Introduction to Wavelets v1.7

© N. F. Chamberlain

Professor of Electrical Engineering
South Dakota School of Mines and Technology
December 2002

Neil.Chamberlain@sdsmt.edu

Table of Contents

Introduction to First Version 3
Continuous Wavelet Transform 7
Haar Wavelet 7
Analysis Using the Haar Wavelet 10
Interpretation of CWT 10
The Discrete Wavelet Transform (Fast Wavelet Transform) 14
The Haar Wavelet as a Digital Filter. 15
Doing the DWT 'By Hand' 18
Reconstruction 19
Application of DWT 24
Multiple Decompositions 25
Daubechies Wavelets of General Order. 29
Synthesizing Half-Band Frequency Responses. 33
Deriving the Wavelet from the Scaling Function. 37
Zero Padding of Wavelet and Scaling Functions 37
Iterating the Wavelet and Scaling Functions 39
Reconstruction Using the Matrix Approach to Wavelets. 42
Upsampling and Downsampling. 45
Frequency Domain Consequences of Subsampling 46
The Dilation and Wavelet Equation 51
Two Dimensional Wavelet Processing 55
Dealing with edges, and odd numbers of data points. 60
Using the MATLAB Interface to do 2D Processing 63
Wave Packet Analysis 68
A Further Note on the Use of Shannon Entropy 78
Biorthogonal Wavelets 85
Biorthogonal Wavelets from Lifting 90
Formulation of the JPEG 2000 9/7 Filter. 96
DWT by Lifting 99

Introduction to First Version

These notes were written to accompany an Electrical Engineering graduate-level course in advanced digital signal processing (DSP) focusing on wavelet signal processing. The notes are intended for engineers or people who are interested in applying wavelets, rather than understanding their deeper mathematical foundations. My objective in the advanced DSP course was to do real-time signal processing using wavelets. I used a combination of MATLAB and C code implemented on an Analog Devices SHARC DSP to do this. Part I of the notes discusses background theory and makes heavy use of MATLAB, and occasional use of the MATLAB Wavelet Toolbox. If you don't have MATLAB, then I recommend you buy the student version. Most of the MATLAB work can be done without the special functions in the Wavelet Toolbox. Next time I revise these notes, I will do a better job of developing the material without use of the Wavelet Toolbox. Part II discusses real-time applications.

Topic-wise, I focus on the discrete wavelet transform (DWT), and the wavelets devised by Ingrid Daubechies (both orthogonal and bi-orthogonal). The Continuous wavelet transform (CWT) is discussed, but not in a lot of detail; the interesting applications are in the DWT.

Background.
Mathematically, you will be lost if you don't as least have a passing acquaintance with the z-transform, Fourier transform, and discrete time convolution. A junior level course in "Signals" and a senior level course in DSP are preferred, and these are prerequisites for the advanced DSP course I teach.

Disclaimer:
This is the first semester that I have taught this course, so they will appear a little disorganized, and no doubt incomplete.

Fourier Analysis and Orthogonality

Fourier analysis is one of the most widely used tools in spectral analysis. The basis for this analysis is the Fourier Integral, which computes the amplitude spectral density $F(\omega)$ of a time-domain signal $f(t)$.

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\exp(-j\omega t)\,dt$$

$F(\omega)$ is actually complex, so one obtains the amplitude spectral density $A(f)$and phase spectral density $\varphi(f)$ as a function of frequency. Another way of looking at the Fourier transform is that it answers the question: what continuous distribution of sinewaves $A(f)\cos(j\omega t+\varphi(f))$ when added together on a continuous basis best represents the original time signal? We call these distributions the amplitude and phase spectral densities (or spectra).

Complex exponentials are popular basis functions because in many engineering and science problems, the relevant signals are sinusoidal in nature. You'll notice that when signals are not sinusoidal in nature, a wide spectrum of the basis function is needed in order to represent the time signal accurately.

An important property of any family of basis functions $\psi(t)$ is that it is orthogonal. The basis functions in the Fourier Transform are $\psi(t) = \exp(+j\omega t)$, so the Fourier Transform could be more generally written as

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\psi^{*}(t)\,dt$$

where * denotes complex conjugate. The test for orthogonality is done as follows

$$\int_{-\infty}^{\infty} \psi_m(t)\psi_n^{*}(t)\,dt = \begin{cases} k & m = n \\ 0 & m \neq n \end{cases}$$

For complex exponentials, because they are infinite in duration, you end up with $k=\infty$, when $m=n$ so it is necessary to define the orthogonality test in a different way

$$\lim_{T\to\infty} \frac{1}{T}\int_{-T/2}^{T/2} \psi_m(t)\psi_n^{*}(t)\,dt = \begin{cases} k & m = n \\ 0 & m \neq n \end{cases}$$

For complex exponential, this becomes

$$\lim_{T\to\infty} \frac{1}{T}\int_{-T/2}^{T/2} \exp(j\omega_m t)\exp(-j\omega_n t)\,dt = \lim_{T\to\infty} \frac{\sin(T[\omega_m-\omega_n]/2)}{T[\omega_m-\omega_n]/2} = \begin{cases} 1 & m = n \\ 0 & m \neq n \end{cases}$$

When the constant $k=1$, the function is said to be orthonormal.

Various types of signals can be analyzed with the Fourier Transform. If f(t) is periodic, then the amplitude spectral density clusters at discrete frequencies that are harmonics (integer multiples) of the fundamental frequency. You need to invoke Dirac Delta functions if you are going to use the Fourier Transform – otherwise you can compute the Fourier series coefficients and get essentially the same result. If f(t) is deterministic and discrete, the discrete time Fourier Transform (DTFT) may be used to generate a periodic frequency response. If f(t) is assumed to be both periodic and discrete, then the discrete Fourier Transform (DFT), or its fast numeric equivalent the FFT may be applied to compute the spectrum.

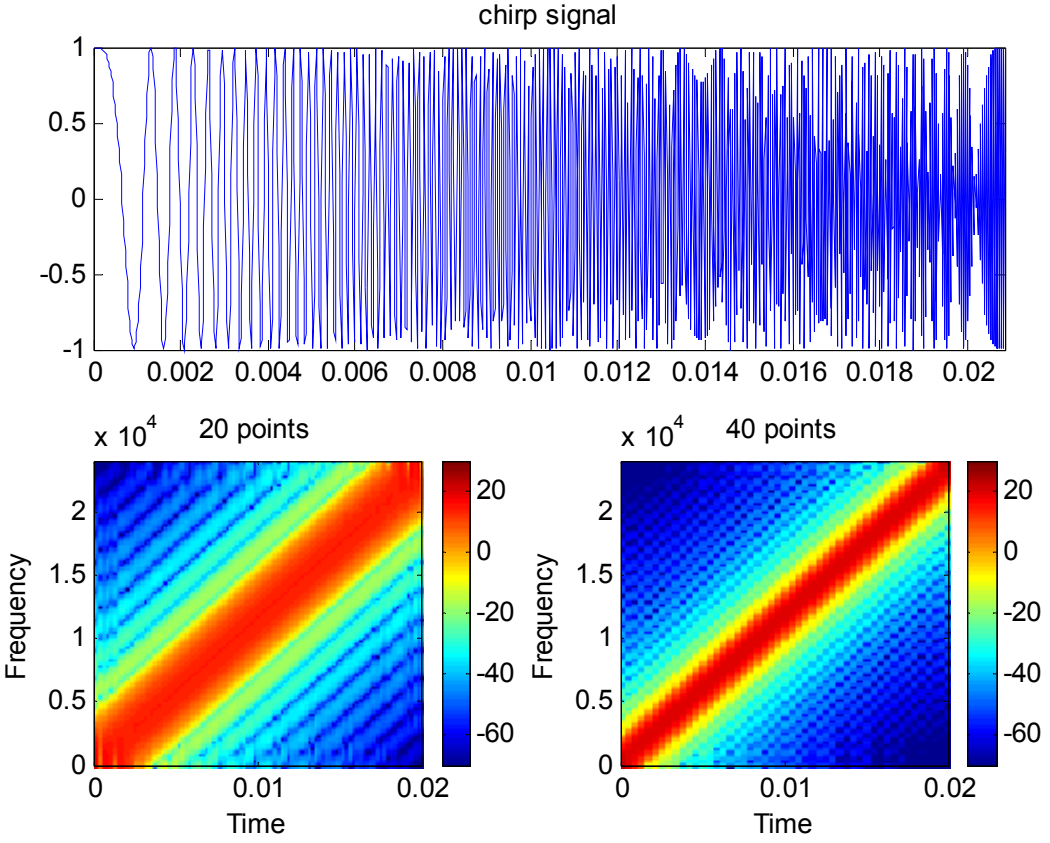
If f(t) is random, then in general you will have a difficult time of computing the Fourier Integral of the random 'data'. You need to treat the input as data and use an FFT, but the result of doing so is a random

spectrum. This single random spectrum can give you an idea of the frequency response, but in many instances it can be misleading. A better approach is to take the average of the random spectra. This leads to the formulation of power spectral density, which is an average over the FFT magnitude spectrum squared.

In certain signals, both random and deterministic, we are interested in the spectrum as a function of time in the signal. This suggests finding the spectrum over a limited time bin, moving the bin (sometimes with overlap, sometimes without), recomputing the spectrum, and so on. This method is known as the short-time Fourier Transform (STFT), or the Gabor Transform.

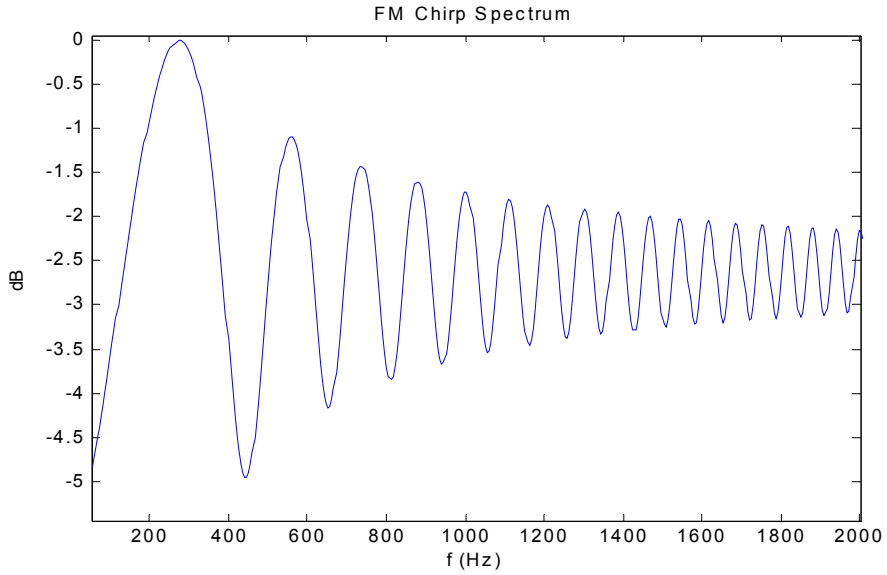
Consider an FM pulse, a deterministic signal, generated as follows

```
N=1000;
n=0:N;
s=cos(pi*n.*n/N/2);
fs=48000;
Ts=1/fs;
t=n*Ts;

subplot(2,2,1:2), plot(t,s), axis([0, max(t), -1, 1]), title('chirp
signal')
subplot(2,2,3), specgram(s,200,fs,20),
axis([0 0.02 0 2.4e4]), caxis([-70,30]), colorbar, title('20 points')
subplot(2,2,4), specgram(s,200,fs,40),
axis([0 0.02 0 2.4e4]), caxis([-70,30]), colorbar, title('40 points')
```



The STFT on the left has a lower frequency resolution because it uses a lower (20 point) window. Both show the linear progression of frequency within the chirp signal. The overlap is (by default) the window

width divided by 2. The graph on the left has better temporal resolution consisting of approximately 1000/10 = 100 data points along the time axis, whereas the graph on the right has 1000/20 = 50 points along the same time axis. Temporal resolution is determined by window width. In comparison, consider the FFT of the entire signal. The spectrum diminishes with frequency because the higher frequency components have less energy. This gives the impression of lower signal amplitude.



Continuous Wavelet Transform

The continuous Wavelet Transform is defined as follows

$$\Psi(a,b) = \int_{-\infty}^{\infty} f(t)\psi_{a,b}^{*}(t)\,dt$$

where the function $\psi_{a,b}(t)$

$$\psi_{a,b}(t) = |a|^{-1/2}\psi\left(\frac{t-b}{a}\right) \text{ or}$$

$$\psi_{a,b}(t) = |a|^{+1/2}\psi(at-b)$$

is a scaled and shifted version of the mother wavelet $\psi(t)$. As a general rule, the first form is used, but the second form is used with respect to discrete Wavelets. The scaling factor $|a|^{\pm 1/2}$ is required in order to make scaled wavelets have unit energy, and hence preserve the property of orthonomality.

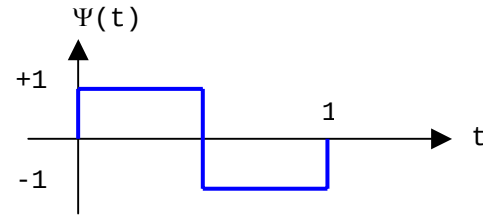
There are differences in the interpretation of scaling in each form. In the first, frequency is inversely proportional to a, in the second, frequency is proportional to a. MATLAB uses the first form, with low values of a (high frequency) occurring at the bottom of the y axis, and high values of a (low frequency) occurring at the top.

Haar Wavelet

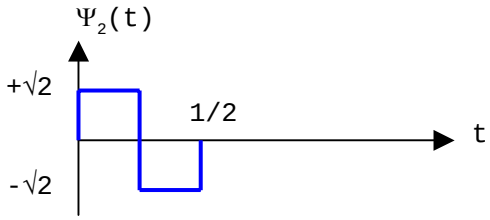
The Haar wavelet is the simplest (and, as it turns out, the earliest) wavelet.

The mother wavelet is defined as

$\Psi(t) = +1 \quad (0 \le t < 0.5)$
$\Psi(t) = -1 \quad (0.5 \le t < 1)$
$\Psi(t) = \ \ 0 \quad (\text{elsewhere})$



Time-scaled wavelets $\Psi_a(t) = |a|^{1/2}\Psi(at)$ can be produced by substituting at for t in the equation above, for example $a=2$:

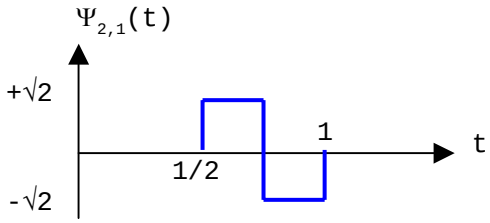


It turns out that these (unshifted) time-scaled wavelets are orthogonal for $2 \le a \le 0.5$. Increasing the scale factor above 1 shrinks the wavelet, in effect increasing its frequency. The term 'scale' in wavelet theory is similar to frequency in Fourier theory with this particular type of scaling.

Time shifted wavelets can be obtained by shifting adding a delay b to the (scaled) time as follows:

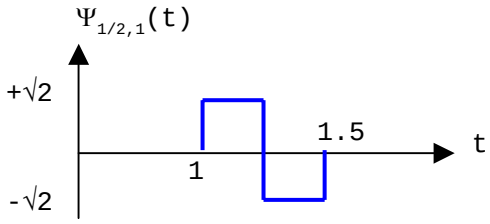
$$\Psi_{a,b}(t)=|a|^{1/2}\Psi(at-b)$$

$$\Psi_{a,b}(t) = +|a|^{1/2} \quad (b/a \le t < (0.5+b)/a)$$
$$\Psi_{a,b}(t) = -|a|^{1/2} \quad ((0.5+b)/a \le t < (1+b)/a)$$
$$\Psi_{a,b}(t) = 0 \quad \text{(elsewhere)}$$



You can interpret this as scale time by ½ and shift by ½

In comparison, the scaled and shifted wavelet $\Psi_{a,b}(t)=|a|^{-1/2}\Psi((t-b)/a)$ for a=1/2 and b=1 has the following waveform



You can interpret this as scale time by ½ then shift by 1. Its clear that this first form has a more intuitive interpretation.

The value of b does not have to be an integer in the continuous wavelet transform, but it is restricted if the basis functions are to be orthogonal. For example (assuming the second form), if a>0 and $0.5a-1 \le b \le 0.5(a-1)$then it is possible to show that b=0.5a-1 in order for the basis functions to be orthogonal. For a=3 the only allowable value of b in the interval $0.5 \le b \le 1$ is b=0.5. For a=1 the only allowable value of b in the interval $-0.5 \le b \le 0$ is b=-0.5

Hence, the Haar wavelet is not always orthonormal for any value of time shift and time scale. However, it can be made to be so by restricting the time scales to be integer powers of 2, and the time shifts to be integers. In this case, it is easy to show that the scaled and shifted wavelet will either 1) lie completely outside another wavelet with a different scale or shift (or both), or 2) lie completely inside the positive portion or negative portion (but never both at the same time) of another wavelet with a different scale or shift (or both).

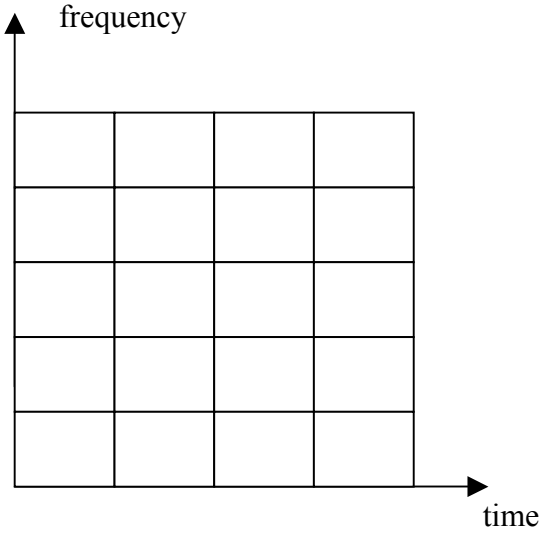
Using scale factors that are integer power of 2 provides for an intuitive way of sub-dividing the temporal and spectral axes in a complementary fashion that is amenable to numeric computation. The formulation of the dyadic Haar wavelet is as follows:

$$\psi_{a,b}(t) = |2|^{+m/2}\psi(2^m t - n)$$

where $a=2^m$ and b=n

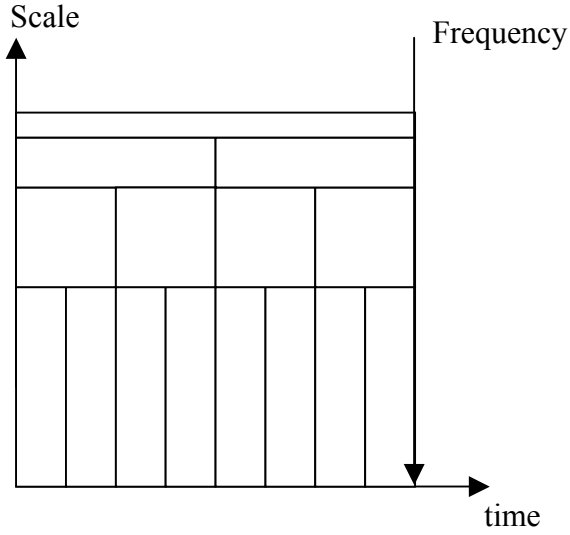
This will lead to the formulation of the discrete wavelet transform (later).

The time window extent in the Gabor Transform is, in effect, the same for all frequencies. So if you want to see things at different resolutions, you have to make separate graphs, like above. As a general rule, we want a large time window for low frequencies (so that we get many periods of the signal in the window), and a shorter time window for higher frequencies. It would be nice if we could do this on a single graph. It turns out that wavelet signal processing allows us to vary the time window width as a function of frequency. Here's a picture of what we are saying:



STFT:

Temporal and spectral resolutions are fixed

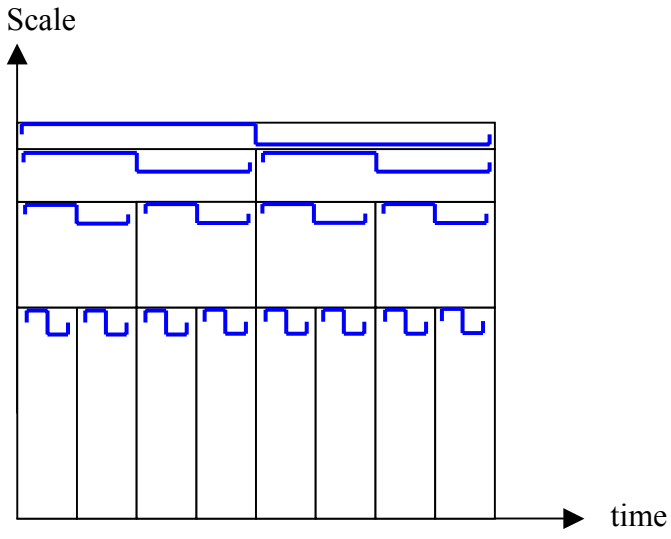


Wavelet Transform

Low frequencies have long time windows and therefore high frequency resolution

High frequencies have short windows and therefore low frequency resolution.

