three dimensional cube of information. For the simple transform, we just first do it in the x-direction, then in the y-direction, then we do it in the z-direction. Now we can do it the same way as we did with x and y, only taking the relevant s elements, but in time it actually seems to be beneficial to transform every element, seeing even the sequences of $d^{n\binom{x}{y},x,y)}$ as functions. Those functions are of course treated in the usual manner, just keeping the low pass parts. The main benefit comes from the pictures that does not change. Even if they might be interesting in the initial picture, like a grainy wall, they will just show up as zeros in wavelet transform over time. The lifting scheme will as usual create elements all over the cube, but they will be evenly spaced in smaller cubes, like the small squares in the two dimensional transform.

2.7.1 Uneven transforms

The trouble comes when we wish to transform something that is not cubic. Generally it is not feasible to take a $256 \times 256 \times 256$ big cube. That many images in the time direction would eat up memory and computer power, as well as containing a lot of scene shifts, lowering our compression ratio. Maybe we also want a more rectangular display, like a TV or computer monitor. If we for example use a more feasible cube of $256 \times 128 \times 16$ we can divide the x-direction 8 times, the y-direction 7 times and the z-direction only 4 times. If we first just do the x-direction, we will have a cube that can be divided $(7, 7, 4)$ times. Then we can transform it in the x and y -directions three more times, then we can do all directions at once, ending up with our $256 \times 128 \times 16$ cube in wavelet space. When we then transform it back we just do it in the reverse order. But a much more general picture resolution is $320 \times 240 \times 24$, of course, the framedepth we can choose ourselves, and 16 sound like a good value, after all it is two thirds of a second. But 320 can only be evenly divided 6 times, and 240 only 4 times. This gives us a partition scheme of $(6, 4, 4)$, meaning we have to take the transforms in the order $\{x, x, x, y, z, x, y, z, x, y, z, x, y, z\}$. This gives us a final s being a cube of dimensions $5 \times 15 \times 1$. The rectangles of figure (14) represents this.

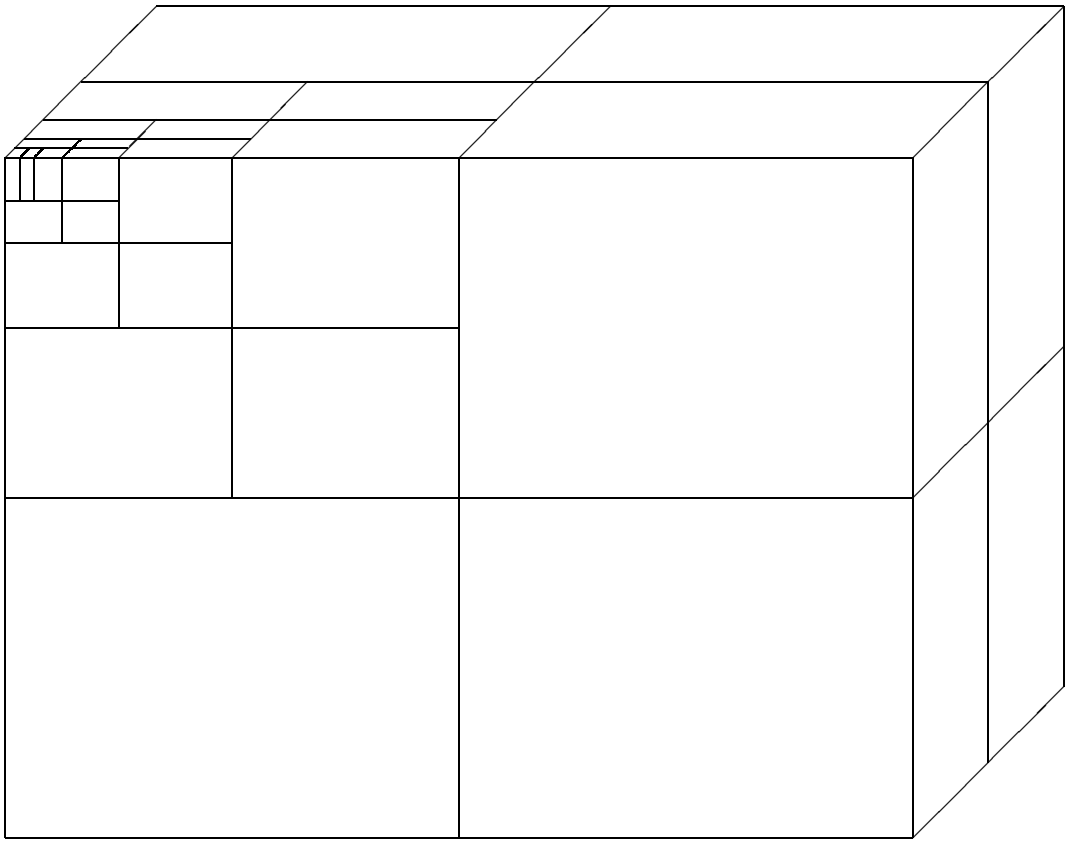


Figure 14: This is a graphical representation how a filterbank cube would develop in an uneven transform. The suare in the upper left corner is the $5 \times 15 \times 1$ pixel square that we start with. First extending it to a $10 \times 15 \times 1$, then to $20 \times 15 \times 1$, followed by $40 \times 30 \times 2$ and then three more times till we reach our resolution of $320 \times 240 \times 16$.

But if we take an ordinary TV resolution of 379×288 we can only divide it evenly once, in the y-direction. This does not give us any good transform. One way of doing this is to divide it into smaller blocks, and transform them. For example $379 \times 288 \times 16 < (24 \times 18)(16 \times 16 \times 16)$. But we would rather suggest that you pad the functions with a known polynomial, preferably zero to create the least coefficients in the fringes. This would of course yield a set of more elements, but it is a quick and dirty way of doing it. If we for example transform a set of 17 elements, it would look like

$$\begin{aligned} f &= \{e_1, e_2, \ldots, e_{17}\} \\ f^0 &= \{e_1, e_2, \ldots, e_{17}, 0_{18}, \ldots, 0_{32}\} \\ f^1 &= \{s_1^1, s_2, \ldots, s_9^1, 0_{10}, \ldots, 0_{16}, d_1^1, \ldots, d_9^1, 0_{25}, \ldots, 0_{32}\} \\ &\vdots \\ f^5 &= \{s_1^5, d_1^5, d_1^4, 0_4, d_1^3, d_2^3, 0_7, 0_8, \ldots, d_1^1, \ldots, d_9^1, 0_{25}, \ldots, 0_{32}\}. \end{aligned}$$

Of course the synthesis would look like

$$\begin{aligned} &\vdots \\ f^0 &= \{e_1, e_2, \ldots, e_{17}, 0_{18}, \ldots, 0_{32}\} \\ f &= \{e_1, e_2, \ldots, e_{17}\}, \end{aligned}$$

meaning we just ignore everything past element 17. In a cube this means a lot of redundant zeroes, still taking memory space and computing power, so it is recommended to at least stay to somewhat even divideable resolutions.

2.7.2 Interpolation

Since wavelets work with moving polynomials to the lowpass part of it's stream, it also makes polynomial approximations between samples. This means that scaling up a wavelet transform interpolates between the pixels, if we choose to scale it up.

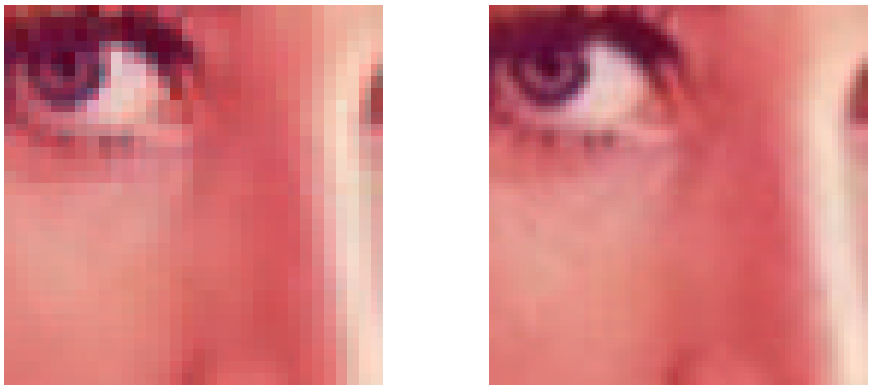


Figure 15: Here we see an example of wavelets ability to interpolate pictures. To the left we have zoomed in Lena's eye. Observe the blockyness around the iris and the stripes over the nose. To the right we have a scaled up version, using a simple wavelet which gives us a second degree spline approximation. The blockyness over the nose is still there, but around the tear channel and the iris it looks much smoother.

3 Video coding

This section will deal with the practical part of the thesis, namely video coding. The two first parts will give an introduction to compression and computer graphics in general. Chapter 3.4 will describe the video codec[10] in general terms, and the following sub chapters will describe the various parts of the video coding process in more detail. The preceding theory chapters are not necessary for understanding how wavelets can be used in video compression, but are recommended reading for those who want to understand the mathematical theory behind wavelets.

3.1 Compression

When speaking of compression in general, two different modes of operation exist: lossless and lossy compression.

Lossless compression is, as the name suggest, lossless and does not lose any information at any stage of the compression or decompression operation. Compression of data works by exploiting redundancy in the input data, where the redundancy can be of contextual or alphabetic nature. Truly random data can not be compressed as it lacks redundant information. It is not possible to compress, without loss, all combinations of input data of a certain size, with at least one bit[11]. Lossless compression is used to compress documents, programs and other data that must be intact after decompression. Compression ratios are usually between 1:2 and 1:8 depending on input data.

Lossy compression on the other hand do lose information in the compression stage, but the information lost is usually of lesser importance. Compression works by exploiting shortcomings or limitations in the human perception model. For example: when compressing audio, frequencies below 20 Hz and above 20 kHz are not perceptible by humans, and can be excluded from the audio data without noticeable degradation in audio quality[12]. Lossy compression also benefits from redundancy in input data, but the biggest gain comes from dropping unnecessary information. Throwing away too much information will result in a reduction in quality, but on the other hand higher compression ratios are achieved. Lossy compression is used to compress audio, images, video and other data where perfect reconstruction is not necessary. Compression ratios are usually between 1:10 and 1:100 depending on level of quality.

[10] Abbreviation for COder/DECoder.

[11] A counting argument using the pigeon-hole principle proves this.