The thickness of the box determines the resolution along a given dimension. The key to achieving this varying temporal/spectral resolution is using an appropriate basis function.



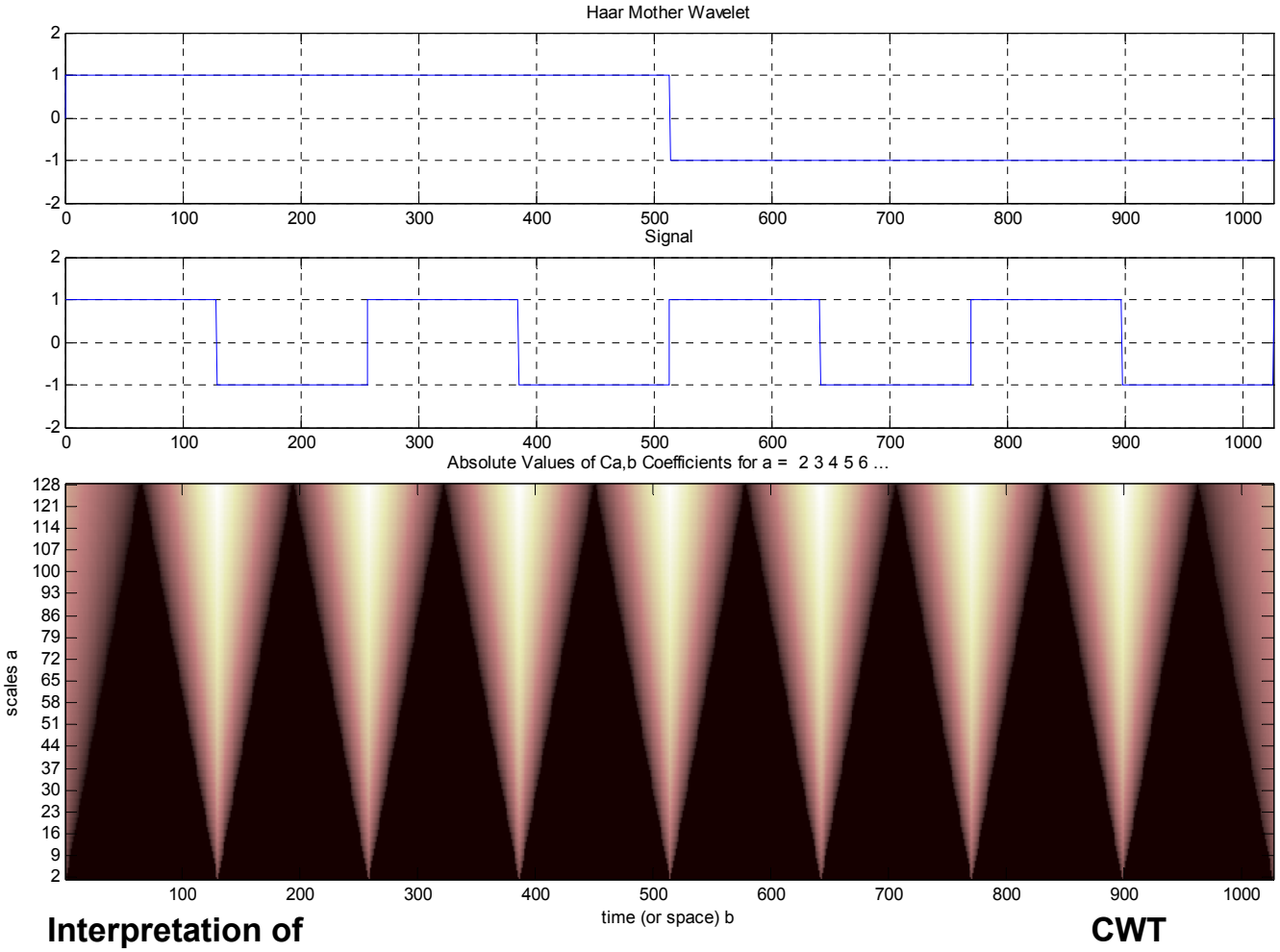
Analysis Using the Haar Wavelet

Let's use the Haar wavelet to analyze a signal that looks somewhat like the Haar wavelet itself: a square wave signal. The CWT is defined as

$$C(a,b) = \int_{-\infty}^{\infty} f(t)\psi_{a,b}(t)\,dt$$

The resulting 'spectrum' is two dimensional, the dimensions being scale factor a, analogous to 1/frequency in the Fourier domain, and time delay b, analogous to phase in the Fourier Domain. The amplitudes of C are analogous to the amplitudes of the Fourier spectrum.

```
[phi,psi,t]=wavefun('db1',10,'plot');
t=linspace(0,1,1026);
s=square(8*pi*t);
a=2:128;

subplot(4,1,1), plot(t*1026,psi), title('Haar Mother Wavelet'),
axis([0 1026 -2 2]), grid

subplot(4,1,2), plot(t*1026,s), title('Signal'),
axis([0 1026 -2 2]), grid

subplot(4,1,3:4), C=cwt(s,a,'db1','plot');
```



Interpretation of CWT

Let's make sure we understand what is going on in the above picture before we proceed.

The inverted cone pattern is periodic and corresponds to the edges in the original signal. Notice that the CWT has high resolution at low scale values (high frequency), and low resolution at high scale values (low frequency).

We will reproduce the CWT, for scale values that are integer powers of 2, using convolution.

First we take the signal `s` generated above, and pad it with 8 zeros. These zeros (16/2 of them) ensure that the convolution provides consistent results with the CWT at the beginning of the series. It is not necessary to pad the input series to the CWT. The number 16 is the largest value of scale factor considered in this example.

Next we compute the MATLAB CWT for a scale of 2. This is plotted as the (blue) dot series in the first subplot below.

```
a=2;
C2=cwt(s,a,'db1');
```

Next, we generate the wavelet for the Haar function, which is Daubechies Wavelet type 1.

```
[phi,psi,t]=wavefun('db1',log2(a));
disp(psi)
            0   1.0000   -1.0000   0
```

the second argument (call it arg2) in `wavefun` determines the number iterations. Specifically, the number of iterations is 2^(arg2), or in other words, `a`. The resulting Haar wavelet will be a+2 points long, because a zero is added to the start and end of the data, as emphasized above. There are two iterations, so we are scaling the wavelet by 2. Later, we will see that iterations corresponds to decompositions of the frequency band into halves.

Next, we scale the wavelet amplitude and flip it left-right (i.e., time reverse it) and convolve with `s`

```
psi=a^-0.5*fliplr(psi);
C2c=conv(psi,s);
```

In order to plot the two results and overlay them properly, it is necessary to pad the CWT data and truncate the convolution data by a/2 points:

```
plot([zeros(1,a/2),C2],'.'), hold on,
plot(C2c(1:(lC2+a/2)),'ro'), hold off
```

lC2 is the length of C2 = 1026+8 = 1034 points.

The results agree exactly, as shown for the first 200 or so points.

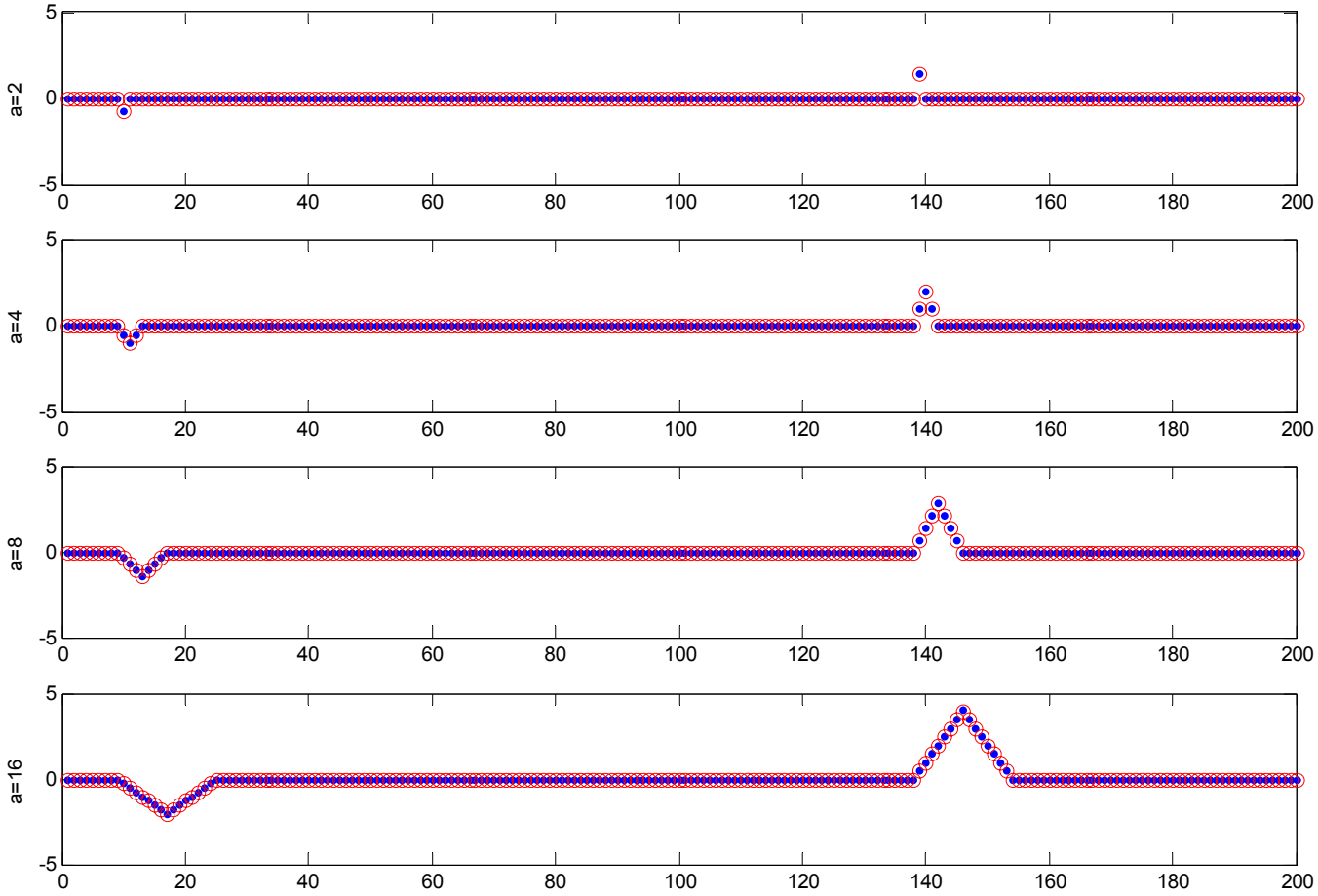
The process is repeated for a=4,8,16 and again, the convolution results agree exactly. Notice that as the scale `a` increases, both the width and amplitude of the triangle function increases. Resolution is decreasing as the scale factor increases, and the amplitude (magnitude) increases because the convolution magnitude scales as $a*a^{-0.5} = a^{+0.5}$. Notice also the CWT has a sign value that corresponds to the slope of the transition in `s`. Actually, the sign of the CWT is the negative of the slope of the transition. This information is lost in the image version of the CWT because absolute values are taken.

Having computed the CWT for a few values of scale factor, we could continue increasing `a` up to a point. That point is where the triangles would start to overlap. Since the edges in the original square wave are 128 points apart, the largest scale factor would be 128. This is the value that is chosen in the image. Of course, you can always increase `a` above 128, but the results are going to reflect superimposed edges.

We might next ask about the other values of scale factor `a` that would occur in the image representation of the CWT. What if we wanted to compute the CWT for a=6, for example? We would use exactly the same procedure as before, except we cannot use the MATLAB function `wavefun` to generate the wavelet (because it can only generate wavelets that are integer powers of 2). We have to construct the wavelet by hand – this is simple to do for the Haar wavelet

```
psi = [0 1 1 1 -1 -1 -1 0];
```

You can check for yourself that the result is correct.



```
t=linspace(0,1,1026); s=square(8*pi*t); s=[zeros(1,8),s];

a=2; C2=cwt(s,a,'db1'); lC2=length(C2);
[phi,psi,t]=wavefun('db1',log2(a)); psi=a^-0.5*fliplr(psi); C2c=conv(psi,s);
subplot(4,1,1)
plot([zeros(1,a/2),C2],'.'), hold on, plot(C2c(1:(lC2+a/2)),'ro'), hold off
axis([0 200 -5 5]), ylabel('a=2')

a=4; C4=cwt(s,a,'db1'); lC4=length(C4);
[phi,psi,t]=wavefun('db1',log2(a)); psi=a^-0.5*fliplr(psi); C4c=conv(psi,s);
subplot(4,1,2)
plot([zeros(1,a/2),C4],'.'), hold on, plot(C4c(1:(lC4+a/2)),'ro'), hold off
axis([0 200 -5 5]), ylabel('a=4')

a=8; C8=cwt(s,a,'db1'); lC8=length(C8);
[phi,psi,t]=wavefun('db1',(log2(a)));psi=a^-0.5*fliplr(psi); C8c=conv(psi,s);
subplot(4,1,3)
plot([zeros(1,a/2),C8],'.'), hold on, plot(C8c(1:(lC8+a/2)),'ro'), hold off
axis([0 200 -5 5]), ylabel('a=8')

a=16; C16=cwt(s,a,'db1');lC16=length(C16);
[phi,psi,t]=wavefun('db1',log2(a));psi=a^-0.5*fliplr(psi);C16c=conv(psi,s);
subplot(4,1,4)
plot([zeros(1,a/2),C16],'.'), hold on, plot(C16c(1:(lC16+a/2)),'ro'),hold off
axis([0 200 -5 5]), ylabel('a=16')
```

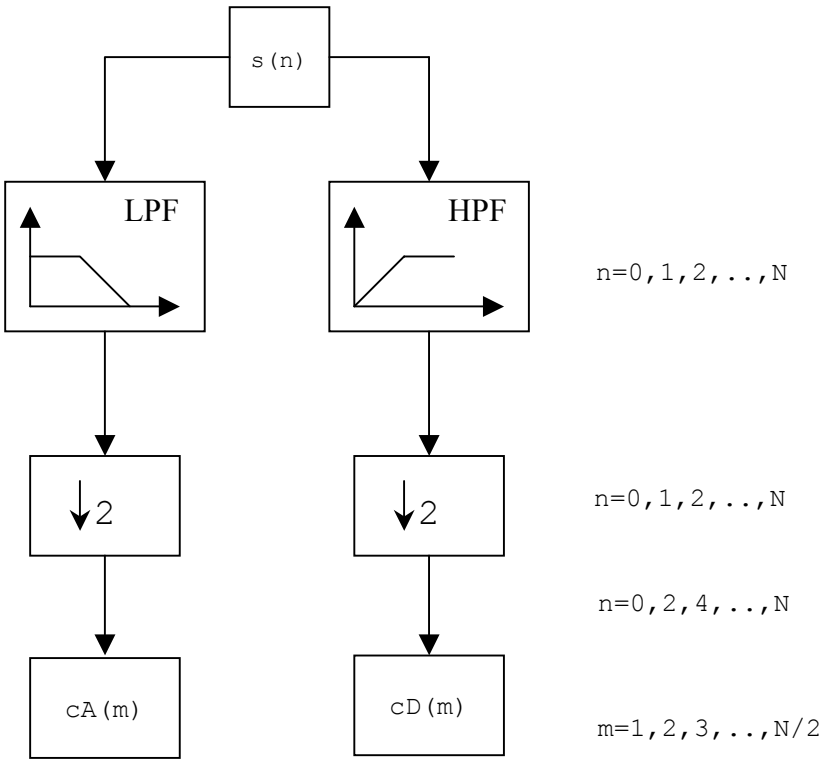
The method described above is essentially the same technique used in the MATLAB function cwt.m, with a minor difference in the implementation of the transform (it first integrates the wavelet, convolves it with the signal, then differentiates the result – this procedure is explained on page 8-30). The CWT in the MATLAB algorithm is slightly more accurate, because it does a computation more akin to a true numerical integration. The differences are on the order of 10^{-16} for the example shown.

Another point that needs to be made is that the range of values for scale and shift (a and b) can be anything you like, i.e. can be continuous. A true CWT would have a continuum of values of a and b. However, the numerically computed CWT chooses a finite set of values of a and b, and does a calculation that accurately simulates a continuous time integration performed on a discrete time signal. Even so, the resulting computation load can be substantial. One might then consider searching for efficient methods of computing the transform, perhaps using FFTs (since the process is essentially one of filtering).

It turns out that the wavelet transform can be accurately computed by simply choosing scale factors that are integer powers of 2. Since a scaling of time by a=2 corresponds to a halving of frequency, the process of producing the transform coefficients for this scale factor is equivalent to a filtering process whereby the band is divided into halves (lower and upper). It also turns out that the first sub-plot in the figure above corresponds to the upper half of the band, the high frequency terms. Of course, this makes sense – we said that low scale factor implies high frequency.

The Discrete Wavelet Transform (Fast Wavelet Transform)

As we have seen, the CWT is essentially a type of convolution, and convolution is a filtering process. The basic notion of the DWT is that this filtering can be done on a dyadic basis, dividing bands into halves. Here is a block diagram of the first stage of the DWT (taken more or less from the MATLAB manual).



Let's do the fast wavelet transform on the square wave signal that we looked at previously

```
[cA, cD] = dwt(s, 'db1');
```

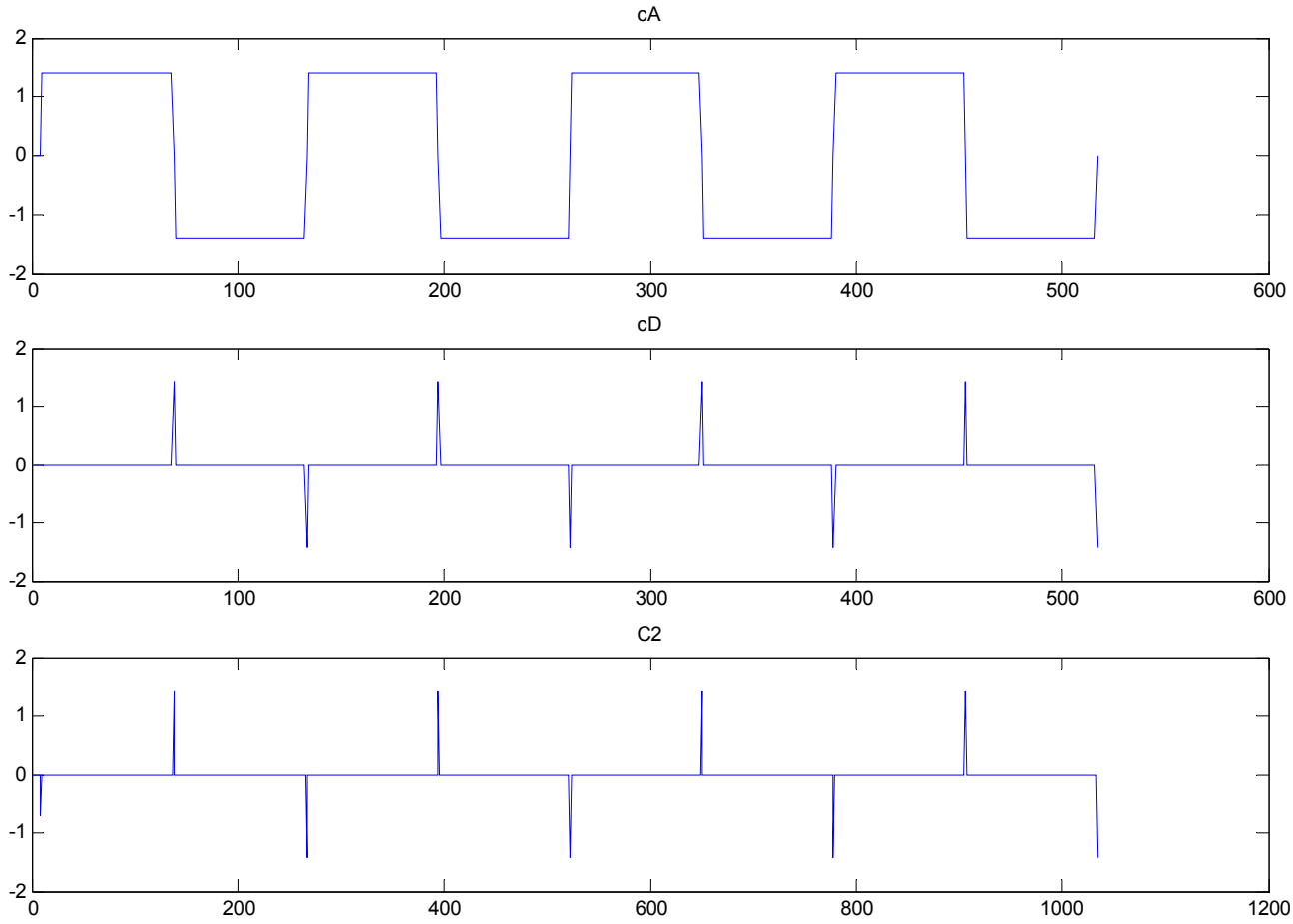
The series cA is known as the approximation coefficients, the series cD is known as the detail coefficients. The detail coefficients look remarkably like the a=2 data generated from the CWT. In fact they are the same with one important difference: the DWT data have been decimated by a factor of two (down sampled by taking only the even data points).

An important property of the DWT is that it conserves the number of points at each sub-division. Normally, downsampling a high pass spectrum results in aliasing, and it does in this case. However, it turns out that this aliasing can be compensated in the reconstruction process (more on this later).

Next, notice that the cA coefficients resemble the original signal. Strictly speaking, the original signal is the sum of the two series, interpolated by a factor of two.

Let's look first at the filtering process, then at the downsampling process.

Figure. Comparison of analysis and detail coefficients with first level (a=2) CWT.



The Haar Wavelet as a Digital Filter.

Let's consider the Haar wavelet in terms of its digital frequency response.

We noted earlier that the wavelet for a=2 is

$$\psi(n) = [0\ 1\ -1\ 0]/\sqrt{2}$$

which, as we have said before, has a corresponding convolution filter kernel

$$h(n) = [0\ -1\ 1\ 0]/\sqrt{2}$$

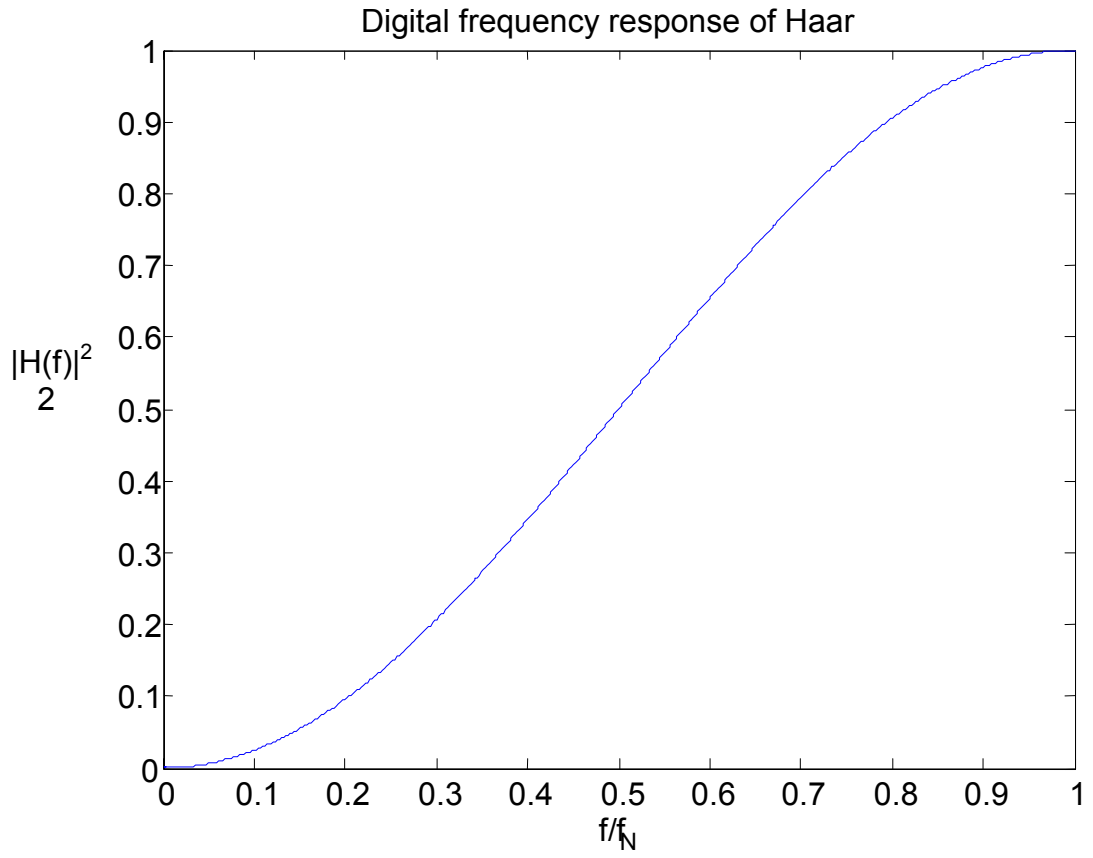
Computing the z-transform we have

$$H(z) = \mathcal{Z}(h(n)) = [-1/z + 1/z^2]/\sqrt{2} = 1/z[-1 + 1/z]/\sqrt{2}.$$

The frequency response is calculated by setting $z=\exp(j\pi f/f_N)$. Since we are interested in the magnitude of the frequency response, we can ignore the factoring $1/z$ term:

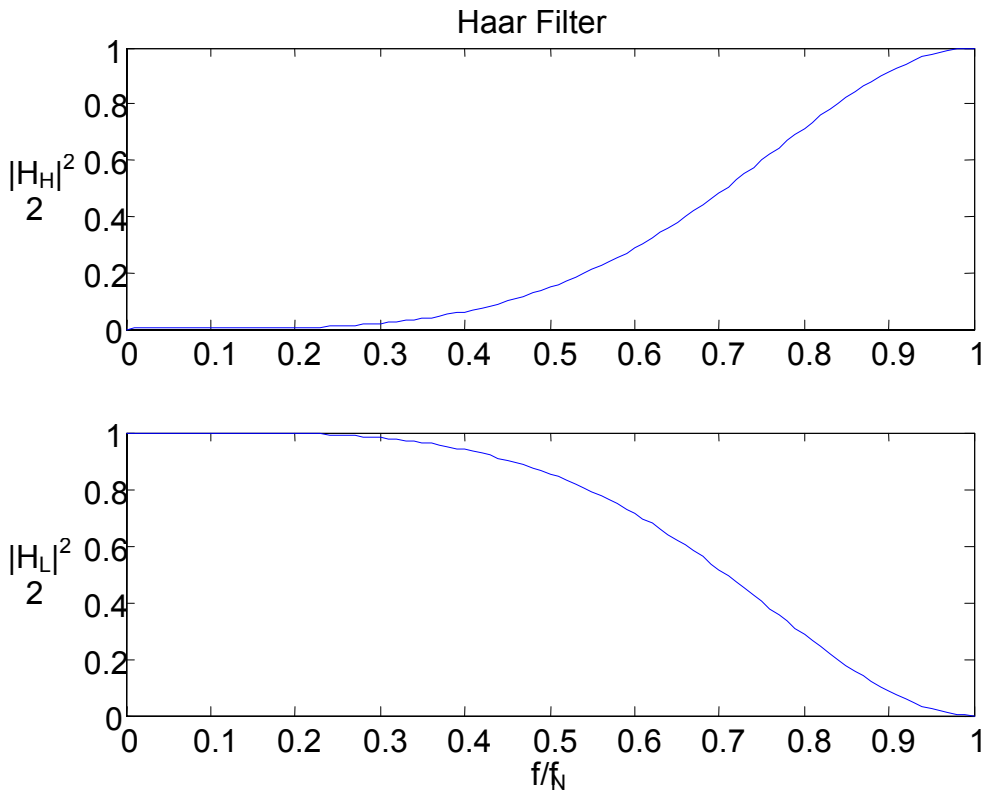
$$|H(z)| = |[-1+\exp(-j\pi f/f_N)]/\sqrt{2}| = \sqrt{2}|\sin(\pi f/f_N/2)|$$

This is clearly a high pass frequency response:



The corresponding low-pass filter (from which the approximation coefficients is derived can be found by subtracting the above frequency response from $\sqrt{2}$, i.e.

$$|H_H(f)|^2 = 2\sin^2(\pi f/f_N/2)$$
$$|H_L(f)|^2 = 2(1-\sin^2(\pi f/f_N/2)) = 2\cos^2(\pi f/f_N/2)$$



Thus, we have set things up so that the total magnitude response is 1. From an impulse response point of view, we have

$$h_H(n) = [0\ -1\ +1\ 0\ ]/\sqrt{2}$$
$$h_L(n) = [0\ +1\ +1\ 0\ ]/\sqrt{2}$$

so that

$$h_H(n) + h_L(n) = [0\ 0\ \sqrt{2}\ 0] = \sqrt{2}\delta(n-2)$$

$$|H_L(z)| = |[1+\exp(-j\pi f/f_N)]/\sqrt{2}| = \sqrt{2}|\cos^2(\pi f/f_N/2)| = \sqrt{2}\cos^2(\pi f/f_N/2)$$

The sum of both filters gives a constant and, in principle, we retain all of the information in the original signal.

Furthermore, we note that $h_L(n)$ is nothing more than the time reversal of the scaling function (properly scaled) that was generated from `wavefun`:

```
[phi,psi,t]=wavefun('db1',1)
phi =    0    1.0000    1.0000         0
psi =    0    1.0000   -1.0000         0
```

To summarize: the wavelet is a highpass filter kernel, the scaling function is a lowpass filter kernel.

Doing the DWT 'By Hand'

It is easy to implement the Haar wavelet and scaling function as FIR filters in MATLAB. Using `s` generated above:

```
t=linspace(0,1,1026); s=square(8*pi*t); s=[zeros(1,8),s];
```

First we do the filtering (high pass and lowpass), the leading 0 is removed, since is delays by 1 sample.

```
BH = [-1 +1 0]/√2
AH = 1
rH = filter(BH,AH,s);
BL = [+1 +1 0]/√2
AL = 1
rL = filter(BL,AL,s);
```

Next we do the decimation. Because the first zero is omitted in the filter kernel, we take the odd points (starting at 2), rather than the even points (starting at 1). Note even indices are odd in MATLAB.

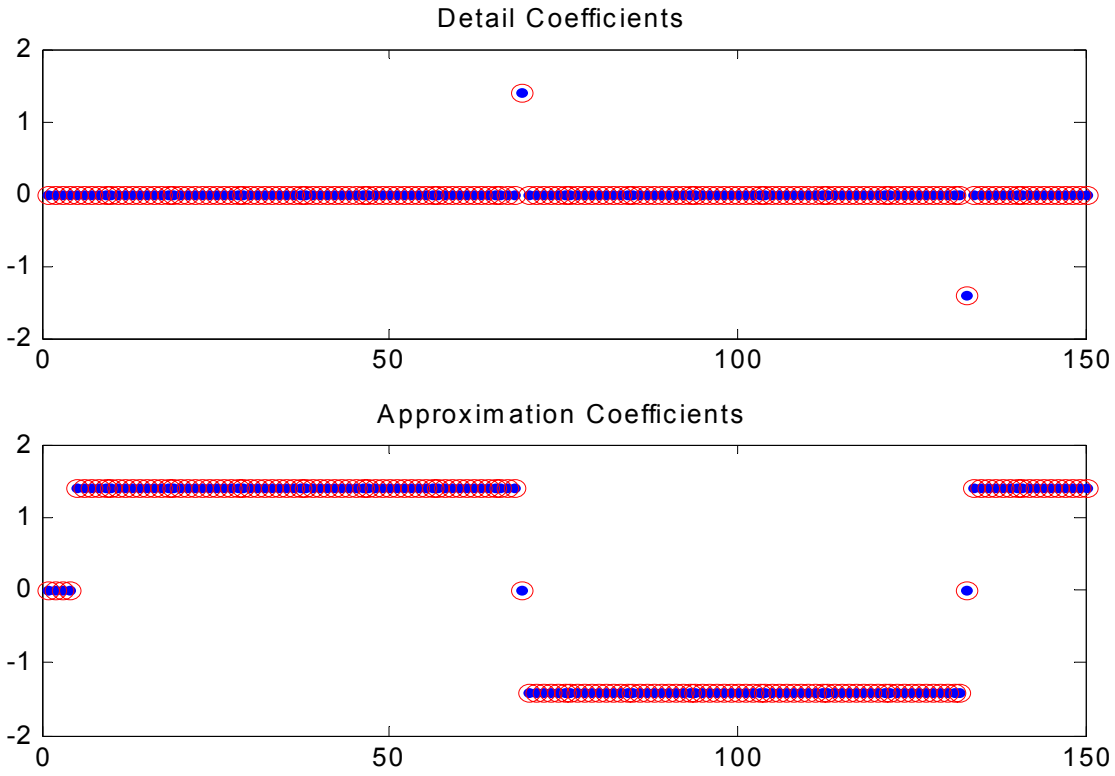
```
rHd = rH(2:2:end);
rLd = rL(2:2:end);
```

We are comparing with the analysis and detail coefficients produced by MATLAB

```
[cA, cD] = dwt(s, 'db1');
```

The plot is shown below each series has 517 points, which is half of the original 1034 points.

```
subplot(2,1,1), plot(1:517, cD,'.', 1:517, rHd,'ro')
subplot(2,1,2), plot(1:517, cA,'.', 1:517, rLd,'ro')
```



Reconstruction

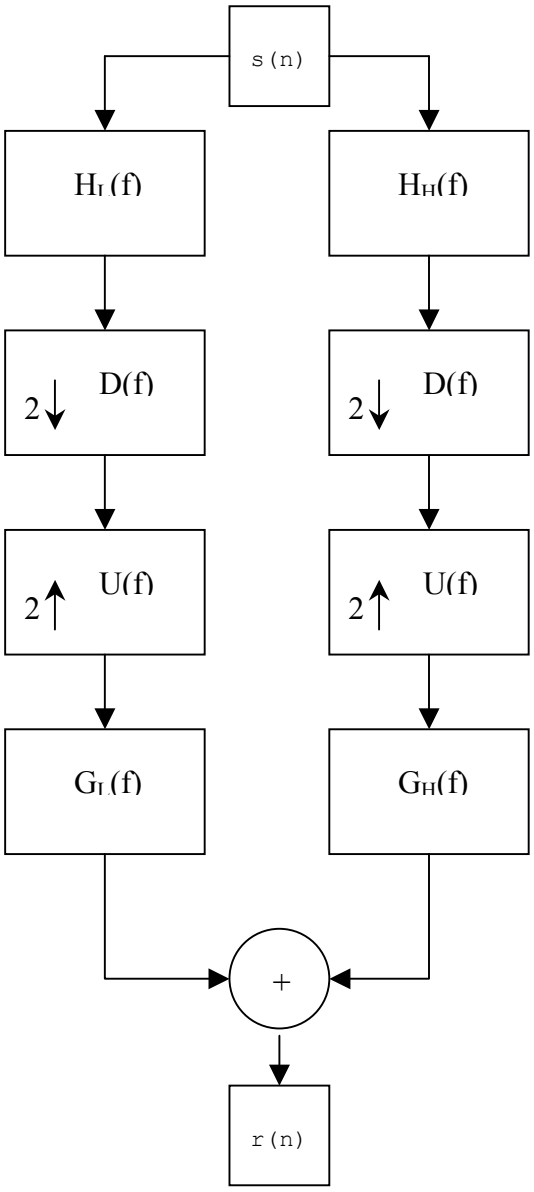
Having split the signal into lowpass and highpass components, we can repeat the process iteratively on these components. Before we do this, let's make sure that we can indeed recover the original signal.

In MATLAB this is easy: you use the function `idwt`

```
r = idwt(cA,cD,'dB1');
```

But what is going on inside this function?

The answer is two processes. First the data are interpolated by inserting zeros into every other point. Second, the zero-interpolated data are passed through a reconstruction filter. If we think of the downsampling and upsampling as filtering processes, then the reconstruction process can be represented as follows. Obviously, we want to make the reconstruction filters `G` so that `r` is as close to `s` as possible. It turns out that perfect reconstruction is possible.



In order to demonstrate reconstruction, I'm going to use an even simpler example than before.

Let the data sequence be `x=[0,1,2,3,5,8,13,21]`

As a short cut, we will use the MATLAB function DWT to derive the Haar coefficients:

```
[cA, cD]= dwt(x,'db1');
```

from which we find

```
cD = [-1 -1   -3   -8]/√2
cA = [ 1  5   13   34]/√2
```

Its rather obvious what is going on here if you look at pairs of x

```
cD = [0-1,  2-3,  5-8,  13-21]/√2
 x = [0,1,  2,3,  5,8,  13,21]/√2
cA = [0+1,  2+3,  5+8,  13+21]/√2
```

what's not so obvious is how this result can be obtained from convolution:

```
cDp = conv(x,[-1 1 0]/sqrt(2)) = [0 -1 -1 -1 -2 -3 -5 -8 21 0]/√2
cAp = conv(x,[+1 1 0]/sqrt(2)) = [0  1  3  5  8 13 21 34 21 0]/√2
                                     ↑     ↑     ↑     ↑
```

Next, we down sample the data at every other point starting at the second point, and stopping when we get to 4 points (half of the original). This is indicated above.

```
cDp = [-1 -1 -3 -8]/√2
cAp = [ 1  5 13 34]/√2
```

which match up with the coefficients produced by the DWT, cD and cA

Since cD and CA were obtained as sums and differences on x, it is reasonable to expect that x can be obtained as sums and differences on cD and cA.

For example (and using regular index notation)

```
cD(0) + cA(0) = (-1+1)/√2 = 0 = x(0)*√2
cD(1) + cA(1) = (-1+5)/√2 = 4/√2 = x(2)*√2
```

It is clear that we can get the even components of x by adding cD to cA and dividing by $\sqrt{2}$

Similarly we can get the odd components of x by subtracting cD from cA and then dividing by $\sqrt{2}$. For example

```
cA(0) - cD(0) = (1+1)/√2 = 2/√2 = x(1)*√2
cA(1) - cD(1) = (5+1)/√2 = 6/√2 = x(3)*√2
```

This is all well and good on paper, but we need a method of implementing this with filters. We start by interpolating the data with zeros in the odd positions, because these were the positions that were removed.

```
cDi  = [-1  0 -1  0  -3  0  -8  0 ]/√2
cAi  = [ 1  0  5  0  13  0  34  0 ]/√2
```

Next, we produce delayed versions of these sequences :

```
cDid = [ 0 -1  0 -1   0 -3   0 -8 ]/√2
cDi  = [-1  0 -1  0  -3  0  -8  0 ]/√2
cAi  = [ 1  0  5  0  13  0  34  0 ]/√2
cAid = [ 0  1  0  5   0 13   0 34 ]/√2
```

Next we add the undelayed versions

```
cDi  = [-1  0 -1  0  -3  0  -8  0 ]/√2
cAi  = [ 1  0  5  0  13  0  34  0 ]/√2
cDi+cAi  = [ 0  0  4  0  10  0  26  0 ]/√2
```

and subtract the delayed versions

```
cDid = [ 0 -1  0 -1   0 -3   0 -8 ]/√2
cAid = [ 0  1  0  5   0 13   0 34 ]/√2
cAid-cDid = [ 0  2  0  6   0 16   0 42 ]/√2
```

Then add the two results

```
  cDi+cAi  = [ 0  0  4  0  10  0  26  0 ]/√2
cAid-cDid  = [ 0  2  0  6   0 16   0 42 ]/√2
         Σ = [ 0  2  4  6  10 16  26 42 ]/√2
```

Then divide by √2

```
         Σ = [ 0  2  4  6  10 16  26 42 ]/2
         Σ = [ 0  1  2  3   5  8  13 21 ] = x.
```

However, what we would really like to do is implement the summing and differencing with filters:

This is easy to do (perhaps even easier if you draw a block diagram) once you recognize that the same out put is derived from

$$\Sigma = (cDi(n) - cDi(n-1))/\sqrt{2} + (cAi(n) + cAi(n-1))/\sqrt{2}$$

Then the reconstruction filter $G_H(z)$ for cDi is $[1 - 1/z]/\sqrt{2}$, whereas the reconstruction filter $G_L(z)$ for cAi is $[1 + 1/z]/\sqrt{2}$.