[12] Although we can not hear frequencies below 20 Hz, we can sense or feel them by other means

3.2 Computer images and video

An image in a computer consists of several ($width \times height$) dots of different colors, these dots are called pixels. Each pixel is described by one or three values. If a pixel consists of only one value, the computer uses the value of the pixel to look up the color in a palette. If a pixel consists of three values, each value represents the amount of red, green and blue. The former color format is called *indexed color*, and the latter is called *true color*. The amount of bits in a pixel is called color depth, and the number of different colors a pixel can have is given by: 2^{depth}. The indexed format uses 1, 4 or 8 bits for depth, and the true color format uses 24 bits (8 bits for each value). We will be using the true color format, or RGB format as it is sometimes known as. The values of a pixel in RGB format are usually stored sequentially, but they can also be interlaced in a couple of different ways as table 1 shows. The interlacing schemes are not limited to RGB format but are also used by other color space formats that chapter 3.5 will deal with.

Interlace	Format
None	RGBRGBRGBRGBRGBRGB
Scanline	RRR ... GGG ... BBB ... RRR ... GGG ... BBB ...
Plane	RRRRRR ... GGGGGG ... BBBBBB ...
Partition	Like plane, but in individual files

Table 1: Interlace scheme for raw image data

An image in a computer is stored as a string that is $width \times height$ items long, with the first item having index 0. It is still possible to address the pixels of an image by Cartesian coordinates[13] using the following translation equation:

$$(x, y) = x + y \times width$$

A video in a computer consists of several consecutive images (frames) of the same resolution and same color depth at some frame rate (frames per second). The resolution and frame rate may be chosen arbitrary, but is usually of a standard resolution and frame rate derived from the analog TV/video world (See table 2 for an overview).

A video in a computer is usually stored as several images after each other, making the string stored in memory $widht \times height \times frames$ items long. Addressing the pixels by Cartesian coordinates can be done by:

$$(x, y, z) = x + y \times width + z \times widht \times height$$

[13]Origo, in computer graphics, is located at the top left corner of the image.

Standard	Resolution	Frame rate
NTSC TV	720×480	29.97
CIF (NTSC)	352×240	29.97
QCIF (NTSC)	176×120	29.97
PAL TV	640×576	25
CIF (PAL)	352×288	25
QCIF (PAL)	176×144	25

Table 2: This is a table of different digital video standards. We can see that CIF and QCIF can be divided with two many more times that NTSC and PAL.

A video format describes the video, and usually it consists of a header that contains various parameters like frame rate and such, maybe an index over the frames and finally the actual video data. Some formats also describe what codec (compression/decompression engine) was used, like QuickTime, so the way to compress the video data can be chosen. Other formats, like MPEG-1, have specified what codec is to be used.

Some video codecs compress only one frame at a time, much like compressing a single image. This technique removes spatial redundancy. Other codecs take a step further and compress a group of consecutive frames. Consecutive frames usually contain a lot of temporal redundancy, and this can be exploited to remove both spatial and temporal redundancy resulting in even higher compression ratios.

3.3 Video and wavelets

First we will take a quick look at how wavelets can be used with images and then extend that view to video.

An image can be seen as a signal in the time domain, where time actually is the location of the individual pixels. A signal in the time domain can be transformed to the frequency domain by a number of different transforms. Fourier transform being one, and wavelet transform another.

When an image has been wavelet transformed, the pixels are called *coefficients* because the value of the pixel no longer represent color but rather amplitude. Coefficients with high amplitude represents big changes, and low amplitude represents small changes. The coefficients are also grouped according to how big part of the image they affect. Coefficients that influence a bigger part of the image are fewer than the coefficients that influence smaller parts. Refer to figure (16) for a graphical representation of the coefficient grouping. The different groups are also called *sub bands*. When talking about low- and high pass sub bands, we are talking about what frequencies they contain.

Each area of figure (16) represents the original image with the most

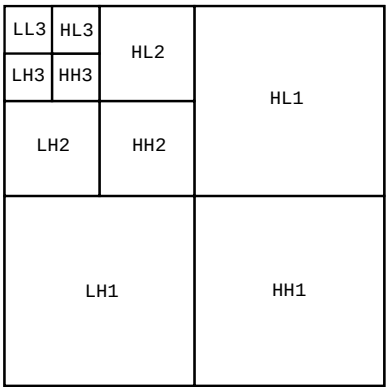


Figure 16: Coefficient grouping after three filter bank transforms in two dimensions

important coefficients at the top left corner. When recreating the image by doing the inverse wavelet transform, the coefficients from the top left corner will give a rough approximation of the original image. Using more coefficients from the adjacent regions will add more and more detail to the recreated image. The coefficients of the lowest importance will usually be zero or close to zero.

Figure (16) represents a single image after wavelet transformation in two dimensions, x and y. To transform a sequence of images, a two dimensional wavelet transform is applied on all images (see left part of figure (17)). This removes spatial redundancy from all images. In order to remove temporal redundancy, a one dimensional wavelet transform is applied in the z dimension involving all frames. See right part of figure (17) for a visual representation of how the coefficients are grouped.

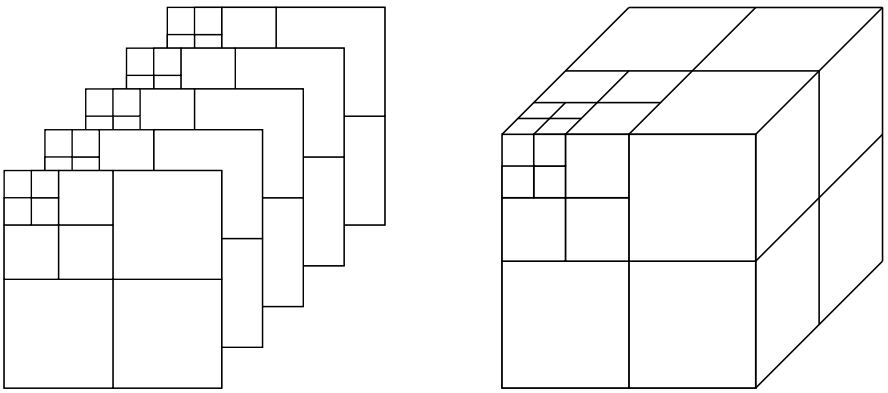


Figure 17: To the left we have a representation of many still images, coded one after another. The right cube represents a true three dimensional transform of a series of images.

3.4 Video codec

Depicted in figure (18) is a principal sketch of our video codec[14]. Not all steps are required for a video codec, and some steps were omitted[15]. There is no wrong or right way to design a video codec, as long as it does what it is supposed to. On the Internet we found a lot of information about video codecs and compression techniques, that helped us when we designed our codec. To uncompress an image or a video, the steps in figure (18) are traversed in reverse order with inverse transforms of all steps.

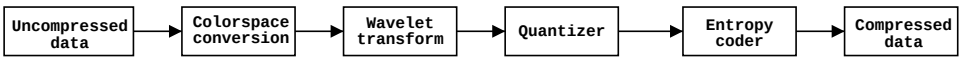


Figure 18: Video codec

Our video codec can compress video in both single and multiple frame mode. Single frame mode is when only one frame at a time is compressed, and multiple frame mode does a wavelet transform on a sequence of frames. The latter mode uses a wavelet transform in the z dimension to remove temporal redundancy, whereas the former does not.

Single frame mode has the advantages of lower memory usage, quicker response to changes in bit rate but the disadvantage of temporal redundancy.

3.5 Color space transform

A color space is a representation of a set of colors. Table 3 lists some of the most popular color spaces and their field of use.

Color space	Field of use
RGB	Computer graphics
YIQ	Analog Video
YUV	Analog Video
YC_bC_r	Digital Video
CMY(K)	Printing

Table 3: Color spaces

We will be using the YC_bC_r (ITU-R BT.601) format, which is a scaled and offset version of the YUV color space. To convert from RGB format to YC_bC_r the following equations can be used:

[14] COder/DECoder engine

[15] For example: motion compensation is a technique that we did not look into at all, and later the entropy coding step was removed.

$$
\begin{aligned}
Y &= 0.257R + 0.504G + 0.098B + 16, \\
C_b &= -0.148R - 0.291G + 0.439B + 128, \\
C_r &= 0.439R - 0.368G - 0.071B + 128.
\end{aligned}
$$

And to convert back:

$$
\begin{aligned}
R &= 1.164(Y - 16) + 1.596(C_r - 128), \\
G &= 1.164(Y - 16) - 0.813(C_r - 128) - 0.391(C_b - 128), \\
B &= 1.164(Y - 16) + 2.018(C_b - 128).
\end{aligned}
$$

The previous equations have been modified for use with computer graphics, where R, G and B have values between 0 and 255 (8 bits). When converting between RGB and $\mathrm{YC}_b\mathrm{C}_r$ format, the values should be saturated (clamped) at the 0 and 255 levels to prevent under- and overflow wraparound problems.

In the RGB format each of the three values represent the amount of red, green and blue. In $\mathrm{YC}_b\mathrm{C}_r$ format, the Y represents the amount of luminance (brightness), C_b and C_r represent the chroma (color). The human eye interprets the world by cones (color sensitive cells) and rods (intensity sensitive cells). The rods outnumber the cones, giving the human eye a lower resolution in color than luminance. This means that we can store the color information in a picture with much lower resolution.

In $\mathrm{YC}_b\mathrm{C}_r$ format the chroma components of an image are often subsampled to a quarter of its original size. We chose not to subsample the chroma channels, instead we later compress those channels much harder. When we convert to $\mathrm{YC}_b\mathrm{C}_r$ format, we store the different values according to the planar interlacing scheme (see table 1).

3.6 Discrete wavelet transform

The theory for the wavelet transform has been discussed in chapter 2.4, and in this chapter we will take a closer look at how we did the wavelet transforms on the computer.

We will be using the integer lifting scheme (see chapter 2.4.2) as this allows us to use ordinary integers instead of floats. In our codec, this step comes after the color transformation step. The color transformation step outputs the image data in three channels, so we need to do the wavelet transform on all three channels. For the sake of simplicity only one channel is considered in any pseudo code, example or figure in this chapter.

We will be using both one, two and three dimensional wavelet transforms. The transition from one to two and three dimensions is not very complex but we will start by explaining the one dimensional first. The one dimensional

transform is done on an array of pixels, dividing the array to scaling and wavelet coefficients as depicted in figure (19). Let, for illustrative purposes, the scaling coefficients be marked as a cross and the wavelet coefficients marked as a ring, and the original pixels marked as a dot. Different wavelet filters are applied depending on whether the coefficient is a scaling or a wavelet coefficient. The filters will be discussed later in this chapter.



Figure 19: A representation before and after the transform. Here the $\times$ represent the lifted low pass coefficients s_n and $\circ$ represent the high pass coefficients, c_n.

The array will be transformed $\log_2(n)$ times where n is the number of items in the array. The array does not have to be transformed the maximum number of times, but doing so gives better compression. Only the scaling coefficients are considered in subsequent runs, that is: disregard the wavelet coefficients and pretend that the scaling coefficients are pixels when doing another wavelet transform on the array. Let the number of wavelet transforms an array has been subjected to be called *levels*. Which coefficients are scaling or wavelet coefficients depends on what level the array is at. Figure (20) shows the steps when an array of eight pixels is being transformed three times, and last is shown the number of transforms each coefficient has been involved in.