In terms of a convolution, we would find the output of $G_H(z)$ and $G_L(z)$ as

```
(conv(cDi,[1 -1]))/sqrt(2) = [-0.5 0.5 -0.5 0.5 -1.5 1.5 -4.0  4.0 0.0]
(conv(cAi,[1  1]))/sqrt(2) = [ 0.5 0.5  2.5 2.5  6.5 6.5 17.0 17.0 0.0]
                         Σ = [ 0.0 1.0  2.0 3.0  5.0 8.0 13.0 21.0 0.0]
```

We get a one extra sample at the end, which we ignore.

In terms of a filter operation

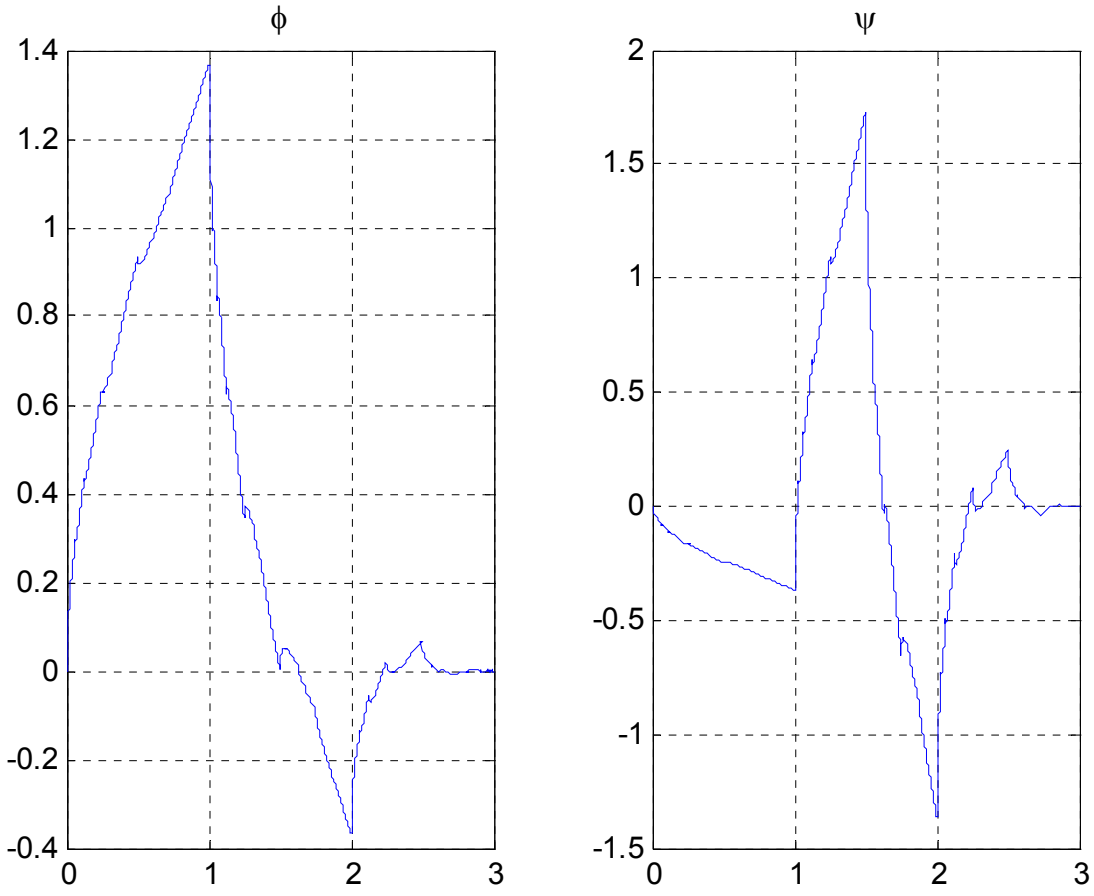
```
filter([1 -1]/sqrt(2),1,cDi) = [-0.5 0.5 -0.5 0.5 -1.5 1.5 -4.0  4.0]
filter([1  1]/sqrt(2),1,cAi) = [ 0.5 0.5  2.5 2.5  6.5 6.5 17.0 17.0]
                           Σ = [ 0.0 1.0  2.0 3.0  5.0 8.0 13.0 21.0]
```

Then we end up with something that looks very similar to the previous result but, more interestingly, we see that the reconstruction filter impulse response is simply the time reversal of the original filter bank (minus the zeros).

Will this process work for other wavelets?

Let's try it out on the 'db2' wavelet. At this point, we don't know how to generate this wavelet, so we will do it using MATLAB. Before we do this, let's take a quick look at the wavelet and its scaling function

```
[psi,phi,t] = wavefun('db2',10);
subplot(1,2,1) plot(t,psi), subplot(1,2,2), subplot(t,phi)
```



As you can see, the wavelet has a dominant positive and negative peak, whereas the scaling function has a dominant positive peak and a much smaller negative one (giving it an overall additive, i.e., lowpass effect). We will see later how to generate this 'detailed picture' from basic filter coefficients.

For a first level decomposition, we obtain the wavelet as follows:

```
[phi,psi,t]=wavefun('db2',1)

phi = [ 0  0.6830  1.1830  0.3170 -0.1830  0  0 ]
psi = [ 0 -0.1830 -0.3170  1.1830 -0.6830  0  0 ]
```

Next, we do the wavelet transform the quick way

```
[cA,cD]=dwt(x,'db2');

cA =  [ 0.3536   0.8966   3.5609   9.3032  26.8701 ]
cD =  [-0.6124  -0.0000  -0.6124  -1.7077   4.8990 ]
```

Notice that there are 5 points (as opposed to 4 from before) in the coefficient data, this number is equal to the length of x divided by 2, plus (n-1), wher n is the order of the Daubechies wavelet.

Next, we interpolate cA and cD with zeros:

```
cAi =  [ 0.3536  0  0.8966 0  3.5609 0  9.3032 0 26.8701 0]
cDi =  [-0.6124  0 -0.0000 0 -0.6124 0 -1.7077 0  4.8990 0]
```

Then do the reconstruction as previously formulated

```
(filter(psi(2:5),1,cDi) + filter(phi(2:5),1,cAi))/sqrt(2)

   =  [0.250 0.433 0.0 1.0 2.0 3.0 5.0 8.0 13.0 21.0]
```

This is the original sequence, with two extra data points added onto the front. We are almost there if we can figure out a way to remove these precursor data points. It turns out that extra points tacked onto the front are always 2(n-1) in number, where n is the index in dBn.

Hence, perfect reconstruction is possible with any of the Daubechies wavelet functions, at least from the first level decomposition.

Let's see how this technique can be used to split a noisy signal into approximation and detail, using the db2 wavelet. We will use MATLAB functions where available, rather than the long-hand methods detailed above – with the understanding that you should be able to do the long-hand methods in a non-MATLAB environment.

Application of DWT

First we load a file that is supplied with wavelet toolbox called noissin.mat

```
load noissin
```

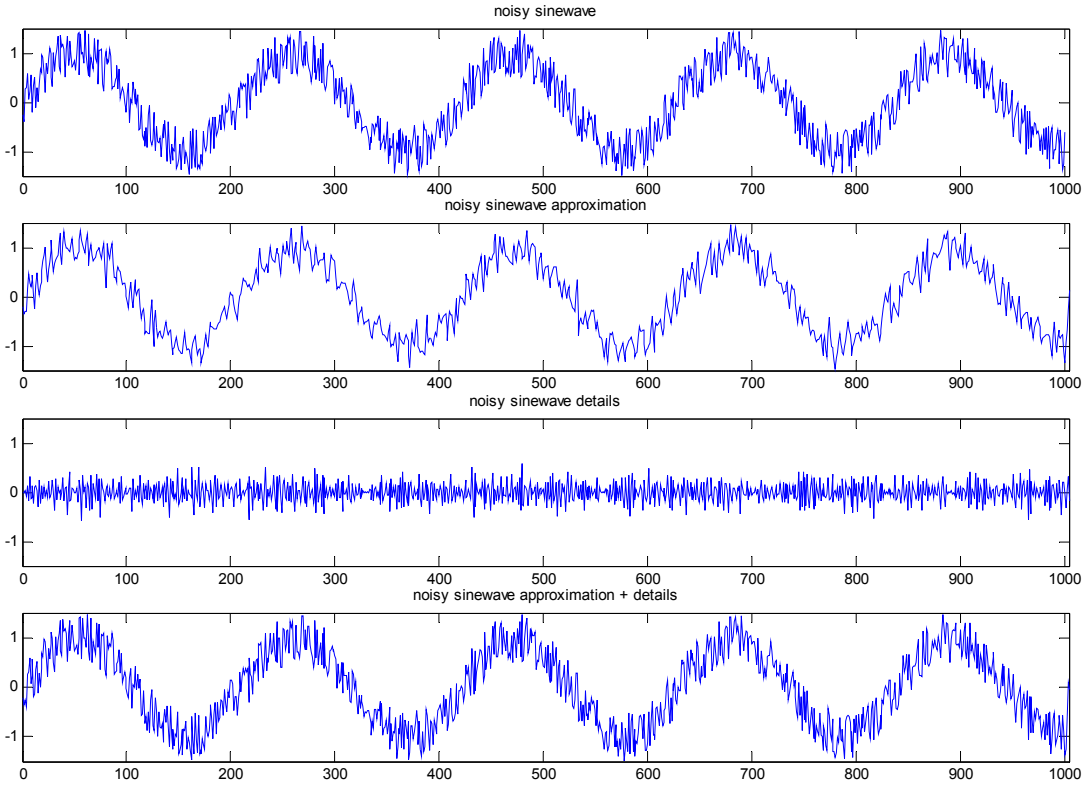
You should find that noissin is a noisy sinusoid with 1000 real data values. Next we do a first-level wavelet decomposition as follows:

```
[cA,cD]=dwt(noissin,'db2');
```

We could equally have obtained the coefficients by filtering, as demonstrated before. Next, we reconstruct the approximation and detail signals separately, but do not add them. Rather than filtering the zero interpolated coefficients with the reversed filter kernels, we use the MATLAB function `upcoef`, which does everything in one function. The argument at the end is the number of steps in the reconstruction. Usually, this is set to 1.

```
sA=upcoef('a',cA,'db2',1);
sD=upcoef('d',cD,'db2',1);
```

Next we plot the results (noissin, sA, sD, and sA+sD):

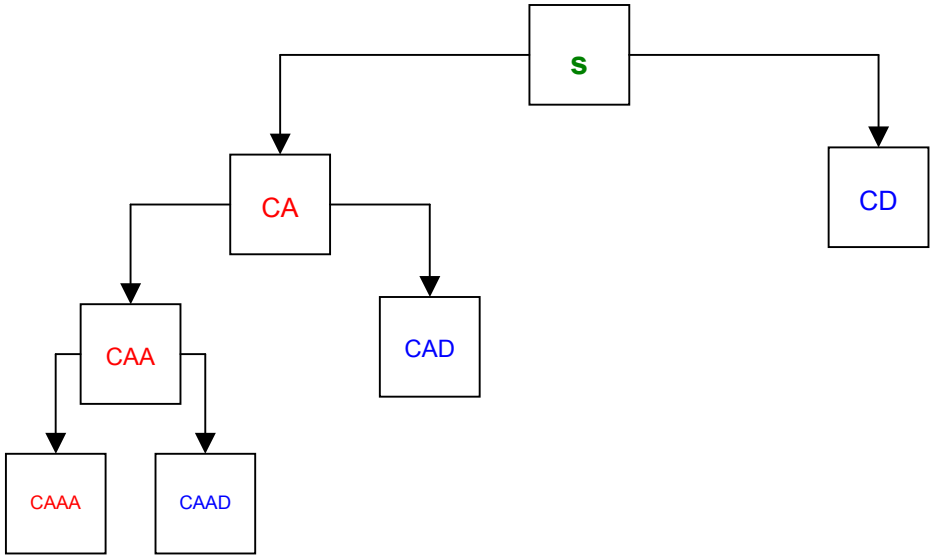


Notice that the details in this decomposition are noise – they could be left out (this is called denoising – more on this later). The approximation is still noisy though; this suggests further decomposition.

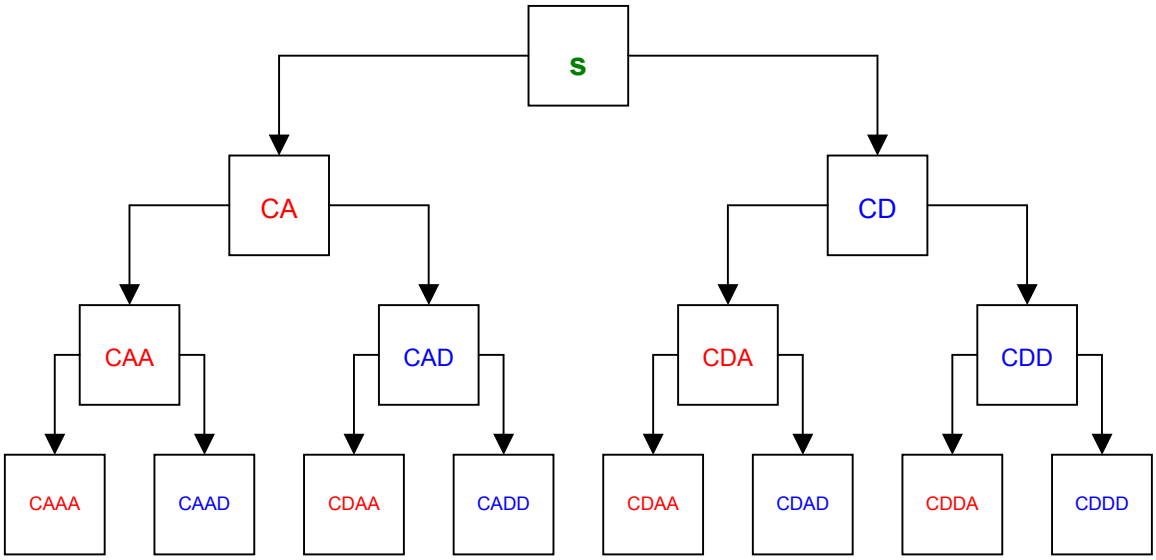
Multiple Decompositions

The notion behind multiple decompositions is that both approximations and details can, in principle, be further subdivided into approximations and details. There are two basic methods for doing this that are recognized in wavelet analysis:

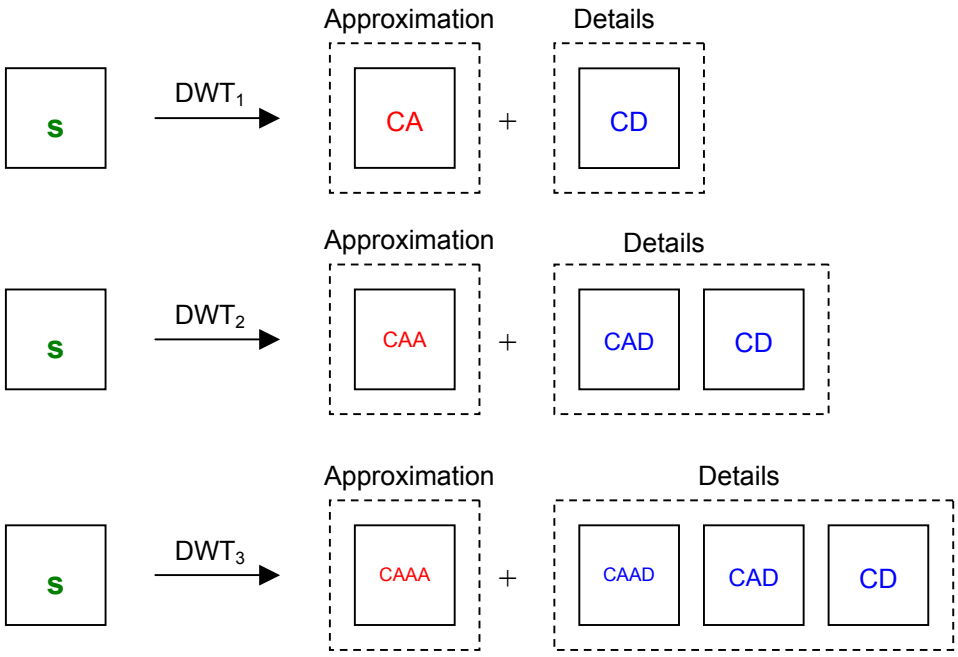
Wavelet Decomposition
Decompose the original signal into approximation and detail. Decompose the resulting approximation into approximation and detail, but leave the detail alone. Repeat decompositions on subsequent approximations only.



Wavelet Packets
Decompose the original signal into approximation and detail. Decompose all subsequent approximations and details into approximations and details.



Traditional wavelet analysis is based on the wavelet decomposition, which is a hierarchical process described as follows.



After each DWT, the approximation coefficients are divided into two bands using the same filter as before, with the result that the details are appended with the details of the latest decomposition. At each level, the signal `s` can be reconstituted from the approximation and detail coefficients.

Let's go through a detailed example of such a decomposition of a fibonnaci sequence using MATLAB's `dwt` to save time

```
x = [0 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987]

[cA, cD] = dwt(x,'db1');

cA = [ 1  5 13  34  89  233  610  1597]/√2
cD = [-1 -1 -3  -8 -21  -55 -144  -377]/√2

[cAA, cAD] = dwt(cA,'db1');

cAA = [ 3  23.5 161 1103.5 ]
cAD = [-2 -10.5 -72 -493.5 ]

[cAAA, cAAD] = dwt(cAA,'db1');

cAAA = [ 18.7383  894.1365]
cAAD = [-14.4957 -666.4481 ]
```

```
C = [cAAA | cAAD, cAD, cD]

      [18.7  894.1                                                  ← cAAA
      -14.5 -666.4                                                  ← cAAD
       -2.0  -10.5  -72.0 -493.5                                    ← cAD
       -0.7   -0.7   -2.1   -5.7  -14.8  -38.9 -101.8 -266.6]      ← cD
```

Where `C` is defined as the complete 3-level wavelet decomposition.

As you might expect, there is a function in MATLAB that does all this for you:

```
[C,L] = wavedec(x,3,'db1');

C = [18.7  894.1  -14.5 -666.4  -2.0  -10.5  -72.0 -493.5
     -0.7   -0.7   -2.1   -5.7 -14.8  -38.9 -101.8 -266.6]

L = [2 2 4 8 16]
```

The first 4 elements of L indicate the lengths of coefficient sequences. The last element (k+2, where k=3 above) is the original length of the input vector, x. Another way to store the coefficient data in MATLAB would be to use a cell array:

```
C{1} = [ 18.7  894.1]
C{2} = [-14.5 -666.4]
C{3} = [ -2.0  -10.5  -72.0 -493.5]
C{4} = [ -0.7   -0.7   -2.1   -5.7  -14.8  -38.9 -101.8 -266.6]
```

Another approach would be to use MATLAB structures:

```
c.A.A.A = [ 18.7  894.1]
c.A.A.D = [-14.5 -666.4]
c.A.D   = [ -2.0  -10.5  -72.0 -493.5]
c.D     = [ -0.7   -0.7   -2.1   -5.7  -14.8  -38.9 -101.8 -266.6]
```

Then c has a logical structure, which can be explored as follows:

```
>> disp(c)        A: [1x1 struct]
                  D: [-0.7 -0.7 -2.1 -5.7 -14.8 -38.9 -101.8 -266.6]

>> disp(c.A)      A: [1x1 struct]
                  D: [-2.0 -10.5 -72.0 -493.5]

>> disp(c.A.A)    A: [ 18.7  894.1]
                  D: [-14.5 -666.4]
```

Now let's see if we can reconstruct the original signal from the decomposition.

First we assume that the decomposition is in the native MATLAB format that results from `wavedec`

```
C = [18.7  894.1  -14.5 -666.4  -2.0  -10.5  -72.0 -493.5
     -0.7   -0.7   -2.1   -5.7 -14.8  -38.9 -101.8 -266.6]

L = [2 2 4 8 16]
```

Next we compute a set of indices for extracting the data

```
Ls = cumsum(L)            = [2 4 8 16 32]
Ls2 = Ls(1:end-1)         = [2 4 8 16]
Ls1 = [0 Ls2(1:end-1)]+1  = [1 3 5  9]
```

We initiate the process by extracting the approximation cAAA and detail cAAD

```
cAAA = C(Ls1(1):Ls2(1));
cAAD = C(Ls1(2):Ls2(2));
```

Next, we must reconstruct `cAA` from `cAAA` and `cAAD`

```
cAA = upcoef('a',cAAA,'db1',1) + upcoef('d',cAAD,'db1',1);
cAA = [ 3  23.5 161 1103.5 ]
```

Then `cA` from `cAA` and `cAD`:

```
cAD = C(Ls1(3):Ls2(3));
cA  = upcoef('a',cAA,'db1',1) + upcoef('d',cAD,'db1',1);
cA  = [ 0.7  3.5  9.2  24.0  62.9  164.8  431.3  1129.2 ]
```

Then `xx` from `cA` and `cD`:

```
cD  = C(Ls1(4):Ls2(4));
xx   = upcoef('a',cA,'db1',1) + upcoef('d',cD,'db1',1);
xx   = [-0 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987]
```

The difference between `xx(1)` and `x(1)` is on the order of 10^{-15}.

Reconstruction can be done in a single step using the `waverec` algorithm:

```
xx = waverec(C,L,'db1');
```

It produces exactly the same result as before. You could easily make your own `waverec` algorithm by automating the above process in a 'for' loop. It would be relatively easy to modify the loop to omit details, where desired.

Let us try to summarize what we have learned so far.

We have learned how to do the continuous wavelet transform and the discrete wavelet transform using the Haar wavelet. We understand the frequency response of the Haar wavelet, and how to go about doing multiple level decomposition and reconstruction. Our next endeavor will be to understand how other wavelets are formulated, starting with the Daubechies wavelets of order higher than 1.

Daubechies Wavelets of General Order.

We have introduced wavelet theory using the Haar wavelet, which was chosen for this purpose because of its simplicity. However, many real signals are not Haar-like in nature, so it behooves us to search for other wavelets that better represent such signals. The Daubechies filters (wavelets) have three important properties:

1. They are orthogonal
2. They are maximally flat
3. They are compactly supported (the impulse response is zero outside a certain time interval)

You may remember an analog filter that is maximally flat – the Butterworth filter. There are two basic approaches to designing the amplitude response of filters: introduce ripple into the passband, or stop band, or both, in order to obtain a sharp transition band, or ensure a flat passband and stopband, and put up with a wider transition band. The Daubechies filters are maximally flat in the sense that a number of derivatives of the filter polynomial are zero at $f/f_N=0$ and $f/f_N=\pi$. It turns out that this property is important for the iterative decomposition that is done in the discrete wavelet transform. Digital filters that introduce ripple tend to be unstable when multiple applications of "downsample by 2 then filter" are applied to a discrete-time data sequence.

In order to understand this property, we need to take a sidestep into a mathematical note on the binomial series.

Binomial Series

The binomial series is simply a polynomial in x^n weighted by the binomial coefficient $\binom{N}{n}=\frac{N!}{n!(N-n)!}$.

This is a formula for counting the number of ways of choosing n out of N possible things. The binomial polynomial of degree k-1 has k coefficients and is formulated as follows:

$$B_k(x)=\sum_{n=0}^{k-1}\binom{k+n-1}{n}x^n=1+kx+\frac{k(k+1)}{2!}x^2+\cdots+\frac{k(k+1)(k+2)\cdots(k+n-1)}{n!}x^{k-1}$$

The polynomial is modified by multiplying it by $2(1-x)^k$ the result is called $P(x)$:

$$P_k(x)=2\sum_{n=0}^{k-1}\binom{k+n-1}{n}x^n(1-x)^k$$

This summation is similar to the Binomial distribution used in probability theory, recognizing x as the probability of a binary random variable, then $\binom{N}{n}x^n(1-x)^{N-n}$ is the probability of n out of N occurrences of one state of the random variable (having probability x). For this distribution

$$\sum_{n=0}^{N}\binom{N}{n}x^n(1-x)^{N-n}=1$$

(the probabilities sum to 1, as they must do so in any discrete probability distribution.) P_K has some interesting properties in its derivatives. It is easy to show this numerically in MATLAB

The coefficients of P_K can be computed as `prod(k:(k+n-1))/factorial(n)`. So the polynomial B can be computed in a for-loop as follows:

```
Bk=[1, zeros(1,k-1)];
for n = 1:k-1, Bk(n+1)= prod(k:(k+n-1))/factorial(n); end
```

To find P, we simply multiply by $(1-x)^k$, which is achieved by convolution with $(1-x)^k$ expanded using the poly function

```
Pk = 2*conv(Bk, poly(ones(1,k)));
```

Some examples are shown below

```
k     Bk                Pk
1     [1]               [2 -2]
2     [1 2]             [2  0 -6   4]
3     [1 3 6]           [2  0  0 -20  30  -12]
4     [1 4 10 20]       [2  0  0   0 -70  168 -140    40]
5     [1 5 15 35 70]    [2  0  0   0   0 -252  840 -1080 630 -140]
```

The coefficients are shown for x^k, so for, example, $P_3(x) = 2-20x^3+30x^4-12x^5$

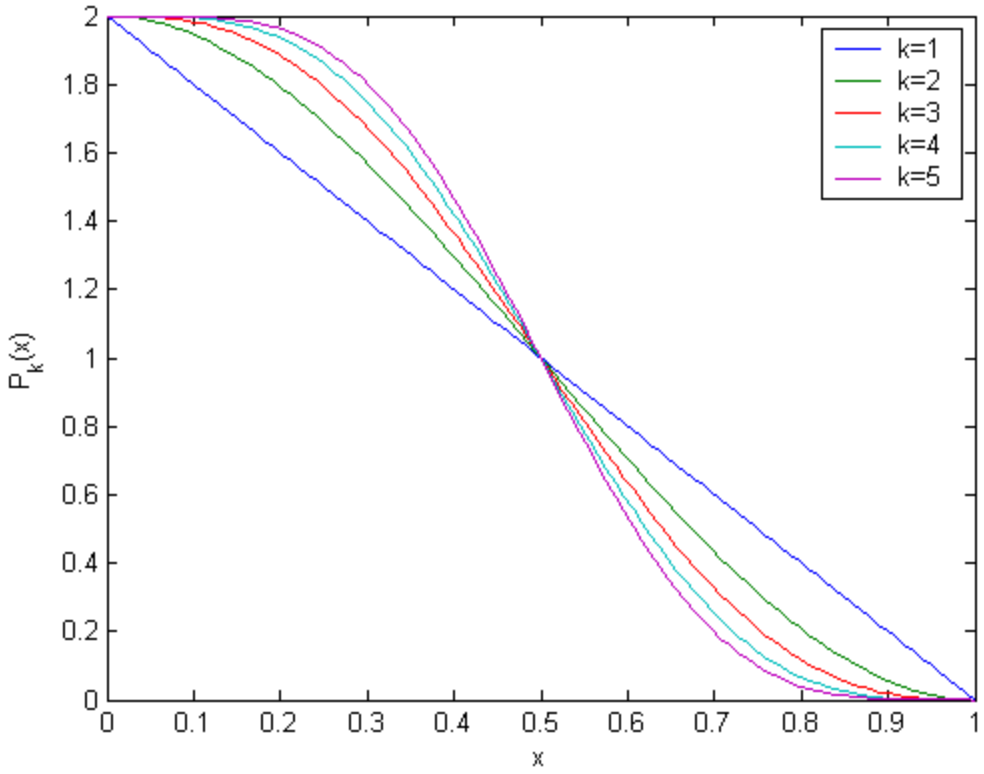
We note that `Pk(1) = 2` and `sum(Pk) = 0`, always.

This means that $P_k(x=0) = 2$ and $P_k(x=1 = 0)$, always.

It turns out that $P_k(x)$ transitions smoothly from 2 to 0 as x ranges from 0 to 1. This is illustrated for the above 5 examples:

```
Pk{1} = [2 -2]
Pk{2} = [2  0 -6   4]
Pk{3} = [2  0  0 -20  30  -12]
Pk{4} = [2  0  0   0 -70  168 -140    40]
Pk{5} = [2  0  0   0   0 -252  840 -1080 630 -140]

x = linspace(0,1,100);                          % create an x-series
for k=1:5
      xx= zeros(2*k-1, length(x));              % initialize matrix
      for m=0:2*k-1, xx(m+1,:) = x.^m; end      % for matrix of x.^m
      Pkx(k,:) = Pk{k}*xx;                      % compute Pk(x)
      plot(x,Pkx), legend('k=1', 'k=2', 'k=3', 'k=4', 'k=5')
end
```

The plot, with labels added, is shown on the next page:



We see a family of curves that transition smoothly from 2 to 0. With the exception of k=1, the curves have zero slope at both x=0 and x=1. It is easy to compute the slopes of these curves (3 shown)

k	Pk	Pk'
1	[2 -2]	[2*0 -2*1]
2	[2 0 -6 4]	[2*0 0*1 -6*2 -4*3]
3	[2 0 0 -20 30 -12]	[2*0 0*1 0*2 -20*3 30*4 -12*5]

For example $P_3'(x) = -60x^3 + 120x^4 - 60x^5$ so $P_3'(0) = 0$ and $P_3'(1) = -60 + 120 - 60 = 0$

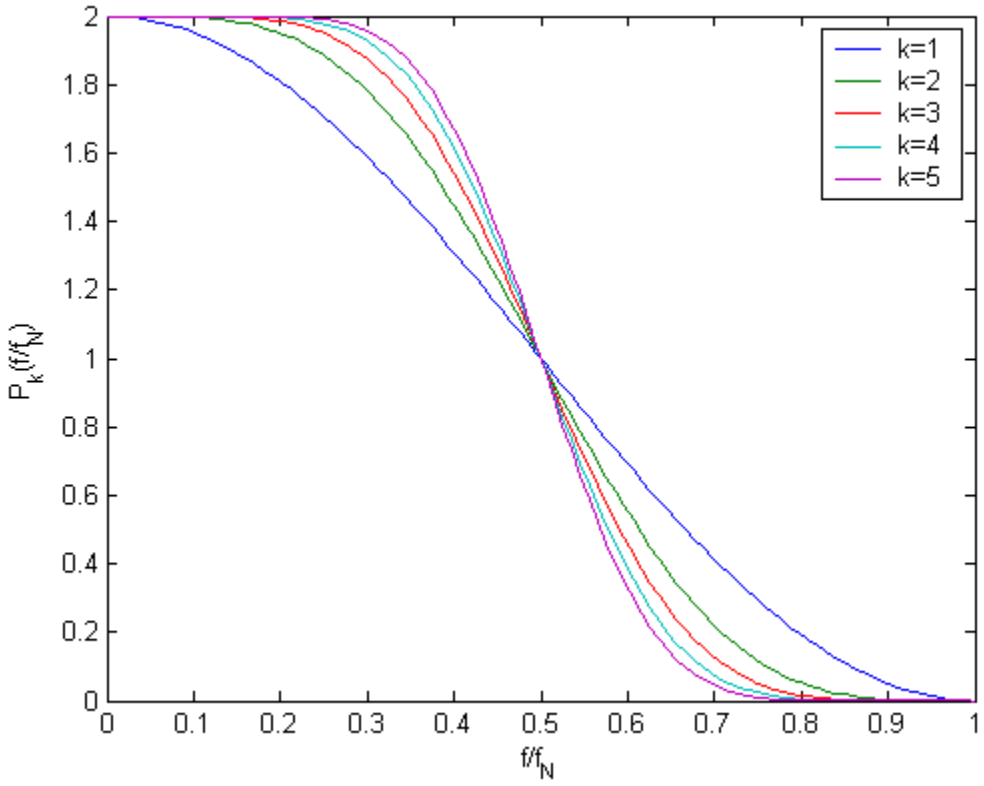
These curves bare a striking resemblance to the frequency response of the Haar scaling function. If you recall, the lowpass Haar filter amplitude response is $|H_L(f)|^2 = 2\cos^2(\pi f/f_N/2)$ – thus it would seem that the power response $|H_L(f)|^2$ is similar in nature to $P_k(x)$, where x has been transformed into normalized frequency f/f_N.

In fact, if we make a substitution $x = (1-\cos(\pi f/f_N))/2$ then

$$P_1(x) = 2-2x = 2-(1-\cos(\pi f/f_N)) = 1+\cos(\pi f/f_N) = 2\cos^2(\pi f/f_N/2) = |H_L(f)|^2$$

Amazing! We have discovered that the frequency response of the Haar scaling function is related to a binomial series weighted by k zeros $(1-x)^k = \cos^{2k}(\pi f/f_N/2)$. To see how the transformation has affected the frequency responses, we rerun the plotting segment above with a minor modification

```
f = linspace(0,1,100);                          % create f-series
x = (1-cos(pi*f))/2;                            % transform to x
for k=1:5
      xx= zeros(2*k-1, length(x));              % initialize matrix
      for m=0:2*k-1, xx(m+1,:) = x.^m; end      % for matrix of x.^m
      Pkx(k,:) = Pk{k}*xx;                      % compute Pk(x)
      plot(x,Pkx), legend('k=1', 'k=2', 'k=3', 'k=4', 'k=5')
end
```



The picture is becoming clearer – the k=1 response is the same as the power response of the Haar scaling function. The higher values of k produce responses with sharper cut-offs and, as it turns out, scaling functions and wavelets in the Daubechies family with an increasingly smooth appearance.

This is a really neat result, and one that mathematicians get excited about. Why do mathematicians get excited about stuff like this? Well, because a seemingly abstract equation from number theory turns out to have an important role in filter design. This role was hidden until discovered recently by Ingrid Daubechies and published in 1988.

We now need a method of synthesizing amplitude responses (and hence impulse responses) from the power response provided by `Pk(x)`. This is addressed next.

Synthesizing Half-Band Frequency Responses.

Let's try to summarize our progress so far. We have a polynomial representation for the power response of a class of filters that are the basis for the Daubechies scaling and wavelet functions.

$$P_k(x) = 2\sum_{n=0}^{k-1}\binom{k+n-1}{n}x^n(1-x)^k$$

substituting $x = (1-\cos(\pi f/f_N))/2$ gives

$$P_k(f/f_N) = 2\left(\frac{1+\cos(\pi f/f_N)}{2}\right)^k \sum_{n=0}^{k-1}\binom{k+n-1}{n}\left(\frac{1-\cos(\pi f/f_N)}{2}\right)^n$$

$$P_k(1-f/f_N) = 2\left(\frac{1-\cos(\pi f/f_N)}{2}\right)^k \sum_{n=0}^{k-1}\binom{k+n-1}{n}\left(\frac{1+\cos(\pi f/f_N)}{2}\right)^n$$

$$P_k(f/f_N) + P_k(1-f/f_N) = 2$$

This final relationship defines a half-band filter. In terms of the z-transform

$$1-2x = \cos(\pi f/f_N) = \left(\frac{z+z^{-1}}{2}\right)$$

$$x = \frac{1-\cos(\pi f/f_N)}{2} = \left(\frac{1-z}{2}\right)\left(\frac{1-z^{-1}}{2}\right)$$

$$1-x = \frac{1+\cos(\pi f/f_N)}{2} = \left(\frac{1+z}{2}\right)\left(\frac{1+z^{-1}}{2}\right)$$

So

$$P_k(z) = 2\left(\frac{1+z}{2}\right)^k\left(\frac{1+z^{-1}}{2}\right)^k \sum_{n=0}^{k-1}\binom{k+n-1}{n}\left(\frac{1-z}{2}\right)^n\left(\frac{1-z^{-1}}{2}\right)^n$$

What is not obvious about this equation is that it factors into

$$P_k(z) = H(z)H(z^{-1})$$

because $P_k(z=\exp(j\pi f/f_N)) = H(\exp(j\pi f/f_N)H(\exp(-j\pi f/f_N)) = \left|H(\exp(j\pi f/f_N))\right|^2$

It is clear that $H(z)$ must have one of the following forms

$$H(z)=\left(1+z^{-1}\right)^{k} G(z^{-1}) \text{ or } H(z)=\left(1+z\right)^{k} G(z)$$

Actually, the first form is the correct one, since the terms 1/z correspond to delays in the FIR (as we have already seen in the Haar wavelet).

How do we formulate $G(z)$? We can compute this from the k-1 zeros of $B_k(x)$ with $x=(1-z)(1-z^{-1})/4$ First we find the roots of $B_k(x)$, call these x_i, then we find the roots of $x_i=(1-z)(1-z^{-1})/4$. We can rearrange the above equation resulting in $z+4x_i-2+z^{-1}=0$ which has roots $z_i=1-2x_i\pm 2\sqrt{x_i^2-x_i}$

There appears to be two values for each root (one for plus and one for minus) – but it will become clear which one we choose – in order to have a stable filter we must have $|z_i|<1$

$$H(z)=\alpha\left(1+z^{-1}\right)^{k}(1-z_1/z)(1-z_2/z)\cdots(1-z_{k-1}/z)$$

Let's illustrate this procedure for the first three members of the Daubechies wavelet family.

Starting at $k=1$, we find $B_1(x) = 1$ and $P_1(x) = 2-2x$. We can work directly on $P_1(x)$ for this simple case. Substituting $x=(1-z)(1-z^{-1})/4$ yields

$$P_1(z) = z/2+1+z^{-1}/2$$

which has a double root $z=-1\pm 0$. So

$$P_1(z) = z^{-1}(z+1)(z+1)/2 = (1+z^{-1})(1+z)/2 = H_1(z)H_1(z^{-1}).$$

It is easy to see that $H_1(z) = \alpha(1+z^{-1})$ Checking the frequency response at $z=1$ $(f/f_N=0)$ and $z=-1$ $(f/f_N=1)$ yields $H_1(1)=2\alpha$ and $H_1(1)=0$. This is a lowpass response. We want $H_1(1)=\sqrt{2}=2\alpha$, so $\alpha = 1/\sqrt{2}$. Thus:

$$H_1(z) = (1+ z^{-1})/\sqrt{2}$$
$$h_1(n) = (\delta(n)+ \delta(n-1))/\sqrt{2} = \sqrt{2}\varphi_1(n)$$

Next we do k=2. $B_2(x)=1+2x$ and $P_2(x)=2(1-x)^2(1+2x) = 2-6x^2+4x^3$. B_2 has a single root $x_1=-1/2$. So we find the roots of

$$z_1=1-2x_1\pm 2\sqrt{x_1^2-x_1}=1+1\pm 2\sqrt{1/4+1/2}=2\pm\sqrt{3}$$

We choose the smaller root because it is stable

$$H_2(z)= \alpha(1+z^{-1})^2(1-(2-\sqrt{3})z^{-1})$$
$$H_2(z)= \alpha(1+2z^{-1}+1z^{-2})(1-(2-\sqrt{3})z^{-1})$$
$$H_2(z)= \alpha(1+2z^{-1}+1z^{-2})(1-(2-\sqrt{3})z^{-1})$$
$$H_2(z)= \alpha(1+\sqrt{3}z^{-1}+(2\sqrt{3}-3)z^{-2}-(2-\sqrt{3})z^{-3}$$

Numerically, we obtain the above (without the scaling coefficient) as

$$H2 = \text{poly}([-1,-1,2-\sqrt{3}]) = [1.0000 \quad 1.7321 \quad 0.4641 \quad -0.2679]$$
$$H_2(1) = \sqrt{2} = \alpha[1.0000 + 1.7321 + 0.4641 - 0.2679] = 2.9283\alpha$$

$$\alpha = 0.4829$$

$$H_2(z) = 0.4829[1.0000 + 1.7321z^{-1} + 0.4641z^{-2} - 0.2679z^{-3}]$$
$$= 0.4829 + 0.8365z^{-1} + 0.2241z^{-2} - 0.1294z^{-3}$$
$$h_2(n) = 0.4829\delta(n) + 0.8365\delta(n-1) + 0.2241\delta(n-2) - 0.1294\delta(n-3)$$

If we scale these numbers by $\sqrt{2}$ we get the Duabechies level 2 scaling function

$$\varphi_2(n) = [0.6830, 1.1830, 0.3170, 0.1830] = \sqrt{2}h_2(n)$$

As if this wasn't enough, we do a final example for Daubechies 3.

$B_3 = [1 \ 3 \ 6]$

```
xi = roots([6 3 1])

xi = [-0.2500 + 0.3227i
      -0.2500 - 0.3227i]

x1 = -0.2500 + 0.3227i
x2 = -0.2500 - 0.3227i

z1 = 2.7127 - 1.4437i or z1 = 0.2873 + 0.1529i
z2 = 2.7127 + 1.4437i or z2 = 0.2873 - 0.1529i
```

Obviously we choose the second values, which have magnitude less than 1.

$$H(z) = \alpha\left(1+z^{-1}\right)^3 (1-(0.2873+0.1529i)/z)(1-(0.2873-0.1529i)/z)$$

Computationally, the coefficients of H(z) are found as follows:

```
H = poly([-1, -1, -1, z1, z2]);
H = [1.0000    2.4255    1.3824   -0.4058   -0.2568    0.1059]
```

To compute the normalization factor, we note that $\Sigma H(z=1) = \sqrt{2}$

```
H = H*sqrt(2)/sum(H);
H = [0.3327    0.8069    0.4599   -0.1350   -0.0854    0.0352]
```

As a function of z, the result is

$$H_3(z) = [0.3327 + 0.8069/z + 0.4599/z^2 - 0.1305/z^3 - 0.0854/z^4 + 0.0352/z^5]$$
$$h_3(n) = [0.3327\delta(n) + 0.8069\delta(n-1) + 0.4599\delta(n-2) - 0.1305\delta(n-3)$$
$$- 0.0854\delta(n-4) + 0.0352\delta(n-5)]$$

This is the more or less the same result that you would obtain from

```
[phi, psi, t] = [phi,psi,t]=wavefun('db3',1);
phi = [0 0.4705 1.1411 0.6504 -0.1909 -0.1208 0.0498 0 0 0 0]
```

except for th4e scale factor of $\sqrt{2}$, and some additional zeros that are added to the beginning and end of the impulse response.