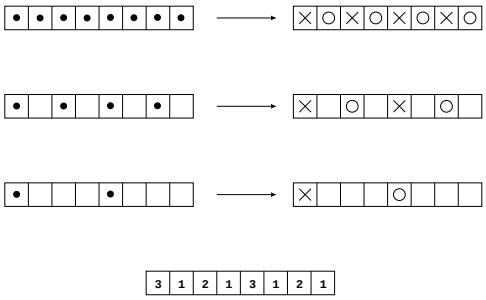


Figure 20: This is a representation of how we keep only the low pass coefficients, using them for the next level of transform. The last bar only shows how many times each cell is the member of any transform.

To extend this view to two dimensions, a matrix is used instead of an array. The scaling and wavelet coefficients are divided in both dimensions, leaving only a quarter of the coefficients available for the next run. The two dimensional counterpart being transformed three times is depicted in figure (21), along with a matrix showing the number of transforms each coefficient has been involved in.

The different steps for a three dimensional wavelet transform are ob-

tained in a similar manner. This time the coefficients are divided in three dimensions, leaving only $^1/_8$ of the coefficients available for the next run.

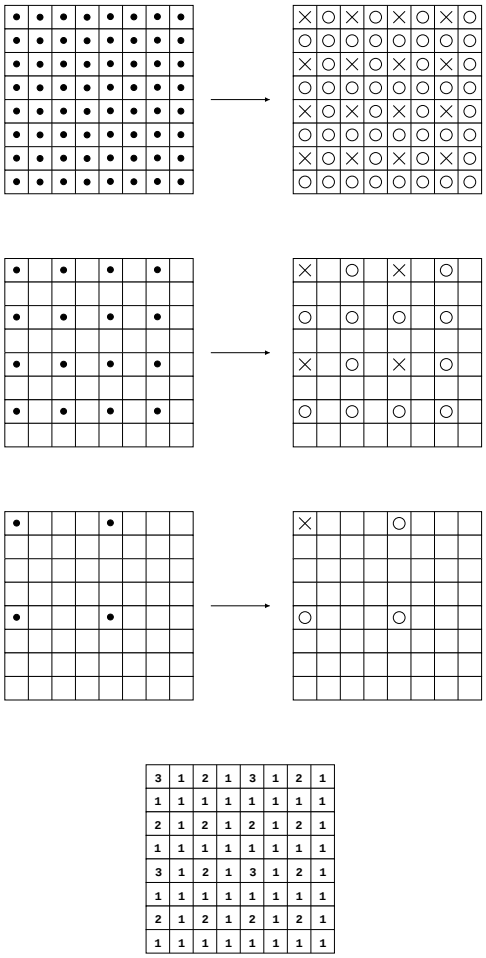


Figure 21: A representation of which elements, the low pass coefficients, are kept in two dimensional lifting transform. The last square shows how many times each element is calculated.

As the observant reader already have noted, the two dimensional wavelet transform obtained in this chapter does not look like the one on figure (16), and neither does the three dimensional counterpart. The only difference is that the coefficients are not grouped together according to what level they belong to. The wavelet transform can be made "in place" if the coefficients are left where they were. There exists wavelet transforms that will produce results like figure (16), and the wavelet scheme we used can be altered to do so. However, figure (16) is easier to understand and will often be referred to, even though the coefficients actually are positioned as in figure (21).

Now we will take a look at what wavelet filters we used, and how they

work. The first filter we built was the one dimensional Haar filter (see page 18), as it is small and easy to work with. To obtain the scaling and wavelet coefficients for an array, different filters are applied on different pixels. Let us call the pixels that will become scaling coefficients for *o-pixels*, and the wavelet coefficients for *x-pixels* (refer to figure (19) for positions of the pixels). The forward one dimensional Haar filter can be obtained by:

$$\begin{array}{lcll} \text{x-pixel} & = & -1 & \bullet \\ \text{o-pixel} & = & \bullet & +1/2 \end{array}$$

Note that the order in which the filters are applied does matter, and that the bullet represents the pixel which the filter is applied on. For example: the x-pixel will the sum of itself and -1 times the o-pixel before, the o-pixel will be the sum of itself and $+1/2$ times the x-pixel after. O-pixels get their value only from other x-pixels and itself, x-pixels get their value only from o-pixels and itself. The filter is applied on every pixel in the array, and subsequent runs are applied on only the scaling coefficients for that level (refer to figure (20)). The inverse, or reverse, filter is obtained by:

$$\begin{array}{lcll} \text{o-pixel} & = & \bullet & -1/2 \\ \text{x-pixel} & = & +1 & \bullet \end{array}$$

The two dimensional Haar filter is somewhat trickier as it involves more pixels. Refer to figure (21) for positions of o- and x-pixels. The two dimensional filter involves one o-pixel and three x-pixels placed as follows:

$$\begin{array}{ll} o & x_0 \\ x_1 & x_2 \end{array},$$

and the forward filter is obtained by applying the following in order:

$$x_0 = \begin{array}{ll} -1 & \bullet \\ 0 & 0 \end{array}, \quad x_1 = \begin{array}{ll} -1 & 0 \\ \bullet & 0 \end{array}, \quad \begin{array}{ll} -1 & 0 \\ 0 & \bullet \end{array}, \quad o = \begin{array}{ll} \bullet & +1/4 \\ +1/4 & +1/4 \end{array}.$$

The reverse filter is:

$$o = \begin{array}{ll} \bullet & -1/4 \\ -1/4 & -1/4 \end{array}, \quad x_0 = \begin{array}{ll} +1 & \bullet \\ 0 & 0 \end{array}, \quad x_1 = \begin{array}{ll} +1 & 0 \\ \bullet & 0 \end{array}, \quad x_2 = \begin{array}{ll} +1 & 0 \\ 0 & \bullet \end{array}.$$

The Neville 2-tap filter was the second filter we built, and it is a little larger than Haar and somewhat more complex (see chapter 2.6.2). However, higher compression ratios are achieved with Neville than Haar and the square behavior is reduced. The forward one dimensional Neville filter is:

$$x = \begin{array}{lll} -1/2 & \bullet & -1/2 \end{array}, \quad o = \begin{array}{lll} +1/4 & \bullet & +1/4 \end{array},$$

and the reverse filter is:

$$o = \begin{array}{ccc} & -1/4 & \\ -1/4 & \bullet & -1/4 \\ & -1/4 & \end{array}, \quad x = \begin{array}{ccc} & +1/2 & \\ +1/2 & \bullet & +1/2 \\ & +1/2 & \end{array}.$$

Because the Neville filter is longer, problems will occur at the beginning and at the end of the array (e.g. there is no x-pixel before the first o-pixel). To remedy this problem we simply folded the filter at the edges, so that a pixel is counted twice. The same solution can be applied to the two dimensional Neville filter when problems occur at the edges and at the corners. The forward two dimensional Neville filter is:

$$x_0 = \begin{array}{ccc} 0 & 0 & 0 \\ -1/4 & \bullet & -1/4 \\ 0 & 0 & 0 \end{array}, \quad x_1 = \begin{array}{ccc} 0 & -1/2 & 0 \\ 0 & \bullet & 0 \\ 0 & -1/2 & 0 \end{array},$$

$$x_2 = \begin{array}{ccc} -1/4 & 0 & -1/4 \\ 0 & \bullet & 0 \\ -1/4 & 0 & -1/4 \end{array}, \quad o = \begin{array}{ccc} +1/8 & +1/4 & +1/8 \\ +1/4 & \bullet & +1/4 \\ +1/8 & +1/4 & +1/8 \end{array},$$

and the reverse filter is:

$$o = \begin{array}{ccc} -1/8 & -1/4 & -1/8 \\ -1/4 & \bullet & -1/4 \\ -1/8 & -1/4 & -1/8 \end{array}, \quad x_0 = \begin{array}{ccc} 0 & 0 & 0 \\ +1/4 & \bullet & +1/4 \\ 0 & 0 & 0 \end{array},$$

$$x_1 = \begin{array}{ccc} 0 & +1/2 & 0 \\ 0 & \bullet & 0 \\ 0 & +1/2 & 0 \end{array}, \quad x_2 = \begin{array}{ccc} +1/4 & 0 & +1/4 \\ 0 & \bullet & 0 \\ +1/4 & 0 & +1/4 \end{array}.$$

Because the coefficients can be negative, an extra bit is needed for the sign. The values can expand even more, but usually not by more than a bit or two.

Since divisors are a power of two, the division operations can be replaced by bit shifts, and the other operations are addition and subtraction, making these filters computationally quite efficient. Of course, not all values are evenly divisible by two, four or eight, so using integer arithmetic will lead to rounding errors. However, by using a programming technique called fixed point arithmetic (see chapter 3.9), rounding errors can be reduced at the expense of using more bits for the coefficients.

3.7 Quantization

So far, as our video codec is concerned, no information has been lost[16], and no actual compression have taken place. This chapter will deal with the lion

[16]Some information may have been lost due to rounding errors.

share of the compression, or information loss if you will, of the codec.

Quantization is a method to reduce the number of unique values in a set. For example: any integers between 5 and 15 will be 10, values between 15 and 25 will be 20 and so on. Figure (22) shows a sinusoid that has been quantized at different thresholds. A high threshold will result in a poor representation of the signal and vice versa. Some information is lost, depending on how high the threshold is. However, the information loss will result in added redundancy in the signal, which can be later removed by entropy coding (see chapter 3.8).

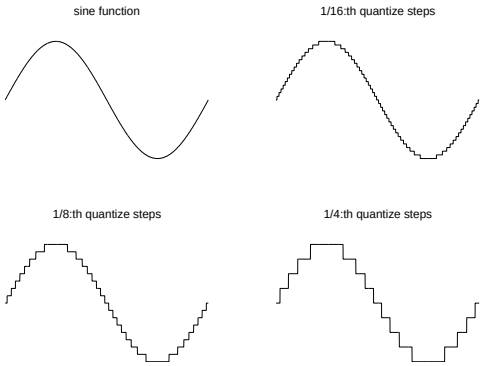


Figure 22: The quantization decreases the amount of data points needed, but as we clearly can see it is at the cost of precision.

The compression in this step comes from the fact that the signal, after quantization, can be described with a smaller alphabet and the signal now contains more redundant information than before.