Let's just clarify where this $\sqrt{2}$ comes from. The Daubechies level 1 (or Haar) mother scaling functions and mother wavelet are respectively [1,1] and [1,-1]. To make these orthonormal, we have to scale the amplitudes by $1/\sqrt{2}$. We looked at the frequency response of $[1,1]/\sqrt{2}$ and found that it was a lowpass filter with amplitude $\sqrt{2}$ at $f=0$. Likewise $[1,-1]/\sqrt{2}$ is a highpass filter with amplitude $\sqrt{2}$ at $f=f/f_N$. As a result, the power spectrum is scaled by a factor of 2 in each case. Scaling the power spectrum by two and halving the bandwidth has the effect of leaving power spectral density unscaled for each filter. This has the effect of conserving power (or energy) at subsequent decompositions of the DWT. For example, say we do a first-level decomposition of $x=[0,1,2,3,5,8,13,21]$ using dB1. We found

```
[cA,cD]=dwt(x,'db1');
cD = [-1 -1   -3   -8]/√2
cA = [ 1  5   13   34]/√2
```

The power in x is found as follows

```
sum(x.^2) = 713
```

Compare this to the power in cA and cD:

```
sum(cA.^2) = 675.5000
sum(cD.^2) =  37.5000
```

Obviously, the powers add to give 713. Now consider a 2nd level decomposition on both cA and cD:

```
[cAA,cAD]=dwt(cA,'db1');
sum(cAA.^2) = 561.2500
sum(cAD.^2) = 114.2500

[cDA,cDD]=dwt(cD,'db1');
sum(cDA.^2) = 31.2500
sum(cDD.^2) =  6.2500
```

At each level of the decomposition, the coefficients contain the same power as the original signal. This comes about from scaling the mother wavelet and scaling function coefficients by $1/\sqrt{2}$.

Don't forget that if you want to do the filtering as a convolution, then you need to time reverse the scaling coefficients:

```
[cA, cD]= dwt([3 5 8 13],'db1')
     cD = [-1.4142, -3.5355]

filter([-1,1]/sqrt(2),1,[3,5,8,13])
        = [-2.1213, -1.4142, -2.1213, -3.5355]
```

downsample (pick every other value, starting at second value) → [-1.4142, -3.5355]

Deriving the Wavelet from the Scaling Function.

Once you have derived the filter coefficients $h_k(n)$ for the Daubechies dBk scaling function, it is a relatively straightforward procedure to obtain the wavelet coefficients.

First you must scale $h_k(n)$ by $\sqrt{2}$ in order to obtain the mother wavelet.

$$\varphi_k(n) = h_k\sqrt{2} \qquad n=(0,1,2,3,4,..N-1) \quad \text{mathematical indexing}$$
$$n=(1,2,3,4,5,..N) \quad \text{MATLAB indexing}$$

Next $\varphi_k(n)$ is time reversed, and the sign of the odd indices (1,3,5,7) is flipped, or if you are working in mATLAB, the sign of the even indices is flipped (2,4,6,8):

$$\psi_k(n) = \varphi_k(N-1-n)(-1)^n \quad n=(0,1,2,3,4,..N-1) \quad \text{mathematical indexing}$$
$$\psi_k(n) = \varphi_k(N+1-n)(-1)^{n-1} \quad n=(1,2,3,4,5,..N) \quad \text{MATLAB indexing}$$

The above can be done concisely in MATLAB as follows:

```
phi_k = h_k*sqrt(2);
psi_k = fliplr(phi_k);
psi_k = fliplr(phi_k).*((-1).^(0:2*k-1));
```

For dB3 (k=3), the result is

```
phi_3 = [0.4705, 1.1411,  0.6504, -0.1909, -0.1208,  0.0498]
psi_3 = [0.0498, 0.1208, -0.1909, -0.6504,  1.1411, -0.4705]
```

Note there are 2k points in the scaling function and wavelet.

Zero Padding of Wavelet and Scaling Functions

When we generate the mother wavelet and mother scaling function in MATLAB, we find that there are some extra zeros padded into the data. To be precise, for dBk, MATLAB does the following padding

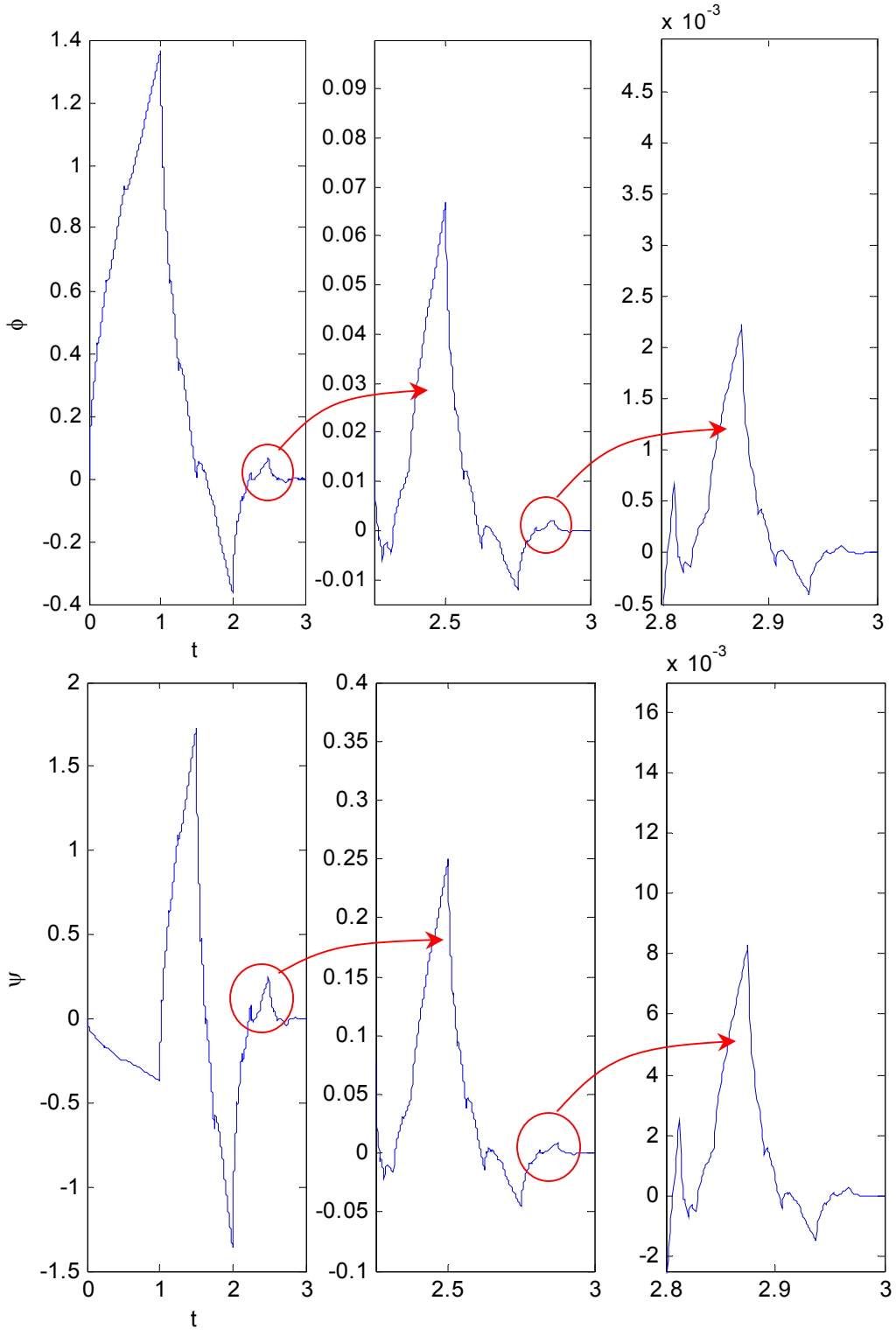
```
phi_k_padded = [0, phi_k, zeros(1,2*(k-1))];
psi_k_padded = [0, psi_k, zeros(1,2*(k-1))];
```

The total length of the padded sequence is 2k non-zero data points plus 2k-1 zeros, making for a total sequence length of 4k-1 points. The nominal time increment is 0.5 seconds, so there are (4k-2) 0.5 second intervals, or 2k-1 seconds of time representing the padded vectors.

The unpadded filter kernel is 2k (non-zero) points long, making for an impulse response duration of (2k-1) sampling intervals. Thus in MATLAB, sufficient zeros are added to make the length of the wavelet (or scaling function) in seconds be the same as the support length, assuming 0.5 second increments. This also has the effect of putting the wavelet in the center of the plot. Be aware, however, that this zero-padding is unnecessary in filter implementation.

Here is a picture of the Daubechies scaling function and wavelet of order 2, created with

```
[phi,psi,t]=wavefun('db2',10);
```



Notice the self-similarity.

Iterating the Wavelet and Scaling Functions

The above wavelet is db2 computed to 10 iterations. What does this mean?

First, we restate the unpadded scaling functions and wavelet function for db2. We call these `phi1` and `psi1` respectively; the 1 indexing the first iteration:

```
phi1 = [ 0.6830    1.1830    0.3170   -0.1830]
psi1 = [-0.1830   -0.3170    1.1830   -0.6830]
```

Next, we regard these functions as approximations that we wish to reconstruct, so we zero interpolate these functions and filter them with the `phi1` filter kernel.

There is a function in MATLAB called dyadup that does the zero interpolation quickly, but you have to be careful how you use it. We need to place zeros in the even MATLAB indices, so we must call the function with an additional even index (we choose 0, which is even)

```
phi2 = conv(dyadup(phi1,0), phi1);
      = [0.4665,  0.8080,  1.0245,  1.2745, 0.5915,
         0.1585, -0.0245, -0.2745, -0.0580, 0.0335]

psi2 = conv(dyadup(psi1,0), phi1);
      = [-0.1250, -0.2165, -0.2745, -0.3415, 0.7075,
          1.4575  -0.0915, -1.0245, -0.2165, 0.1250]
```

The process is then repeated for `phi3` and `psi3`:

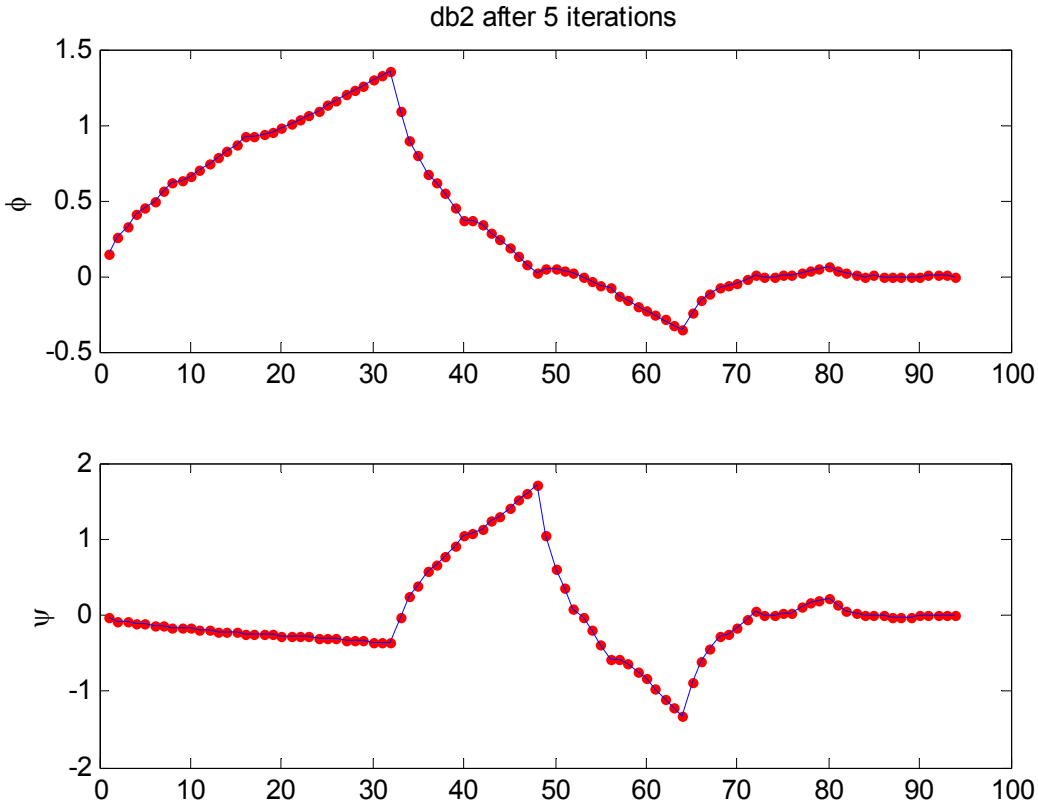
```
phi3 = conv(dyadup(phi_2,0), phi1);
psi3 = conv(dyadup(psi_2,0), phi1);
```

The following piece of MATLAB code works for any number of iterations (5 are shown)

```
M=5;
phi1 = [0.6830    1.1830    0.3170   -0.1830];
phim = phi1;
psi1 = fliplr(phi1);
psi1 = psi1.*((-1).^(0:3));
psim = psi1;

for m=2:M
    phimu = dyadup(phim,0);
    phim  = conv(phi_mu,phi1);

    psimu = dyadup(psim,0);
    psim  = conv(psimu,phi1);
end
```

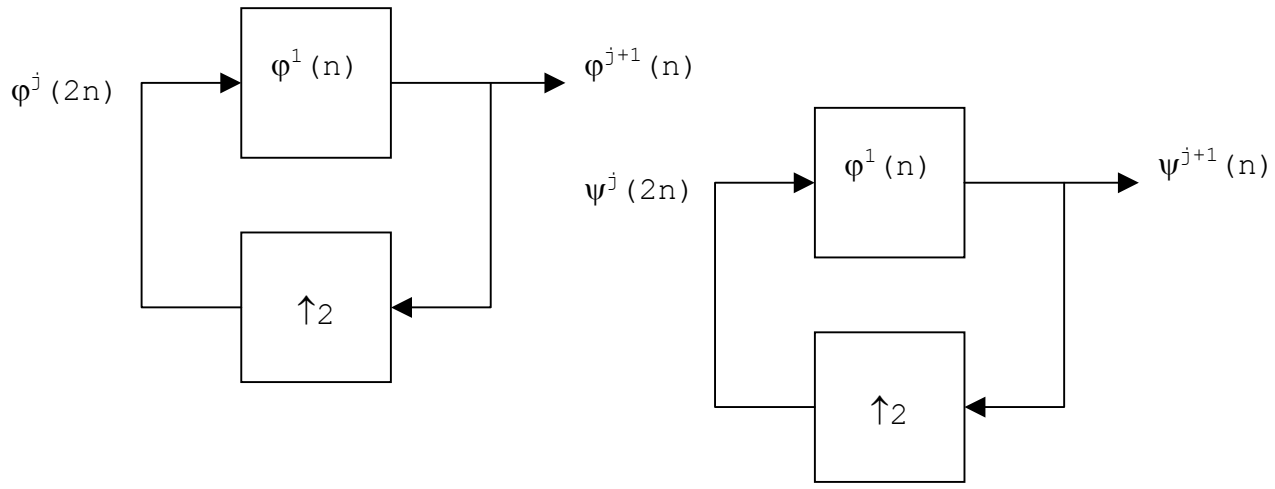
We now compare the result of the above and plot it relative to result of

```
[phi,psi,t] = wavefun('db2',5);

phi = phi(2:(length(phim)+1));      % remove zeros
psi = psi(2:(length(psim)+1));      % remove zeros
subplot(2,1,1), plot(phim,'r.'), hold on, plot(phi,'b'), hold off
subplot(2,1,2),  plot(psim,'r.'),  hold  on,  plot(psi,'b'),  hold  off
```



The data (dots) fit exactly with the results of the MATLAB function. What does this mean, other than the fact that we now seem able to reproduce the results of yet another MATLAB function?

In simple terms, the wavelet and scaling functions are produced on the jth iteration as follows:



Later, we will see how this result relates to the so called wavelet and scaling equations.

Matrix Computation of the DWT

Let's say we have a 4 sample data sequence x=[1 2 3 4] and we want to find the two-level DWT decomposition of it, suing the Haar wavelet. The quick way would be to do this:

```
[c,L]=wavedec([1 2 3 4], 2,'db1')
c = [5, -2, -0.7071, -0.7071]
L = [1, 1, 2, 4]
```

Lets do it by hand to check the result

```
x  = [1, 2, 3, 4]
cA = [1+2, 3+4]√2                    cD = [1-2, 3-4]/√2
cA = [  3,   7]√2                    cD = [ -1,  -1]/√2

cAA = [3+7]/√2/√2  cAD = [3-7]/√2/√2
cAA = 10/2         cAD = -4/2
cAA = 5            cAD = -2
```

The entire decomposition becomes

```
c = [cA | cAD, cD] = [5, -2, -1/√2, -1/√2]
```

Let's do the above symbolically and use MATLAB matrix notation. In order to make the results consistent with standard matrix algebra, we convert x from a row vector into a column vector:

```
x =   [x(1); x(2); x(2); x(4)]
cA =  [x(1)+x(2), x(3)+x(4)]√2       cD = [x(1)-x(2), x(3)-x(4)]/√2
cAA = [x(1)+x(2)+x(3)+x(4)]/√2/√2    cAD= [x(1)+x(2)-x(3)-x(4)]/√2/√2
```

In matrix notation

```
c(1) = [1;  1;  1;  1] [x(1);x(2);x(3);x(4)]/√2/√2   % cAA(1)
c(2) = [1;  1; -1; -1] [x(1);x(2);x(3);x(4)]/√2/√2   % cAD(1)
c(3) = [1; -1;  0;  0] [x(1);x(2);x(3);x(4)]/√2      % cD(1)
c(4) = [0;  0;  1; -1] [x(1);x(2);x(3);x(4)]/√2      % cD(2)

c    = [c(1); c(2); c(3); c(4)]
```

The result, in matrix notation, is also a column vector

There is a more compact way of writing this. Let's use `a=1/√2` as short hand, and indicate vectors and matrices (where the context is not clear) in boldface

$$\mathbf{c} = \mathbf{A}\mathbf{x}$$

$$\mathbf{A} = \begin{bmatrix} a^2 & a^2 & a^2 & a^2 \\ a^2 & a^2 & -a^2 & -a^2 \\ a & -a & 0 & 0 \\ 0 & 0 & a & -a \end{bmatrix}$$

Reconstruction Using the Matrix Approach to Wavelets.

If you are familiar with linear algebra, you will know that an important aspect of this mathematical discipline is solving "inverse" problems, i.e., getting back what you originally had after some transformation, known or unknown, has been applied. In our problem, it can be stated as follows:

Given a set of DWT coefficients obtained as follows

$$\mathbf{c} = \mathbf{Ax}$$

how do you get the data x back? This is easy for this particular problem: matrix multiply both sides by the inverse of A, and you get

$$\mathbf{A}^{-1}\mathbf{c} = (\mathbf{A}\mathbf{A}^{-1})\mathbf{x} = \mathbf{Ix} = \mathbf{x}$$

and you are done. As a general rule, finding matrix inverses is a computationally burdensome problem, and in many cases not necessary at all. For example, when solving a set of simultaneous equations, it is often possible to use methods such Gaussian elimination (or triangular factorization) to solve the problem without matrix inversion. MATLAB does precisely this with its backslash operator \. Say to have a matrix equation **Ax**=**b** (where **x** and **b** are column vectors), then the solution to **x** is computed as **A\B**.

Having said all that, the solution of the above equation is even simpler than Gaussian elimination. Because the columns of **A** are orthogonal, the matrix itself is said to be orthogonal and has the important property that its inverse is equal to its transpose (assuming that the elements are real). If the elements of **A** are complex and the inverse of **A** can be found by taking the transpose and complex conjugate (in either order), then it is said to be unitary.

Hence

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{c} = \mathbf{A}^{T}\mathbf{c} = \mathbf{Sc}$$

S is called the synthesis matrix. Let's check that this works for the above example. Remember, we have to work with column vectors for **x** and **c**

```
c = [5; -2; -0.7071; -0.7071]
A = [0.5000    0.5000    0.5000    0.5000
     0.5000    0.5000   -0.5000   -0.5000
     0.7071   -0.7071         0         0
          0         0    0.7071   -0.7071]

S = [0.5000    0.5000    0.7071         0
     0.5000    0.5000   -0.7071         0
     0.5000   -0.5000         0    0.7071
     0.5000   -0.5000         0   -0.7071]

S = transpose(A);

Sc = [1; 2; 3; 4] = x.
```

This is an elegant solution of the inverse problem; perhaps not readily amenable to real-time applications, but nice nonetheless. It is interesting to note that **A** can be factored into 2 matrices that represent the two levels of the decomposition:

$$\mathbf{A} = \begin{bmatrix} a^2 & a^2 & a^2 & a^2 \\ a^2 & a^2 & -a^2 & -a^2 \\ a & -a & 0 & 0 \\ 0 & 0 & a & -a \end{bmatrix} = \begin{bmatrix} a & a & 0 & 0 \\ a & -a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & a & 0 & 0 \\ 0 & 0 & a & a \\ a & -a & 0 & 0 \\ 0 & 0 & a & -a \end{bmatrix}$$

We attempt to show the reasoning behind this factoring:

$$\mathbf{Ax} = \begin{bmatrix} a & a & 0 & 0 \\ a & -a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & a & 0 & 0 \\ 0 & 0 & a & a \\ a & -a & 0 & 0 \\ 0 & 0 & a & -a \end{bmatrix} \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ x(4) \end{bmatrix}$$

$$\mathbf{Ax} = a \begin{bmatrix} a & a & 0 & 0 \\ a & -a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x(1)+x(2) \\ x(3)+x(4) \\ x(1)-x(2) \\ x(3)-x(4) \end{bmatrix}$$

$$\mathbf{Ax} = \begin{bmatrix} (x(1)+x(2)+x(3)+x(4))a^2 \\ (x(1)+x(2)-x(3)-x(4))a^2 \\ (x(1)-x(2))a \\ (x(3)-x(4))a \end{bmatrix}$$

If you hadn't already noticed, **s** (the synthesis matrix) has a form in its columns that is bears a remarkable resemblance to the scaling function and Haar wavelet. To emphasize this, we factor out a^m out of each column, where m is the level of the decomposition. This yields the so-called $\mathbf{H_4}$ matrix:

$$\mathbf{H_4} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 0 \\ 1 & -1 & 0 & 1 \\ 1 & -1 & 0 & -1 \end{bmatrix}$$

Thus we see column 1 is the Haar scaling function, and the other columns are scaled versions of the wavelet function.

We will now see how to extend this method to higher levels of decomposition. For a three level decomposition, there will be 8 values in the H matrix $\mathbf{H_8}$

$$\mathbf{H_8} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & -1 & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & -1 & 0 & 0 & -1 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & -1 & 0 \\ 1 & -1 & 0 & -1 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & -1 & 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\mathbf{a} = \begin{bmatrix} a^3 & a^3 & a^2 & a^2 & a^1 & a^1 & a^1 & a^1 \end{bmatrix}$$

$$\mathbf{S} = \begin{bmatrix} a^3 & a^3 & a^2 & 0 & a^1 & 0 & 0 & 0 \\ a^3 & a^3 & a^2 & 0 & -a^1 & 0 & 0 & 0 \\ a^3 & a^3 & -a^2 & 0 & 0 & a^1 & 0 & 0 \\ a^3 & a^3 & -a^2 & 0 & 0 & -a^1 & 0 & 0 \\ a^3 & -a^3 & 0 & a^2 & 0 & 0 & a^1 & 0 \\ a^3 & -a^3 & 0 & a^2 & 0 & 0 & -a^1 & 0 \\ a^3 & -a^3 & 0 & -a^2 & 0 & 0 & 0 & a^1 \\ a^3 & -a^3 & 0 & -a^2 & 0 & 0 & 0 & -a^1 \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} a^3 & a^3 & a^3 & a^3 & a^3 & a^3 & a^3 & a^3 \\ a^3 & a^3 & a^3 & a^3 & -a^3 & -a^3 & -a^3 & -a^3 \\ a^2 & a^2 & -a^2 & -a^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a^2 & a^2 & -a^2 & -a^2 \\ a^1 & -a^1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a^1 & -a^1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a^1 & -a^1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a^1 & -a^1 \end{bmatrix}$$

The elements of `a` multiply each column of $\mathbf{H_8}$. To do this in MATLAB, you would need to do something like

```
N=8;
av = (a*ones(1,N)).^[3,3,2,2,1,1,1,1];
am = (av'*ones(1,N))';
S  = H_N.*am;
A  = transpose(S);
```

The automatic computation of H_N and the vector `[3,3,2,2,1,1,1,1]` are left as exercises.

Upsampling and Downsampling.

An important part of the wavelet decomposition is to downsample (decimate by 2) and upsample (interpolate by 2). We should understand what is happening, particularly in the frequency domain, when we do this process. Downsampling takes a sequence

```
[x(1), x(2), x(3), x(4) … ]
```

and converts it into

```
[x(1), x(3), x(5), …]
```

We simply throw away every other data point. As a matrix operation, we would write

$$\begin{bmatrix} y(1) \\ y(2) \\ y(3) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \end{bmatrix}$$

$$y(n) = D_2 x(n)$$

Likewise upsampling can be written as

$$\begin{bmatrix} y(1) \\ y(2) \\ y(3) \\ y(4) \\ y(5) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x(1) \\ x(2) \\ x(3) \end{bmatrix}$$

$$y(n) = U_2 x(n)$$

Note that U2 and D2 are transposes of each other $D_2 = U_2^T$

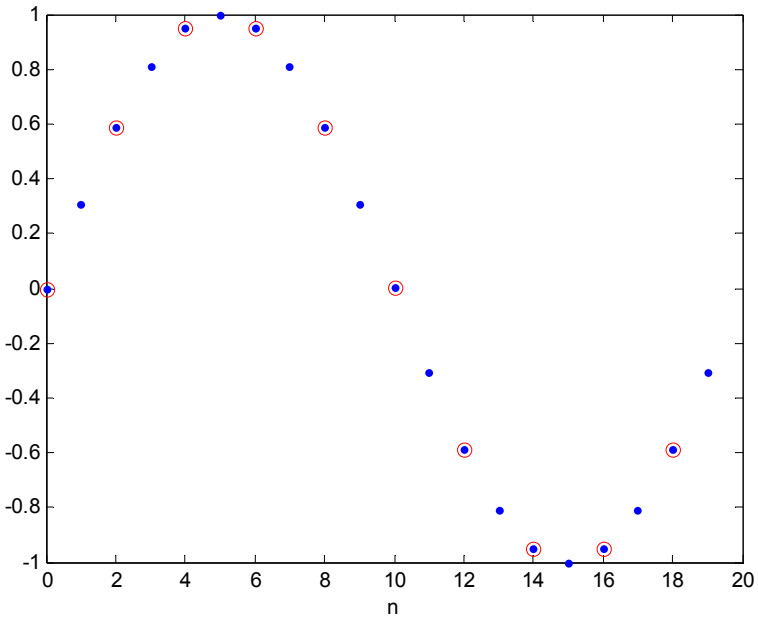
Upsampling followed by downsampling gives $D_2 U_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

However, downsampling followed by upsampling gives

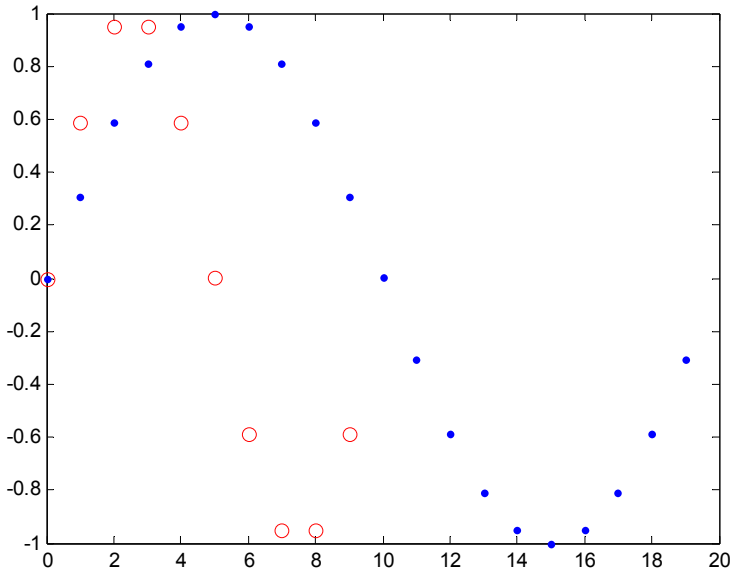
$$U_2 D_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Frequency Domain Consequences of Subsampling

Down sampling increases frequency. First we take every other point (every 2Ts)



But then we represent those sub-sampled points on a new time axis with 1Ts increments



The result is a doubling of the frequency

Let's analyze the frequency response of the downsampled signal

$$y(n) = D_2 x(n) = x(2n) \qquad n = 0,1,2,3\ldots$$

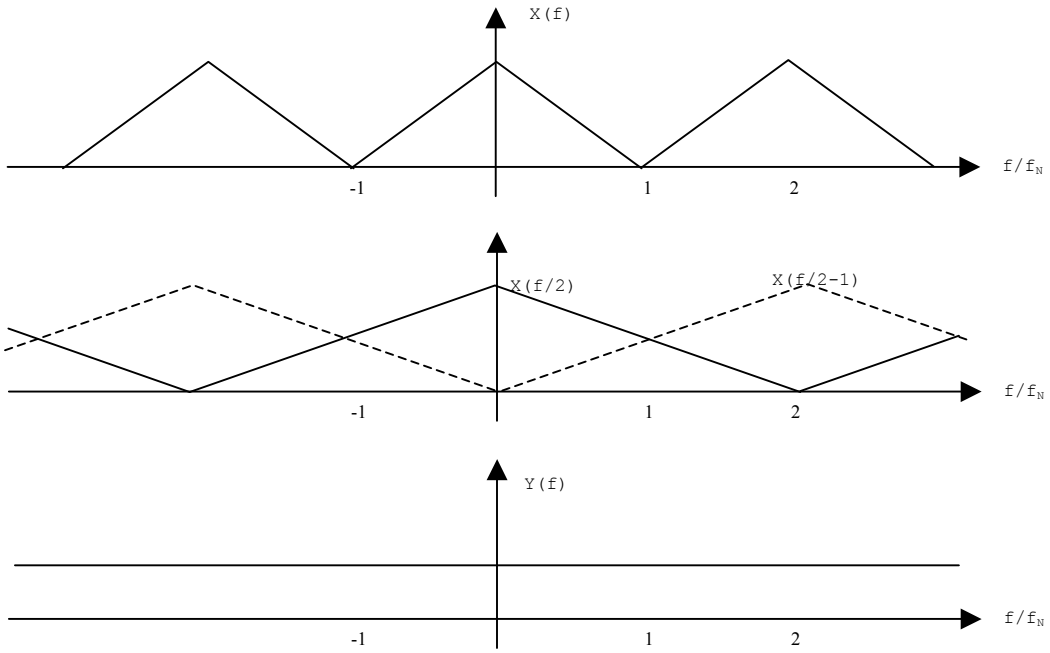
$$Y(f) = \sum_n y(n) e^{-jn\pi f} = \sum_n x(2n) e^{-jn\pi f} = [X(f/2) + X(f/2-1)]/2$$

where it is understood that frequency f is normalized with respect to the Nyquist frequency.

A more general result for downsampling to every M[th] sample (via D_M) is

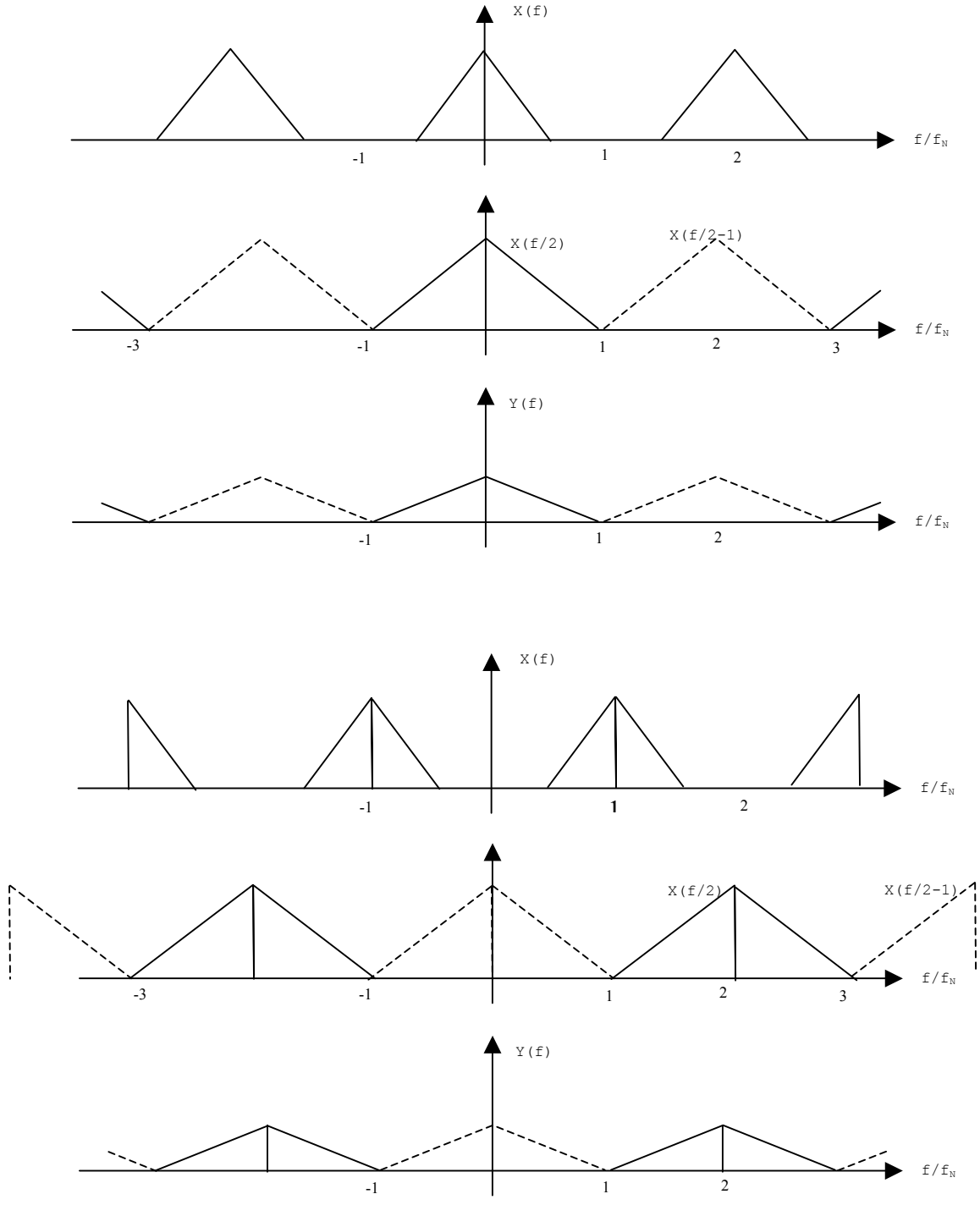
$$Y(f) = \frac{1}{M} \sum_{m=0}^{M-1} X([f-2m]/M)$$

You have to be careful when you downsample something. This makes sense from a Nyquist point of view, if the sampling frequency is 8kHz, when the Nyquist frequency is 4 kHz. But if you downsample a signal that has more than 2kHz of bandwidth, you will get aliasing. A simple example shows why

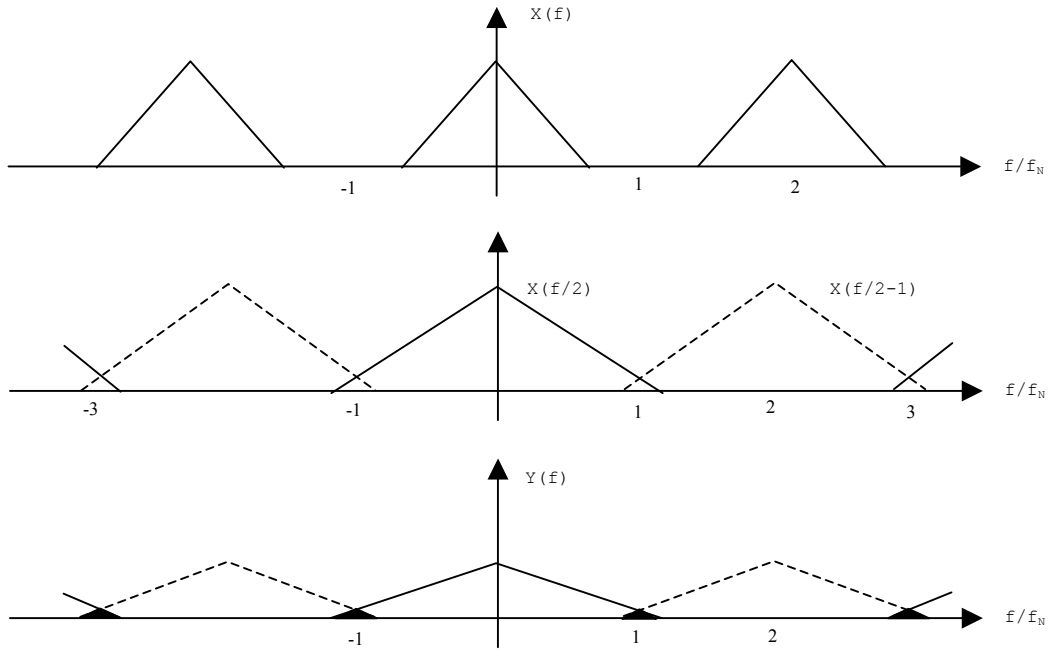


Note that while X(f) has period 2 (i.e., $f=2f_N=f_S$), X(f/2) has period 4 (i.e. $f=4f_N=2f_S$)

The resulting signal is a constant in the frequency domain (i.e. an impulse). However if the signal is low pass bandlimited to half the Nyquist frequency, we do not get the above problem (aliasing).



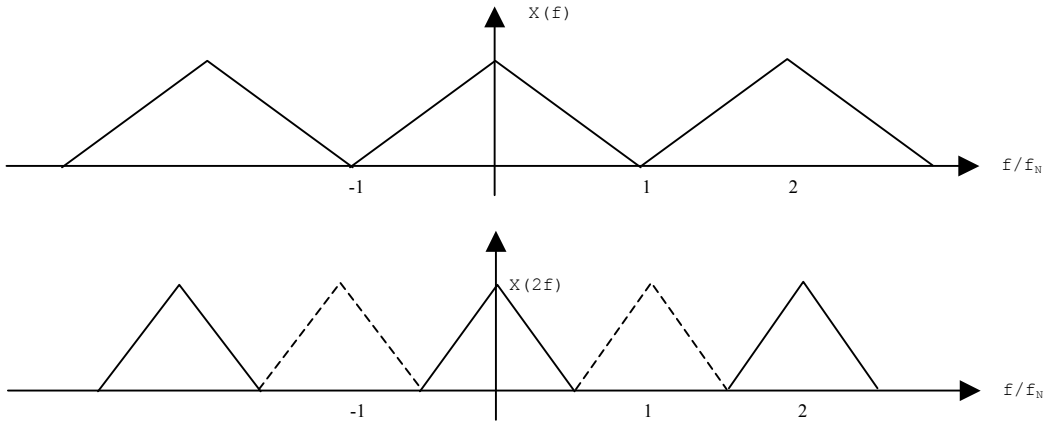
In practice, we note that the quadrature mirror filters do not completely limit the signal to half of the Nyquist frequency. There is some residual aliasing, but despite this, it is possible (as we have seen) to perfectly reconstruct the signal. This situation is shown for a lowpass response below:



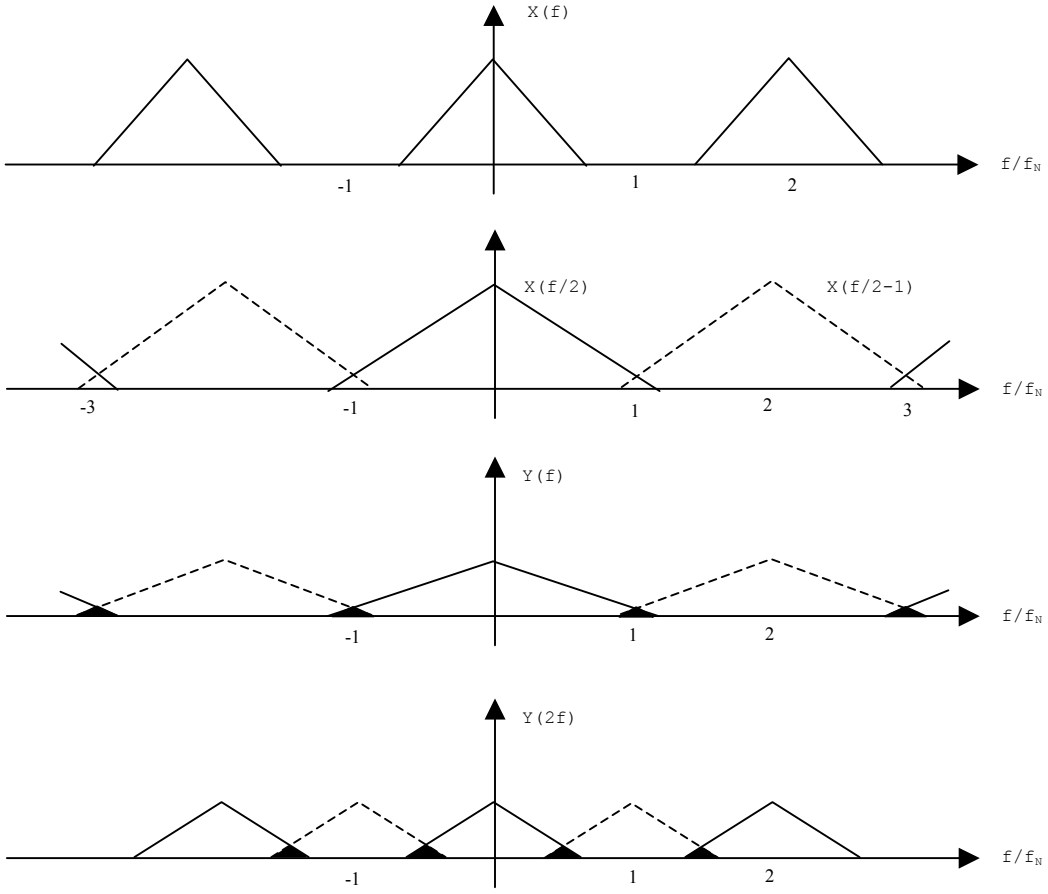
The frequency response of an upsampled signal is simply

$$y(n) = U_2 x(n)$$
$$Y(f) = X(2f)$$

However, where X(f) has period 2, X(2f) has period 1. This leads to a process known as imaging. It is illustrated below (the images are shown dotted).