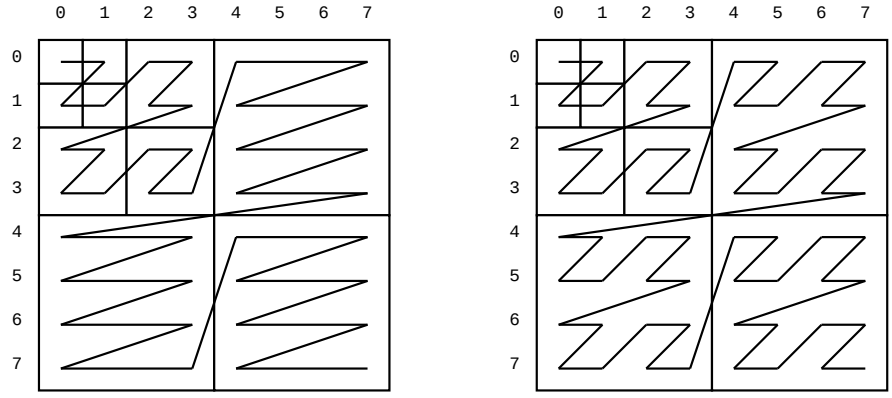


Figure 23: To the left we have the raster scan order, it might look simpler than the Morton scan order on the right. But Morton scan order keeps us in the same general part of the image, it is also easy to do recursively.

Usually the most important coefficients will be grouped in the top left corner (see figure (16)), with importance decreasing for each sub band, and a lot of the coefficients in the lower sub bands will have a value close to zero. Since small values represent small changes in the original image, they can often be discarded without visible loss of quality. Discarding low valued coefficients[17] will increase compression for a modest degradation in quality. However, doing so will lead to a problem: the coordinates of the coefficients. A lot of coefficients will be zero, and storing them gives no compression. The problem is closely related to the problem with sparse matrices. On the other hand, not storing them leads to the problem with coordinates: what coefficient should be at what coordinate? One solution is to define a scan order to follow. Two such scan orders are depicted in figure (23) for a eight by eight matrix.

The coefficients are simply stored in the order they are found according to scan order. When the desired compression ratio is achieved, storing is stopped. With this technique some coefficients that are stored are close to zero or zero, thus lowering the compression ratio. There exists a number of algorithms that address this problem. The EZW (Embedded Zero Tree) algorithm by Shapiro [Sha93] is a way to both quantize and store the coordinates of the coefficients. The SPIHT (Set Partitioning in Hierarchical Trees) algorithm by Said and Pearlman [SP96] can be described as a more advanced version of the EZW algorithm. After evaluating both algorithms we chose to implement the SPIHT algorithm which is described in the next chapter.

3.7.1 SPIHT

The SPIHT (Set Partitioning in Hierarchical Trees) algorithm by Said and Pearlman [SP96] is an algorithm that stores the most important coefficients in an embedded way, along with information for the coordinates. Quantization is done implicitly by starting with a high threshold that is successively lowered as the algorithm progresses.

Examine figure (24) and note the similarity between the different sub bands. A tree like structure can be built from this observation, where each node have four branches except the top node which is a special case (refer to figure (25)). A closer inspection of figure (24) shows that usually a bright patch in the lower[18] sub bands seems to propagate into the higher sub bands. In this figure, white stands for a low value and black stand for a high value. Low values stand for small changes in the original image, and are thus of lesser importance. What we want to do is to store the values of all the coefficients above a certain threshold along with information about the coordinates. Since we have this tree like structure, we can represent all the

[17] We do not really discard them, only set their value to zero.

[18] The closer a sub band is to the top left, the lower the frequencies it contains are.

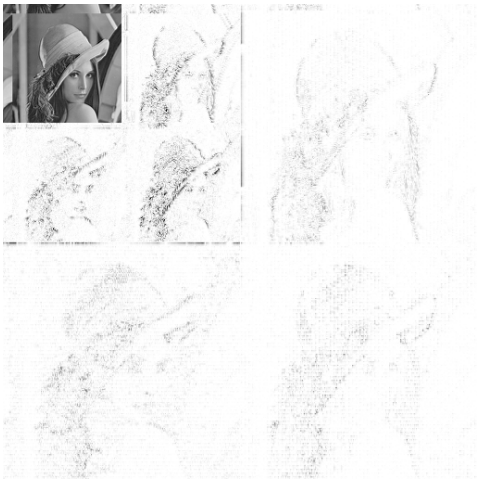


Figure 24: This is the filter bank transform of Lena. It can be noted that the uninteresting areas, like the hat and shoulder, turns out white in the high sub bands, whereas the feathers and other detailed areas are described with many high coefficients.

coordinates of a branch with no coefficients over a certain threshold with just one bit.

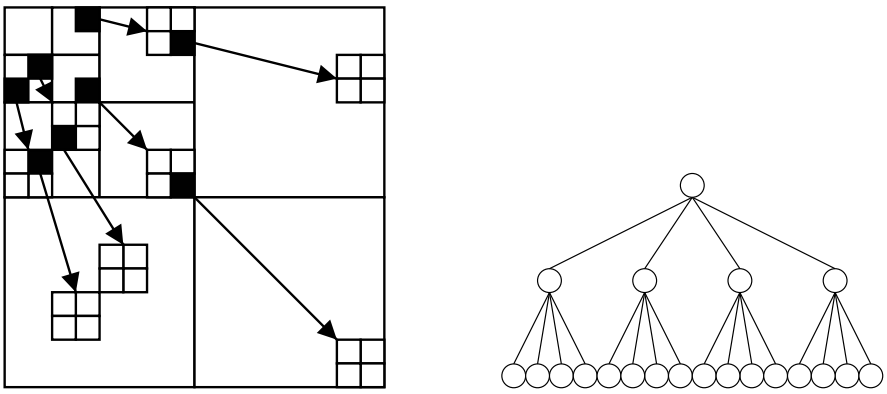


Figure 25: This is an illustration how the parents expand to four clustered children in a filter bank set of coefficients. We can see that the children stay in the vicinity of the parent when the picture is scaled up.

The SPIHT algorithm uses three lists to keep track of the nodes. The lists are: sign-list, desc-list and ref-list. The sign-list contains nodes that will be checked for significance, the desc-list contains nodes whose descendants will be checked for significance and the ref-list consists of nodes that have been found significant earlier. The nodes in the desc-list can be of two kinds: normal nodes and multi nodes. A multi node can be seen as a node that

represents all it's four children at once.

The algorithm consists of three major parts: the sign-pass, the desc-pass and the ref-pass.

sign-pass: Here the significant coefficients are identified.

desc-pass: Sometimes a node may have significant descendants, and those are identified in this pass.

ref-pass: In this pass the values of identified coefficients are refined in subsequent passes.

Below is a short description of each pass, and a code listing in pseudo code in appendix C.1.

The algorithm is initialized by adding the four first nodes to the sign-list, and the three children of the top node to the desc-list marked as normal nodes. The threshold is initially equal to the highest bit of the coefficient with the largest absolute value. The three passes are executed until desired compression ratio is achieved. After each run, the threshold is halved.

The sign-pass consists of checking the absolute value of each node in the sign-list if it is significant with respect to the current threshold. If the node was significant it is moved from the sign-list to the ref-list, and a bit is written to the output to indicate that the node was significant, along with a bit to indicate the sign of the node. Else a bit is written to output to indicate that the node was not significant.

In the desc-pass, each node in the desc-list is checked if it is a multi node or a normal node. If the node is a multi node, it's four children are checked for significant descendants. The outcome of that check is written to output. If there were any significant descendants, all four children are added to the desc-list marked as normal nodes and the current node is removed from the desc-list. On the other hand, if the node was a normal node it is checked if it has any descendant that is significant with respect to the current threshold. A bit is written to the output to indicate the outcome of the check. If the node is deemed to have significant descendants it is moved to the end of the desc-list marked as a multi node only if it has grandchildren else it is removed, and a sign-pass is executed with a temporary sign-list consisting of the node's children. The children that was not moved to the ref-list as a result of the sign-pass, are moved to the real sign-list.

The ref-pass checks all nodes in the ref-list, except those nodes that were added in the most recent sign-pass, with respect to the current threshold, and outputs the corresponding bit. This pass refines the values of the coefficients.

The description above is for two dimensional data, but extending to three dimensions is easy: each node have eight branches instead of four.

Decoding is done in a similar manner: all output parts are exchanged for input and all comparisons with threshold are exchanged for updates to the coefficients. When updating the coefficients, their initial value is set to 1.5 times the current threshold, because the threshold tells us that the value is at least threshold but less than two times threshold. So a good guess might be that the value will be somewhere in between. Of course, when refining the values in the ref-pass, one has to take into consideration that the values have been interpolated. The tree structure is only for understanding the algorithm. When we implemented the algorithm we ordered the data according to Morton scan order.

The SPIHT algorithm relies on the fact that a high value represent an important coefficient, and that coefficients in the lower sub bands usually are much larger than those in the higher sub bands. However, the wavelet transforms that we use produce coefficient values of roughly the same magnitude in all sub bands. This can be easily rectified by either making sure in the wavelet transforms that the coefficient values expand for each sub band, or by making a small adjustment to the SPIHT algorithm: when comparing the node against the threshold make sure to also consider which sub band that node is from.

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

0	16	4	17	1	20	5	21
32	48	33	49	36	52	37	53
8	18	12	19	9	22	13	23
34	50	35	51	38	54	39	55
2	24	6	25	3	28	7	29
40	56	41	57	44	60	45	61
10	26	14	27	11	30	15	31
42	58	43	59	46	62	47	63

Figure 26: Numerical correspondence between Morton scan order and the order the wavelet transform outputs the data in.

There is still one hurdle to overcome: the output from the wavelet transform look nothing like figure (16) (see figure (21) for how the coefficients actually are grouped), and subsequently the Morton scan order can not be used directly. We developed an algorithm to transform the coefficient order directly to Morton scan order.

The relations between the different sub bands in the wavelet transformed data can not be seen directly, but the tree structure can be worked out. Refer to figure (27) and (28) for a mental image of the tree structure. If the coefficients are numbered according to Morton scan order, the matrix will

look like the right part of figure (26). The left part is the matrix with corresponding numbers for how the coefficients appear after the wavelet transform. Compare the right part with figure (27). Although the coordinates of a node with branches looks hairy, three different patterns can be seen.

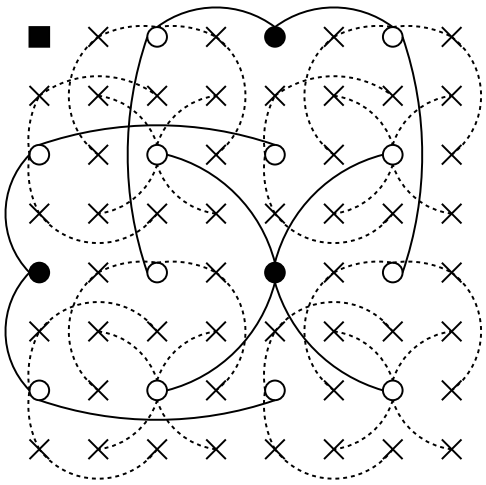


Figure 27: This is the order of the branches and children in two dimensional lifting

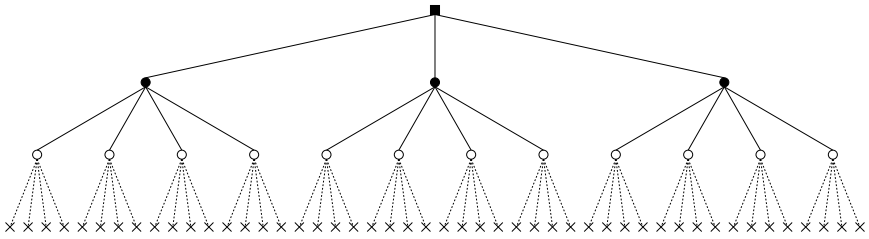


Figure 28: The data from the square above, now ordered in a tree according to the paths we traverse the square in.

With a little imagination M-, E- and X-like patterns can be spotted. Let us take a closer look at the M-like pattern, in figure (27), originating from coordinate (3,0). The black bullet has four children, and each child branches in a similar manner, only with smaller steps. The four children can be seen as the four corners of a square. Let a square represent all four children. A new smaller square appears at each corner of the square. The step size is halved for each branch, and the coordinates of each new square can be worked out. The same is true for the other patterns, the only difference is that the starting node is at another coordinate and that the imagined square propagates differently. Pseudo code for an algorithm that sorts two dimensional wavelet data can be found in appendix C.2. An implementation for three dimensional data is similar, each node have eight children instead of four and one have to look for a cube instead of a square.

The inverse wavelet transform expects the input data to be formatted in the same way that the forward wavelet transform wrote it, so we need an inverse version of the above algorithm as well. Once the forward version is understood, a reverse version is easy to do.

3.8 Entropy encoding

When talking about compression, entropy means information content of a message. Entropy measures how much information there is in a message with regard to the size of the message. It is defined as $-\log_2(P)$ where P is the probability of a given symbol. If a particular symbol occurs more often than the other symbols, then that particular symbol can be coded using less bits. Entropy coding is used to remove redundancy, both contextual (the letter Q is almost always followed by the letter U) and alphabetic (the letter E occurs far more often than any other letter), from messages.

Initially we wanted to use an entropy coder called range coder [Mar79], which is similar to an arithmetic coder or a huffman coder. Compared compression wise to the arithmetic coder the performance is slightly worse but compared computationally almost twice as fast. However, twice as fast is still pretty slow as the arithmetic coder is a complex and computationally intense algorithm.

The numerical results from Said and Pearlman's paper [SP96] regarding omitting the arithmetic coder to compress the output from the SPIHT algorithm resulted in a degradation of PSNR[19] by 0.3 to 0.6 dB. We chose to completely drop the entropy coding step from the video codec as the added complexity would only give a modest quality enhancement.

3.9 Fixed point arithmetic

On computers, calculations with floating point numbers are time consuming. Often when speed is more important than precision, fixed point arithmetic is used. For the sake of simplicity, the numbers in the examples are in decimal format. At the end we will apply the ideas to binary format. When using fixed point, we simply pretend that we have a fractional part. Normal integer arithmetic is actually a special case of fixed point arithmetic, that is: no fractional part at all. To convert a floating point number in radix, r, format to a fixed point integer with n digits for the fraction part, we simply multiply the floating point number with r^n. For example: to convert the number 1.01 to a fixed point integer with two decimals we multiply it with 100, which gives the fixed point number 101. To convert back to floating point format, we simply divide with the same number we used to convert to fixed point format.

[19] see chapter 3.10 for explanation of Peak to Signal Noise Ratio.

Addition and subtraction is done in the same way as usual. Here is an example with both floating point numbers and fixed point numbers where $n = 2$:

$$\begin{aligned} 1.01 + 1.01 &= 2.02 \\ 101 + 101 &= 202 \end{aligned}$$

Converting the fixed point number back to floating point by dividing the fixed point answer with 10^2 yields the same sum as the floating point equation. Multiplication and division, however, are done in a slightly different way. Consider the following multiplication:

$$\begin{aligned} 1.01 \times 1.01 &= 1.0201 \\ 101 \times 101 &= 10201 \end{aligned}$$

The digits in the fixed point equation are a factor of 100 too large. Consider the following division:

$$\begin{aligned} 40.96/1.28 &= 32.0 \\ 4096/128 &= 32 \end{aligned}$$

The digits in the fixed point equation are a factor of 100 too small. So: when multiplying, the result must be divided by r^n, and when dividing, the dividend must be multiplied by r^n.

Applying these ideas to binary numbers are easy. Instead of multiplying or dividing with r^n, the values are bit shifted n steps right respectively left. That gives n bits for the fraction part. Table 4 lists some of the most common operations and the code[20] required. Let q be the number of bits for the fraction part, a a normal integer, x and y fixed point integers with q bits for the fraction part.

Operation	Code
$x + a$	`x + (a << q)`
$x + y$	`x + y`
$x \times a$	`x * a`
$x \times y$	`(x * y) >> q`
x/a	`x / a`
x/y	`(x << q) / y`
$x \rightarrow a$	`a = x >> q`
$a \rightarrow x$	`x = a << q`

Table 4: Fixed point arithmetic. The `<<` and `>>` operators are the left and right bit shifts.

[20]The code examples are in the C programming language.

3.10 Peak Signal to Noise Ratio

Peak Signal to Noise Ratio is a measurement commonly used when comparing images between each other. The PSNR is calculated as

$$\mathrm{PSNR} = 10 \log_{10} \frac{255^2 N^2}{\sum_{i,j} \left(f(i,j) - \tilde{f}(i,j) \right)^2}.$$

Here the 255 comes from the maximum value we can have anywhere on one bitplane of the picture, multiplying it with N, wich is the size of a bitplane, in our case 128×128. The transformed and retransformed image elements are named $f(i,j)$ and they are of course compared to the original image elemnts $\tilde{f}(i,j)$. We then take the logarithm of this, to get a comparable number and multiply it by 10 to get within the range of about 10 to infinity, but generally it lies somewhere between 20 and 40. Based on our own observations we have noticed that a picture lying above 30 is of tolerable quality, and above 40 is close to flawless.

PSNR is commonly used to comparing different image compression formats to each other, but as a measure of quality it really is not very good. It does not take in consideration the human eye model for example, as well as contour preservation. We can have a blocky picture with better quality than a smoothed out one. Where the smooth picture really is visually much better to look at, especially when moving. But it is a very wide used measure and it is relatively easy to implement, just remember to not only work towards minimizing PSNR, visual quality is more important.

4 Implementation

This chapter will in general terms describe our implementation of the codec. The details of the various steps have already been described in previous chapters, and the reader should consult them when needed.

We have developed a video codec from scratch. It uses wavelets as it's base. Then we used the codec to make an application that can stream out video, in various bit rates, to a network. An application that can receive this video stream and display it on a computer screen was also developed. Basically it works like this: the streaming application reads raw RGB images from the hard drive, compresses them and transmits the compressed video stream to the client application. The client application receives this video stream, uncompresses it and displays the images on the screen. The codec can work in two different modes of operation: single and multiple frame mode. In single frame mode each raw image is compressed individually. Multiple frame mode works by compressing several raw images at once. The advantage of multiple frame mode is that the redundancy between consecutive frames is exploited to achieve higher compression ratios.

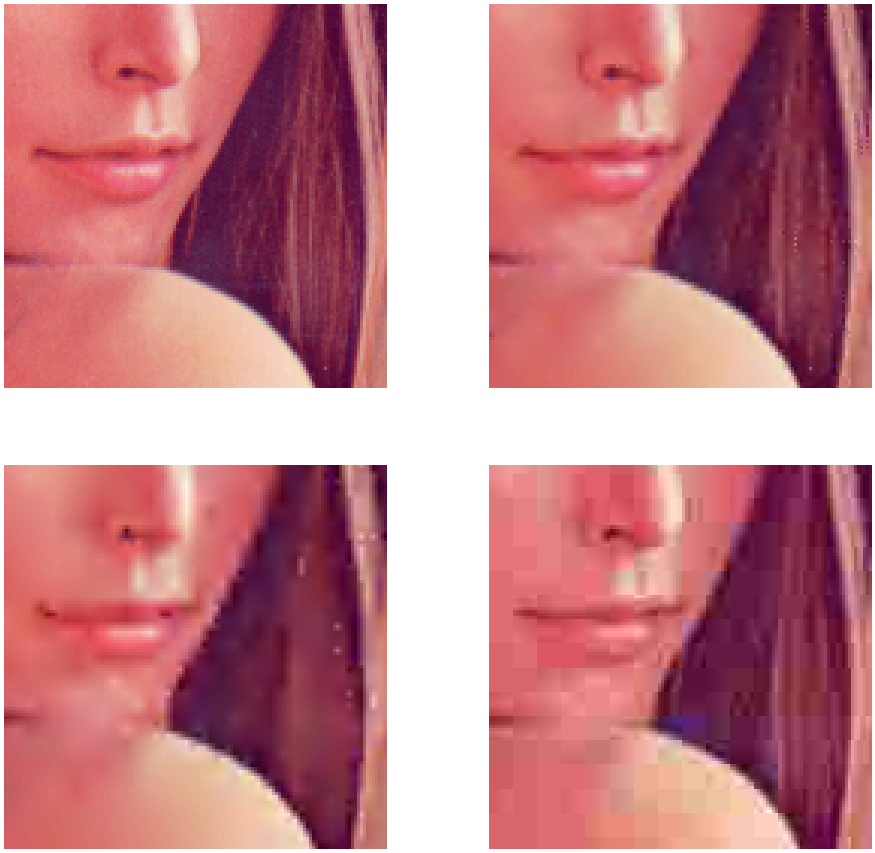


Figure 29: Here we see Lena's eye nose and mouth. The top left image is the original, followed on the right by a Neville wavelet compression to 0.7 bits per pixel. It has a PSNR value of about 30.87. We can see that this is not too far from the original. The picture to the lower left is compressed with Neville wavelets to 0.28 bits per pixel, and has a PSNR of 28.46. This can be compared to the JPEG compressed image at lower right, with a PSNR of 28.49. Here we see that PSNR does not take effects such as blockyness in it's consideration.

The compression ratio, wavelet transform and video clip can be changed, during runtime, on the client application. On the server side a thread is listening for requests that the client make regarding parameters, and encode the next video clip accordingly.

The video stream consists of packets of wavelet compressed data. The packet is made up of a small header that contains parameters for the various transforms, and the actual data separated in three parts (see figure (30)). Since the raw RGB image goes through the color space transform (that outputs image data in planar format) the data must be splitted into three parts.

Every transform can handle both single and multiple frame modes, so the packets look the same for both formats. Of course, the multiple frame mode packets are larger, but on the other hand they contain information for

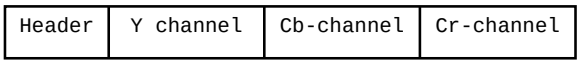


Figure 30: Wavelet video packet

several frames.

Let us take a closer look at the server side of the implementation. The server is made up of three main threads of execution. One thread for reading raw RGB frames from disk, the second for compressing the frames and the last thread takes care of transmitting the compressed data to the client. Data structures are passed between the threads. The data structure contains the raw data, the compressed data and various parameters for the different transforms.

The thread that reads raw frames from disk, read either one frame at a time or several depending on the frame mode. It then calls the color space transform on the raw data. When the transform have completed, the data is put in the wavelet data structure, and is inserted to a fifo[21] queue and the thread starts all over. Each thread have a fifo queue from where it receives the next data structure. When a thread is done processing data, the data structure is inserted into the next threads fifo.

Now the thread that compresses the newly color transformed data, takes the data structure from it's input fifo. The first thing we do is to convert the data to fixed point integers, in order to lessen the impact of roundoff errors. The image data will now be wavelet transformed, sorted and encoded by the SPIHT algorithm before the structure is inserted to the next threads queue.

The network thread first takes the data structure from it's input queue, then sends parameters that describe how the data have been compressed to the client. And finally the compressed data is on it's way to the client.

The client side works in a similar manner, only that everything is inverted. The first thread is the network thread, the second the decompressor thread and the last the screen[22] thread. In every thread, each transform's inverse version is applied in the reverse order of which they were applied on the server side. The client side is built, just like the server side, with threads, fifo queues and data structures. One thing that the client have, but not the server, is a GUI[23]. In this GUI the user can choose which wavelet transform to use, what clip to view and at what compression ratio (see figure (31) for a screen shot).

The client side was also ported to run on an iPAQ. iPAQ is a hand held computer that can be outfitted with wireless network extensions. No part of

[21] first in, first out

[22] On the client side, the raw RGB frames are not stored on disk, instead they are displayed on screen.

[23] Graphical User Interface



Figure 31: Screen shot of GUI

the source code needed to be rewritten in order for the application to run, only recompiled. Since the iPAQ uses another architecture we had to install a cross compiler that could generate executables in the correct format.

There exists limitations in what the codec can handle as far as dimensions go. In order to simplify implementation of the various transforms, we limited the codec to only handle frames of quadratic size, and multiple frames of cubic size.

The codec was written and implemented in the C programming language on a GNU/Linux system. The source code should compile on any POSIX compliant system with a ANSI C compiler. Since the GUI parts use the GTK+[24] toolkit, the gtk libraries are required for the client side.

5 Results

Figure (32) shows a comparison between JPEG and our codec in single frame mode. The graph shows that JPEG is superior to our wavelet coder, except for very low bit rates. JPEG's decline probably comes from the fact that the image is divided into 8x8 pixels large blocks before the DCT[25] is applied, and the overhead becomes significant at low bit rates. When comparing the Neville filter against the Haar filter, the difference in PSNR is roughly 2 dB.

[24]GTK stands for "GIMP Tool Kit", and GIMP stands for "GNU Image Manipulation Program", and GNU stands for "GNU's not UNIX". Programmers love acronyms, especially recursive acronyms.

[25]Discrete cosine transform, as used in JPEG and MPEG.

Comparing the JPEG against the Neville filter we end up just 1 dB lower in PSNR.

The implementation was done on an iPAQ with a resolution of 128×128 pixels in 24 bit true color. With some 30 times compression we were able to achieve a frame rate of about 7. This is not very good, but much can be blamed on the fact that the routines we were using are not optimized, and the modest hardware that the iPAQ consists of. The SPIHT might have to be put aside and replaced by something much faster, as well as speeding up the wavelet transform.

Figure (33) shows a comparison between MPEG-1 and our codec in multiple frame mode. The graph shows PSNR for the 128 first frames of the "foreman" video (which was cropped from its original size of 176×144 to 128×128). The PSNR for MPEG-1 is roughly 4 dB better than our encoder.

This was implemented on a 1.4 GHz Pentium 4 with 256 MB of 800 MHz main memory. The memory requirements for multiple frame mode was too large for the iPAQ to handle, so we could not try it out there. But running the server and application on the same machine was no problem at all, even when we also ran two single frame compression threads for the iPAQs. The frame rate was a stable 30 fps, even when we tried running the frame by frame codec. The frame by frame codec is actually slower than the full cubic one, since it can be compressed better, and thus the SPIHT algorithm works faster. Of course, this was still only 128 frames per cube in a resolution of 128×128. We did not try to do any scaling of the video either.

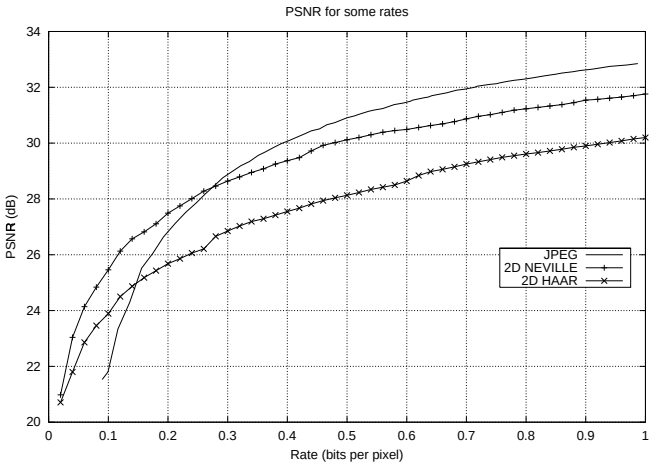


Figure 32: This is a graph over the different PSNRs for the "Lena" picture. Here we see how much our Neville filter surpasses the Haar filter when it comes to PSNR. An even more complex filter would give JPEG a run for it's money, at least at very low bit rates.

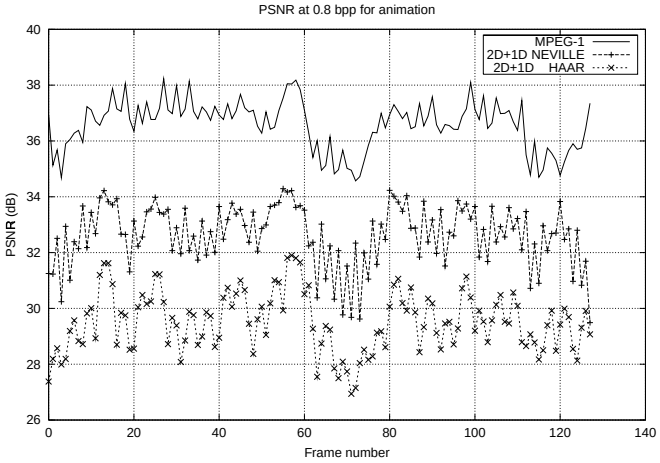


Figure 33: Here we see the different PSNRs for the "Foreman" video clip. This is a comparison between MPEG-1 and our wavelet filters. We can see how much PSNR is gained just by going from Haar to Neville, but they are both still beaten by MPEG-1 PSNR wise.

5.1 Notes

To create the MPEG-1 videos from raw RGB frames we used a program called "mpeg2encode" by the MPEG software simulation group. Default parameters were used for all options, except for the width and height. When processing the JPEG pictures we used the ImageMagick program suite.

The "Lena" picture that we used in the comparison is a classical picture that is often used in image processing. The "foreman" clip that is used in the video comparison graph is a semi-classical video clip used in image processing. The server where we found these images and clips are no longer operational, but they can be found at other places all over the Internet.

6 Conclusions

Lifting is indeed a fast and efficient way of doing wavelet transform. But two dimensional lifting easily gets too cumbersome. The filters swell out in all dimensions and you end up doing a lot of lifting steps, even if a pass does both the x and y dimensions at the same time. The amount of calculations are more than for the double one dimensional transform, but if you get cache hits this might improve matters. Lifting is indeed very useful, but two dimensional lifting easily gets too cumbersome in comparison to what it is worth. It is also much more advanced technically, since you have to take all the fringe effects into consideration. With a one dimensional filter you only have the beginning and the end of the sequence, with two dimensional

lifting you suddenly have two more edges that you have to bother about.

As we can see from our results, a bigger filter is necessary. It would be able to cope with more advanced polynomials, and thus make better approximations. An entropy coder would improve things further, but not by very much. This also takes a lot of processing power, which sometimes is a bigger hindrance than actual bandwidth issues. The motion compensation techniques used in MPEG formats may be of some help compressing the stream and increasing the quality, but then the scalability is harder to achieve.

7 Further development

If you read through chapter 8, the history chapter, you will note that the mathematical theory of wavelets is pretty young, and the implementations of it has just begun. Methods like MPEG-4 and H.263 still gives better compression, but then they are almost fully optimized. Complexity wise walelets should go faster than Fourier transform. This is not really true yet, since the FFT[26] is so optimized programming wise, but this is just something that has to be further developed.

7.1 Resolution

It is important in a video transform to be able to resize the window as you see fit. Downscaling a resolution is generally done by simply removing pixelrows in between, scaling down to two thirds of the original image means every third row and column is to be removed. Wavelets do have the ability that if you scale down to half size, the transform will actually go faster, you just do not have to add the last steps in the resolution scaling. Upscaling works fine as well, since wavelets do make polynomial interpolations between pixels, so if we scale something up hugely, we will not have blocks, the picture will just become more and more unfocused. You can see this in figure (15), on page 37. Actually as can be seen in the figure, some details may even be expressed better, of course up to a limit, we can not come up with information we do not have.

7.2 Uneven scaling

Quite often, the input we get to our codec is not in square shape, but rather oblong, like for example a movie. Since the eyes are spaced apart a bit this is preferable for good visualisation. In 2.7.1 we get into depths with how this is achived, but basically it consists of doing the transform in a limited amount of dimensions at first, adding the others when it is feasible. Sometimes it

[26]Fast Fourier Transform

would be very useful with a varying time dependence as well. Ideally the time depth should match the scenes. So that each scene change ends up on a new cube, but this is very hard to do. The thing is that a high resolution picture, with a lot of timeframes will require massive amounts of memory, it also means that you will have an unwilling delay, at least three times as long as the timeframes. Since you have to read one, code one and send one.

7.3 Timing

Sometimes the network is just fine, it is the processor that is too slow. The execution time of the SPIHT algorithm is highly dependable on the amount of information, so it might be necessary to cut it short, just settling with a few terms. In the same way the high amount of zeroes that would collect in the wavelet transform should be used to decrease the calculations possible. All to make the codec less platform dependant, able to adapt to both processor and connection speed.

7.4 Optionality

It should also be possible to choose if you want high or low resolution, or many or few frames per second. Maybe on the same stream. If you do not have a screen with full resolution, there is no point in getting the extra high resolution information for example. A simple way of doing this is to split the signal up on different ports on the machine. One port then sends out the most basic information, the scaling coefficients. Then we can have other ports open, sending the resolution information, and yet others sending the resolution in framerate. This way the ports you connect to govern wich preferences you want. Another solution to this is to implement a different network protocol than TCP/IP, a protocol that does not guarantee the transmission of all packets, but rather just the important ones. Dumping the ones that cannot get through anyway. This would be useful for multicasting purposes.

8 History

The history of wavelets is not very long, at least as far as a mathematical theory goes. It's roots can be traced back to the beginning of the 18:th century, when Joseph Fourier proposed that a periodic function could be approximated with series of trigonometric functions. It did not even have to be continious. The idea did not pass unchallenge amongst other mathematicians though. Laplace was one of the most well known advocates against Fourier transform, but the theory has been worked over, and now it is hard to doubt it's versatility. Fourier transform shows up in many fields of both

applied and unapplied mathematics, solutions for differential quations, image compression and just about anything really.

8.1 Fourier transform

As stated previously Fourier transform is an orthogonal transform with trigonometric base functions. The basis looks like

$$\cos\left(\frac{2\pi nx}{l}\right),\ \{n|\, n \geq 0,\, n \in \mathbb{Z}\},$$

$$\sin\left(\frac{2\pi nx}{l}\right),\ \{n|\, n > 0,\, n \in \mathbb{Z}\},$$

where l is the length of a period of the function, and x is the variable of the function. The first base function $\cos(0)$ is really just a flat one, working the same way as our scaling function. For all other n we will get mutually orthogonal functions, that is an orthonormal basis. Observe that the basis is not orthonormal though, since

$$\int_0^l \sin^2\left(\frac{2\pi nx}{l}\right)\mathrm{d}x = \left[\frac{x}{2} - \frac{\sin\left(\frac{4\pi nx}{l}\right)}{\frac{8\pi nx}{l}}\right]_0^l = \frac{l}{2},$$

wich is dependent on the length of the function. This is not a drawback though, since the interval length of one period of the function is constant. In reality this is the same thing as having a basis wich are all scaled to the different intervals. It means we can use the same coefficients no matter wich length l we have of the period. To describe a function $f(x)$ we naturally use the spectral theorem, see page 8, to get

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left(a_n \cos\left(\frac{2\pi nx}{l}\right) + b_n \sin\left(\frac{2\pi nx}{l}\right)\right)$$

where

$$a_0 = 2\,,\ a_n = \frac{2}{l}\int_0^l f(x)\cos\left(\frac{2\pi nx}{l}\right)\mathrm{d}x \text{ and } b_n = \frac{2}{l}\int_0^l f(x)\sin\left(\frac{2\pi nx}{l}\right)\mathrm{d}x,$$

for $n > 0$. Here we see the $\frac{2}{l}$ terms, used to normalize the equations. All this gives us a basis for approximating most limited energy functions[27] arbitary well. Discontinuities are handled, albeit not perfectly, and Fourier transforms have a hard time handling very fractal and discontinous functions.

[27] All functions with $\int f^2(x)\mathrm{d}x < \infty$.

8.2 Early wavelets

It was actually this shortcoming that lead to the development of wavelets. Geophyscisits needed something to describe the short interval and very strong movements of the earth during tremours. To describe this fractal and short lived process with an infinite trigonometric series is not feasible, something new was needed. It was now that the reseach of one Alfrad Haar come in focus. In 1909 he proposed his Haar wavelet wich we have already worked with. Now this is not very good for describing tremours either, but the idea inspired some other mathematicians to try and find better orthonormal bases for functions. Since trigonometric functions are very smooth, and Haar functions are discontinous, something in between was needed, to describe those not so smooth but still continous functions. In 1920 Faber and Shauder tried taking the primitive function of Haar function[28], the pyramid function.

$$\Delta(x) = \begin{cases} 2x & : \quad 0 \leq x < \frac{1}{2} \\ 2 - 2x & : \quad \frac{1}{2} \leq x < 1 \\ 0 & : \quad x \notin [0, 1) \end{cases}.$$

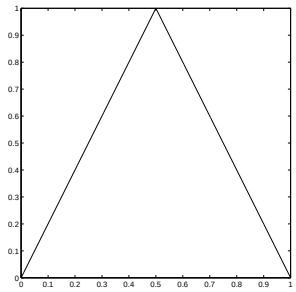


Figure 34: The primitive function of Haar's orthogonal basis. Calculated by Faber and Shauder.

But this function had the drawback of not being orthonormal for all dilations of it. Seven years later Philp Franklin succeeded in creating an orthonormal basis, by using the Gram-Schmidt method to orthogonalize the pyramid base. But it became to cumbersome to use.

8.3 The birth of wavelets

It took a long while before anything new was tried, but the ideas of a compactly supported orthogonal basis lived still. It was in 1980 that Grossman and Morlet started trying to use Fourier transforms of functions to define them in frequency space at first. This then lead to wavelets with compact support in both frequency as well as time space, and the behaviour we see with wavelets today. It was now that the word wavelet, meaning small wave as they have a compact support, was initiated. Differing them from big waves, like the trigonometric bases.

[28] Because the term wavelet was not used yet.

8.4 Modern wavelets

With the tools of frequency space analysis the research gained new momentum, and soon J. O. Strömberg created the first smooth wavelet basis. He is still a researcher with a good reputation in the field of wavelets. But the one closing the first inital chapter of wavelets is Ingrid Daubechies originally from the Netherlands, now working in America. In 1987 she designed a set or rather a method to create continous wavelets of different orders, with a predetermined set of vanishing moments. The inital wavelet being Haar's. She also helped developing the lifting methods for wavelets together with Wim Sweldens. Lifting was a method that had been used in signal processing, to get new kinds of filters, it was now shown that many wavelets was equvivalent to certain lifting formulas, meaning you could take use of the faster and more efficients calculations of lifting when doing wavelet transforms.

8.5 Future

The future of wavelets looks bright, firstly developed as an analysis tool it has many areas of interest there, but it is quickly conquering many fields where Fourier transform was the only tool to choose. Similarities between the two transforms makes them excelent counterparts in such widespread areas as interpollation, smoothing, noise reduction, differential equations, speech analysis and of course video compression. New articles are published monthly, and many more wavelets are still to be found and implemented. So there is much work to be done.

References

[BGG98] C. Burrus, R. Gopinath, and H. Guo. Introduction to wavelets and wavelet transforms: A primer, 1998. A very good book, presenting the theory in an interesting way. It contains a lot of matlab examples as well, for anyone interested in immediately starting to experiment with wavelets.

[CDSY98] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. Wavelet transforms that map integers to integers. *Applied and Computational Harmonic Analysis (ACHA)*, 5(3):332–369, 1998.

[Dau92] Ingrid Daubechies. *Ten Lectures on Wavelets*, volume 61. Society for Industrial and Applied Mathematics, Philadelphia, 1992. The fundamental book of applied wavelet transform. A must read for anyone interested in the working fundamentals of wavelets.

[DS96] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. Technical report, 1996. This is the original report on using wavelet lifting. A little dense theory, but it provel all the why and how to make filter banks into lifting wavelets. Numerous examples makes it worthwhile for anyone who wants to experiment with this.

[KS92] Jelena Kovacević and Wim Sweldens. Nonseparable two- and three-dimensional wavelets. Technical report, 1992. Mostly just a presentation of the concept.

[KS97] J. Kovacevic and W. Swelden. Wavelet families of increasing order in arbitrary dimensions, 1997. Here the Neville filters are presented as well as a scheme to make our own n-dimensional Neville filters. It contains lot of examples of one, two and three dimensional filters.

[KXP97] B. Kim, Z. Xiong, and W. Pearlman. Very low bit-rate embedded video coding with 3d set partitioning in hierarchical trees, 1997.

[KXP00] B. Kim, Z. Xiong, and W. Pearlman. Low bit-rate scalable video coding with 3d set partitioning in hierarchical trees, 2000.

[Mar79] G.N.N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. Technical report, IBM UK Scientific Center, 1979.

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

[SK92] I. Shah and A. Kalker. Theory and design of multidimensional qmf sub-band filters from 1-d filters using transforms, 1992. How to make a one dimensional filter into a multi dimensional one.

[SP96] Amir Said and William A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6:243–250, 1996.

A Wavelets

A.1 Haar filterbanks

The h^* and g^* of Haar appears as

$$h^* = 1 + z^{-1},\ g^* = 1 + \frac{z^{-1}}{2},$$

and it naturally follows that

$$h = 1 + z,\ g = 1 + \frac{z}{2}.$$

A.2 Haar lifting

The Haar one dimensional lifting filters look like

$$p = \begin{array}{ccc} 0 & \underline{0} & \frac{1}{2} \end{array} \ and\ u = \begin{array}{ccc} 1 & \underline{0} & 0 \end{array}.$$

This gives us the equations

$$\begin{aligned} d &= d^0 - s^0, \\ s &= s^0 + \frac{d^0 - s^0}{2} = \frac{s^0 + d^0}{2}, \end{aligned}$$

where we can see that s is the average of ^{0} and d^0, and d is the difference between d^0 and s^0.

If we then look at the two dimensional lifting of Haar we will get

$$p_x = \begin{array}{ccc} 0 & 0 & 0 \\ 1 & \underline{0} & 0 \\ 0 & 0 & 0 \end{array},\ p_y^x = \begin{array}{ccc} 1 & 0 & 0 \\ 0 & \underline{0} & 0 \\ 0 & 0 & 0 \end{array},\ p_y = \begin{array}{ccc} 0 & 1 & 0 \\ 0 & \underline{0} & 0 \\ 0 & 0 & 0 \end{array}\ and\ u = \begin{array}{ccc} 0 & 0 & 0 \\ 0 & \underline{0} & \frac{1}{4} \\ 0 & \frac{1}{4} & \frac{1}{4} \end{array}.$$

Leading to the equations

$$\begin{aligned} d_x &= d_y^0 - s^0, \\ d_y^x &= d_y^{x0} - s^0, \\ d_y &= d_x^0 - s^0, \\ s &= s^0 + \frac{d_x^0 + d_y^{x0} + d_z^0 - 3s^0}{4} = \frac{s^0 + d_x^0 + d_y^{x0} + d_z^0}{4}. \end{aligned}$$

The Haar lifting has no border effects, unless you have an uneven width or height of the two dimensional lattice you wish to transform.

A.3 Neville lifting

All Neville filters and how they are calculated are presented in [KS97], we will only describe the two we have been using. The first one is actually the Haar wavelet seen above, the second one is a four tap two dimensional filter whose lifting looks like

$$p_x = \begin{matrix} 0 & 0 & 0 \\ \frac{1}{2} & \underline{0} & \frac{1}{2} \\ 0 & 0 & 0 \end{matrix}, \; p_y = \begin{matrix} 0 & \frac{1}{2} & 0 \\ 0 & \underline{0} & 0 \\ 0 & \frac{1}{2} & 0 \end{matrix}, \; p_y^x = \begin{matrix} \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & \underline{0} & 0 \\ \frac{1}{4} & 0 & \frac{1}{4} \end{matrix} \; and \; u = \begin{matrix} \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{4} & \underline{0} & \frac{1}{4} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \end{matrix}.$$

Here the equations look like

$$\begin{aligned} d_x &= d^0 - p_x * s^0, \\ d_y^x &= d^0 - p_y * s^0, \\ d_y &= d^0 - p_z * s^0, \\ s &= s^0 + u * d, \end{aligned}$$

where d^0 means all of d_x, d_y^x and d_y taken together. More examples of one dimensional lifting can be found in [DS96].

B Matlab code

B.1 Wavelet transform

Here follows a simple program for a two dimensional wavelet transform. It is written in Matlab but the code should be simple enough to follow to port to for example C.

```
function [ w ] = twodtrans( p,h0,h1 )
% This function transforms an ordinary two dimensional function p
% with the help of the wavelet filters h0 and h1, as you get from
% the script wavelet.m

[l,m]=size(p);

lh=floor(l/2);
mh=floor(m/2);

w=zeros(l,m);

% for i=1:l
%     [lx,hx]=f2w(p(i,:),h0,h1);
%     w(i,1:mh)=lx;
%     w(i,mh+1:2*mh)=hx;
% end
% for i=1:m
%     [ly,hy]=f2w(w(:,i),h0,h1);
%     w(1:lh,i)=ly';
%     w(lh+1:2*lh,i)=hy';
% end

for i=1:l
    [lx,hx]=f2w(p(i,:),h0,h1);
    w(i,1:mh)=lx;
    w(i,mh+1:2*mh)=hx;
end
for i=1:m
    [ly,hy]=f2w(w(:,i),h0,h1);
    w(1:lh,i)=ly';
    w(lh+1:2*lh,i)=hy';
end
```

B.1.1 Function to wavelet space transform

This calls on the function below, which takes us from ordinary function space, to wavelet space via a lifting transform.

```
function [ s,d ] = f2w( f,h0,h1 )
%F2W Transform a function from function space to wavelet space.
%  F2W(F,H0,H1) transform the function F from function space
%  to a wavelet space with the low pass filter H0 and the
%  high pass filter H1.
```

```
%
%  Remember, H0=G0*2
%  Remember, H1=G1*2
%
%  See also, W2F                                                    10

l=length(f);

% Filtering

o=zeros(l,1);
p=zeros(l,1);
for i=1:length(f);
    for j=1:length(h0)
        k=mod(i+j-1,l)+1;                                           20
        o(k)=f(i)*h0(j)+o(k);
        p(k)=f(i)*h1(j)+p(k);
    end
end

% Downsample

length(f)/2;

for n=1:floor(length(f)/2)                                          30
    s(n)=o(2*n);
    d(n)=p(2*n);
end
```

B.2 Different wavelets in matlab

It requires that you have at least defined some kind of filter wavelet. Here are a couple examples of some. This have to be run with at least some value for t or you will have to define $h0$, $h1$, $g0$ and $g1$ by hand.

```
% Acceptable settings for 't'
%
% 'haar' - Haar wvavelet
% 'daub' - Daubechie's 4 long wavelet
% 'resn' - Resnikoff's wavelet with angles a and b

if t=='haar'
    h=[1 1]/sqrt(2);
elseif t=='daub'
    h=[0.68301 1.18301 0.31699 -0.18301]/sqrt(2);                   10
elseif t=='resn'
    a=pi/4;
    b=pi/2;
    if a==b
        h=[1 1]/sqrt(2);
    elseif b==0
        h0=(1-cos(a)+sin(a))/2;
```

```
        h1=(1+cos(a)+sin(a))/2;
        h2=(1+cos(a)−sin(a))/2;
        h3=(1−cos(a)−sin(a))/2;
        h=[h0 h1 h2 h3]/sqrt(2);
    else
        h0=((1+cos(a)+sin(a))*(1−cos(b)−sin(b))+2*sin(b)*cos(a))/4;
        h1=((1−cos(a)+sin(a))*(1+cos(b)−sin(b))−2*sin(b)*cos(a))/4;
        h2=(1+cos(a−b)+sin(a−b))/2;
        h3=(1+cos(a−b)−sin(a−b))/2;
        h4=(1−h0−h2);
        h5=(1−h1−h3);
        h = [h0 h1 h2 h3 h4 h5]/sqrt(2);
    end
elseif t=='dauX'
    N2=8;
    a=1;
    p=1;
    q=1;
    h=[1 1];
    for j=1:N2−1,
        h=conv(h,[1,1]);
        a=−a*0.25*(j+N2−1)/j;
        p=conv(p,[1,−2,1]);
        q=[0 q 0]+a*p;
    end
    q=sort(roots(q));
    h=conv(h,real(poly(q(1:N2−1))));
    h=h*sqrt(2)/(sum(h));
else
    disp('Not a known wavelet!')
end

k=(1:length(h))−1;
g0=h;
g1=fliplr(g0).*(−1).^k;
h0=fliplr(g0);
h1=fliplr(g1);
```

B.3 Inverse wavelet transform

Then we need a reverse transform, this is a program similar to the first.

```
function [ p ] = twodinvtrans( w,g0,g1 )
% This function inversely transforms an ordinary two dimensional
% wavelet function p with the help of the inverse wavelet filters
% h0 and h1, as you get from the script wavelet.m

[l,m]=size(w);

lh=floor(l/2);
mh=floor(m/2);
```

```
p=zeros(l,m);

% for i=1:l;
%     p(i,:)=w2f(w(i,1:lh),w(i,lh+1:l),g0,g1)';
% end
% for i=1:l;
%     p(:,i)=w2f(p(1:lh,i),p(lh+1:l,i),g0,g1);
% end

for i=1:l;                                                    20
    p(i,:)=w2f(w(i,1:mh),w(i,mh+1:m),g0,g1)';
end
for i=1:m;
    p(:,i)=w2f(p(1:lh,i),p(lh+1:l,i),g0,g1);
end
```

B.3.1 Wavelet to function transform

Which calls on the inverse function of the one above, namely this.

```
function [ f ] = w2f( s,d,g0,g1 )
%W2F Transform a function from wavelet space to function space.
%  F2W(S,D,H0,H1) transform the scalinig coefficinats S and the
%  wavelet coefficients D from function space to a wavelet space
%  with the low pass filter H0 and the high pass filter H1.
%
%  Remember, G0=H0/2
%  Remember, G1=H1/2
%
%  See also, F2W                                              10

l=length(s)*2;

% Upsampling

o=zeros(l,1);
for n=1:length(s)
    o(2*n)=s(n);
end
                                                              20
p=zeros(l,1);
for n=1:length(d)
    p(2*n)=d(n);
end

f=zeros(l,1);

% Filtering

for i=1:length(f);                                            30
    for j=1:length(g0);
```

```
        k=mod(i+j-2-length(g0),l)+1;
        f(k)=o(i)*g0(j)+p(i)*g1(j)+f(k);
    end
end
```

C Pseudo code

All pseudo code is written in a C like language. The code examples are for helping in understanding some of the algorithms used.

C.1 SPIHT

```
/* initialize */
init() {

  threshold = highest_bit_of_largest_coeff;

  list_add(sign_list, 0);
  list_add(sign_list, 1);
  list_add(sign_list, 2);
  list_add(sign_list, 3);

  list_add(desc_list, 1, NORMAL);
  list_add(desc_list, 2, NORMAL);
  list_add(desc_list, 3, NORMAL);
}

/* refinement pass */
ref_pass() {

  for( all_nodes_in_reflist_except_recently_added ) {
    if( (current_node.value & threshold) == 0 )
      output("0");
    else
      output("1");
  }
}

/* significance pass */
sign_pass() {

  for( all_nodes_in_signlist ) {

    /* if the coeff. is significant */
    if( abs(current_node.value) >= threshold ) {
      output("1");

      /* the sign of the coeff. */
      if(current_node.value > 0)
        output("0");
      else
        output("1");

      list_add(ref_list, current_node);
      list_del(sign_list, current_node);

    } else {
```

```
      output("0");
    }

  }
}

/* descendants pass */
desc_pass() {

  for( all_nodes_in_desclist ) {

    if(current_node.type == MULTINODE) {
      /* multinode */
      if( any_child_has_significant_descendants ) {
        output("1");
        list_add(desc_list, current_node.child1, NORMAL);
        list_add(desc_list, current_node.child2, NORMAL);
        list_add(desc_list, current_node.child3, NORMAL);
        list_add(desc_list, current_node.child4, NORMAL);
        list_del(desc_list, current_node);
      } else {
        output("0");
      }

    } else {
      /* normal node */
      if( current_node_has_significant_descendants ) {
        output("1");
        tmp_list = current_node.all_children;
        sign_pass(tmp_list);
        list_add(sign_list, tmp_list);
        list_del(desc_list, current_node);

        if( current_node_have_grandchildren )
          list_add(desc_list, current_node, MULTI);

      } else {
        /* no significant descendants */
        output("0");
      }
    }
  }
}
```

C.2 Wavelet sort algorithm

This recursive function sorts a wavelet transformed image that have coefficients ordered like the right part of figure (26) into order (just like if one had followed the Morton scan order).

```
/* recursive function that sorts a wavelet transformed image */
step(int index, int pos, int step, int diff) {

  if(diff > 1) {
    /* need to go deeper into the tree */
    step(index,              pos+0*diff, step/2, diff/4);
    step(index+step,         pos+1*diff, step/2, diff/4);
    step(index+step*w,       pos+2*diff, step/2, diff/4);
    step(index+step*w+step, pos+3*diff, step/2, diff/4);
  } else {
    /* we have a position and a corresponding index */
    dest[pos+0] = src[index];
    dest[pos+1] = src[index + step];
    dest[pos+2] = src[index + step*w];
    dest[pos+3] = src[index + step*w + step];
  }

}

/* the main function that calls the above function
 * for each level.
 */
convert_from_wavelet() {

  /* initialize */
  levels = log_2(width);
  index = width/4
  pos   = 4;
  step  = w/2;
  diff  = 1;

  /* special case for level 0 */
  step(0, 0, width/2, 1);

  for( each_level_except_zero ) {
    step(index,          pos*1, step, diff); /* for the M-shape */
    step(index*w,        pos*2, step, diff); /* E-shape */
    step(index*w+index, pos*3, step, diff); /* X-shape */

    index = index/2;
    pos   = pos*4;
    diff  = diff*4;

  }
}
```