As we have also seen, upsampling followed by downsampling leaves the original sequence, but downsampling followed by updampling (which is what happens in the reconstruction process) makes every other data point zero. Let us look at this in the context of the slightly aliased lowpass response above:



The net result of downsampling followed by upsampling is to produce a spectrum given by

$$Z(f) = [X(f) + X(f-1)]/2$$

Now at this point we have a signal `z(n) = [x(0), 0, x(2), 0, x(4), 0 …]` (using regular DSP indexing rather than MATLAB indexing). We can actually recover the original sequence `[x(0),x(1), x(2),x(3)…]` if it is bandlimited to less than $f_N/2$ by lowpass filtering with a filter of bandwidth equal to $f_N/2$. Since we have exceeded the $f_N/2$ limit, we will have aliasing if we were to apply the recovery filter.

However, the complementary action of the quadrature mirror filter causes the aliasing in the low-pass response to cancel with the aliasing in the high-pass response, leading to the exact original sequence.

The Dilation and Wavelet Equation

We saw previously how a wavelet and scaling function can be iterated. Let's take a deeper look at this process.

First, we state the dilation equation:

$$\varphi(t) = \sum_m c_m \varphi(2t - m) \qquad m = 0, 1, \ldots, M-1$$

and the wavelet equation

$$\psi(t) = \sum_m d_m \varphi(2t - m) = \sum_m (-1)^m c_{M-1-m} \varphi(2t - m)$$

and justify them on the basis that the wavelet and scaling functions are nothing more than scaled and shifted versions of a certain prototype scaling function.

Next we find some constraints on c_m. Integrating both sides of the first equation with respect to t yields

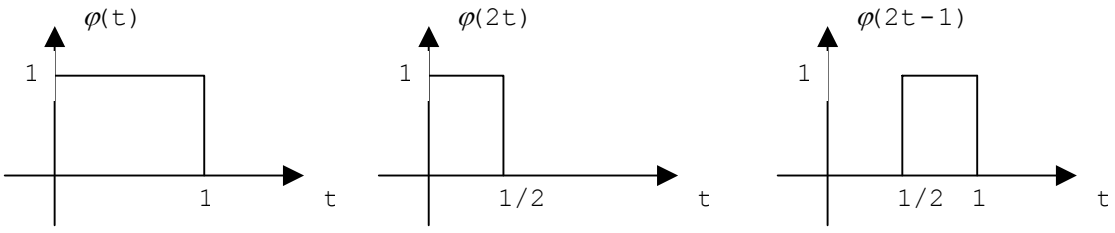
$$\int \varphi(t)\, dt = \int \sum_m c_m \varphi(2t - m)\, dt = \sum_m c_m \int \varphi(2t - m)\, dt = \sum_m c_m \int \varphi(x)\, dx/2$$

Note that in the last integral, we made the substitution $2t-m=x$, $dx/dt=2$, $dt=dx/2$. Canceling the integral terms, we find

$$\sum_m c_m = 2$$

At this point, we can start to make up some values for c_m. The simplest possible choice is one value ($m=0$), leading to $c_0 = 2$. Thus we need a function such that $\varphi(t) = 2\varphi(2t)$, and it turns out that the only function that satisfies this requirement is a delta function: $\varphi(t) = 2\delta(2t)$. To see why this is, you must integrate both sides with respect to t. This is not a familiar result – let's look at one that is. Choose $c_0 = c_1 = 1$. This leads to the Haar Wavelet scaling function

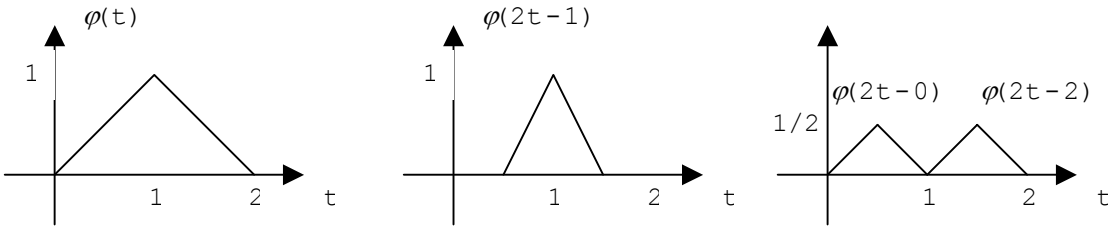
$$\varphi(t) = \varphi(2t) + \varphi(2t-1)$$



Another simple function arises from $c_0 = c_2 = 1/2, \quad c_1 = 1$. This leads to a triangular scaling function that is sometimes called a hat. The hat is a bit trickier to understand, so let's go through it step by step. First we show the scaling function from the dilation equation

$$\varphi(t) = 0.5\varphi(2t-0) + \varphi(2t-1) + 0.5\varphi(2t-2)$$

Next we show the scaling function and the three scaled and shifted versions of it:



Clearly, the two figures to the right add up to the figure on the left. What is not clear is how we came up with $\varphi(t)$ in the first place. Actually, we have seen this process before as an iteration on the original scaling function coefficients [1/2, 1,1/2], which appear to form a crude triangle. More on this later.

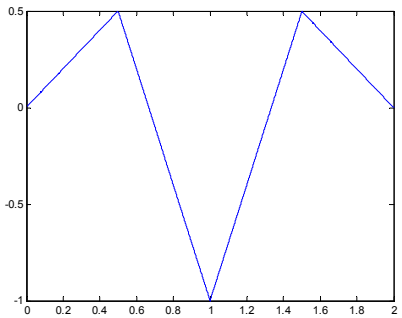
Are you beginning to see the self-similarity that we saw before? In these examples, it is rather simple: you make one big triangle out of three smaller ones. The scaling function was obtained previously by recursion; here is a quick way of obtaining the triangle function

```
c0=[1/2,1,1/2]; x=c0    = [0.50 1.00 0.50]
x=conv(dyadup(x,0),c0) = [0.25 0.50 0.75 1.00 0.75 0.50 0.25]
x=conv(dyadup(x,0),c0) = [0.125 0.25 0.375 0.50 0.625 0.75 0.875 1.00 0.875
                          0.75 0.625 0.50 0.375 0.25 0.125], … etc.
```

In other words, the process previously described whereby the scaling function and the wavelet are derived from a repetitive sequence of upsampling and convolution (with filter coefficients c_m) is mathematically described by the dilation equation and the wavelet equation respectively.

Let's see what the wavelet looks like for the filter coefficients $c_0 = c_2 = 1/2, \quad c_1 = 1$.

```
d0=[1/2,-1,1/2]; y=d0   = [0.50 -1.00 0.50]
y=conv(dyadup(y,0),c0) = [0.25  0.50 -0.25 -1.00 -0.25 0.50 0.25]
y=conv(dyadup(y,0),c0) = [0.125 0.25 0.375 0.50 0.125 -0.25 -0.625 -1.00
                         -0.625 -0.25 0.125 0.50 0.375 0.25 0.125] … etc.
```



A similar constraint on the integral of $\psi(t)$ (being zero) leads to $\sum_m d_m = 0$

Now we revisit Daubechies db2 scaling functions and wavelets. Previously we had called these φ^1 and ψ^1 and had generated them using the MATLAB command `wavefun`:

```
[phi1, psi1,t] = wavefun('db2',1);
phi1 = [ 0.6830    1.1830    0.3170   -0.1830]
psi1 = [-0.1830   -0.3170    1.1830   -0.6830]
t    = [0 0.5 1.0 1.5 2.0 2.5 3.0]
```

The db2 wavelet has a support length of 3, hence t extends from 0-3. Notice that `sum(phi1) = 2` and `sum(psi1) = 0`, so `phi1` and `psi1` are actually the coefficients c_m and d_m in the dilation and scaling equation, respectively. Let's apply the dilation equation to the db2 scaling function for integer values of t (0,1,2,3,4). For berevity, we denote

`phi1` = φ^1 = `[ c0 c1 c2 c3 ]`

$$\varphi(0) = c_0\varphi(0) + c_1\varphi(-1) + c_2\varphi(-2) + c_3\varphi(-3)$$
$$\varphi(1) = c_0\varphi(2) + c_1\varphi(1) + c_2\varphi(0) + c_3\varphi(-1)$$
$$\varphi(2) = c_0\varphi(4) + c_1\varphi(3) + c_2\varphi(2) + c_3\varphi(1)$$
$$\varphi(3) = c_0\varphi(6) + c_1\varphi(5) + c_2\varphi(4) + c_3\varphi(3)$$

However, $\varphi(t)$ is zero outside its support range (0<t<3), so $\varphi(0)$ and $\varphi(3)$ are both 0, as are $\varphi(-1)$, etc. The above reduces to

$$\varphi(1) = c_0\varphi(2) + c_1\varphi(1)$$
$$\varphi(2) = c_2\varphi(2) + c_3\varphi(1)$$

from which following MATRIX equation can be derived.

$$\begin{bmatrix}\varphi(1)\\ \varphi(2)\end{bmatrix} = \begin{bmatrix}c_1 & c_0\\ c_3 & c_2\end{bmatrix}\begin{bmatrix}\varphi(1)\\ \varphi(2)\end{bmatrix}$$

This is an eigenvalue problem, which we solve numerically in MATLAB as follows

```
C    =  [1.1830    0.6830;   -0.1830    0.3170]       %[c1, c0; c3, c2]
[V,D]=  eig(C)
V    =  [0.9659   -0.7071
        -0.2588    0.7071]
D    =  [1, 0.5]
```

There are two eigenvalues (1 and 0.5), and two corresponding eigenvectors `[0.9659; -0.2588]` and `[-0.7071; 0.7071]`. However, these eigenvectors have a norm of 1. We multiply these eigenvectors by $\sqrt{2}$ so that the norm is 2, resulting in

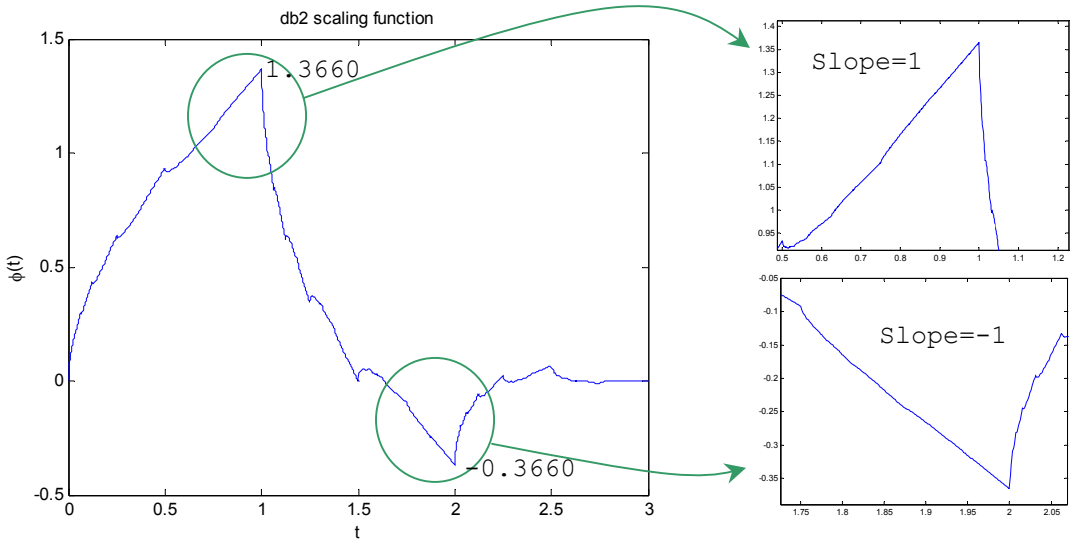
```
V√2  =  [1.3660   -1.0000
        -0.3660    1.0000]
```

The first eigenvector corresponds to $\varphi(t)$ at the integer values 1 and 2.

$$\begin{bmatrix}\varphi(1)\\ \varphi(2)\end{bmatrix} = \begin{bmatrix}1.3660\\ -0.3660\end{bmatrix} = \begin{bmatrix}0.6830 & 1.1830\\ 0.3170 & -0.1830\end{bmatrix}\begin{bmatrix}1.3660\\ -0.3660\end{bmatrix}$$

The second eignevalue corresponds to derivatives of the scaling function.

If you look closely at $\varphi(t)$ for db2, you will see that these integer values are accurately depicted.



Two Dimensional Wavelet Processing

Perhaps the easiest way to introduce 2-D wavelets is to do a simple example. We use a special matrix (for no particular reason other than it is simple to generate) called a magic matrix.

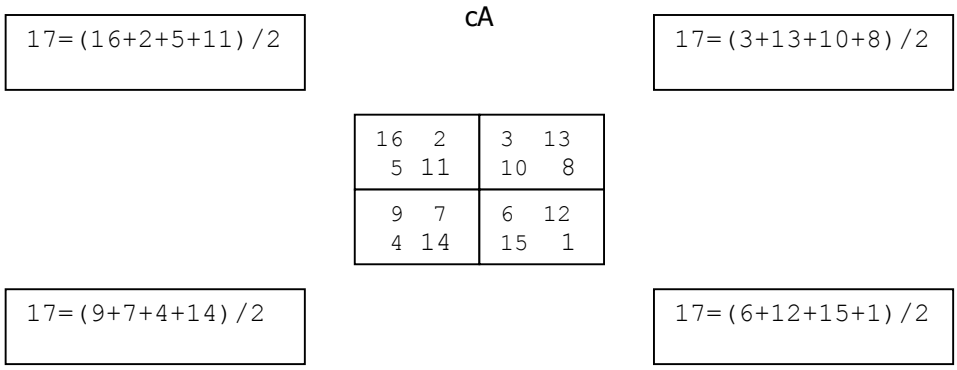
```
Y=magic (4)

Y = 16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

The 'magic' quality of this matrix is that all rows and columns sum to 34, as do two major diagonals.

A 2D decomposition produces 4 sets of coefficients. Again, we choose the dB1 wavelet because it is simple to illustrate. Before doing the DWTs, we set the extension mode to zero-padding. This determines how the convolution affects data at the end and beginning of the arrays.

```
dwtmode('zpd')
[cA,cH,cV,cD] = dwt2(Y,'db1')

cA =  17.0000   17.0000
      17.0000   17.0000

cH =   1.0000   -1.0000
      -1.0000    1.0000

cV =   4.0000   -4.0000
      -4.0000    4.0000

cD =  10.0000   -6.0000
       6.0000  -10.0000
```

The 2D DWT produces one set of approximation coefficients (cA) and 3 sets of detail coefficients (cH, the horizontal details, cV, the vertical details, and cD, the diagonal details). First lets look at the approximations:



cH

16 2 5 11	3 13 10 8
9 7 4 14	6 12 15 1

1=(16+2-5-11)/2

-1=(3+13-10-8)/2

-1=(9+7-4-14)/2

1=(6+12-15-1)/2

cV

16 2 5 11	3 13 10 8
9 7 4 14	6 12 15 1

4=(16-2+5-11)/2

-4=(3-13+10-8)/2

-4=(9-7+4-14)/2

4=(6-12+15-1)/2

cD

16 2 5 11	3 13 10 8
9 7 4 14	6 12 15 1

10=(16-2-5+11)/2

-6=(3-13-10+8)/2

6=(9-7-4+14)/2

10=(6-12-15+1)/2

A pattern begins to emerge in terms of elementwise multiplication sub-arrays

For cA the sub-array is $\begin{bmatrix} +1 & +1 \\ +1 & +1 \end{bmatrix}/2$

for cH the sub-array is $\begin{bmatrix} +1 & +1 \\ -1 & -1 \end{bmatrix}/2$

for cV the sub-array is $\begin{bmatrix} +1 & -1 \\ +1 & -1 \end{bmatrix}/2$

for cD the sub-array is $\begin{bmatrix} +1 & -1 \\ -1 & +1 \end{bmatrix}/2$

We see the basic elements of the Haar scaling function and wavelet appearing in the multiplication arrays, but the process is not clear. We need to understand that what is going on here is an averaging in both the x and y directions. We are dividing by 2 (and not root 2) because of this double filtering. Let's look at the entire process in detail (perhaps overly tedious detail).

We start with a row-by-row convolution with the scaling function:

```
[16     2     3    13 ]⊗[1 1]/√2 = [16 18  5 16 13]/√2
[ 5    11    10     8 ]⊗[1 1]/√2 = [ 5 16 21 18  8]/√2
[ 9     7     6    12 ]⊗[1 1]/√2 = [ 9 16 13 18 12]/√2
[ 4    14    15     1 ]⊗[1 1]/√2 = [ 4 18 29 16  1]/√2
```

Next we down sample every other datapoint, starting at the second datapoint:

```
[16     2     3    13 ]⊗[1 1]/√2 ↓2 = [18 16 ]/√2
[ 5    11    10     8 ]⊗[1 1]/√2 ↓2 = [16 18 ]/√2
[ 9     7     6    12 ]⊗[1 1]/√2 ↓2 = [16 18 ]/√2
[ 4    14    15     1 ]⊗[1 1]/√2 ↓2 = [18 16 ]/√2
```

Then we apply the same filter function in the vertical direction

```
[18;   16;   16;   18 ]/√2⊗[1;1]/√2  = [18; 34; 32; 34; 18]/2
[16;   18;   18;   16 ]/√2⊗[1;1]/√2  = [16; 34; 36; 34; 16]/2
```

And then down-sample

```
[18; 34; 32; 34; 18]/2 ↓2  = [34; 34]/2 = [17; 17]
[16; 34; 36; 34; 16]/2 ↓2  = [34; 34]/2 = [17; 17]
```

So we now know how to compute the approximation coefficients, at least for db1. Actually, the result we have is the transverse of the detail coefficients. It is not easy to see this in the current example, because the matrix is symmetric.

Let's repeat this process for the vertical detail coefficients. First we filter each row with the wavelet as if it were the input to a 1D DWT

```
[16     2     3    13 ]⊗[-1 +1]/√2 = [-16 14  -1 -10  13]/√2
[ 5    11    10     8 ]⊗[-1 +1]/√2 = [-5  -6   1   2   8]/√2
[ 9     7     6    12 ]⊗[-1 +1]/√2 = [-9   2  -1  -6  12]/√2
[ 4    14    15     1 ]⊗[-1 +1]/√2 = [-4 -10  -1  14   1]/√2
```

downsample

```
[16     2     3    13 ]⊗[-1 +1]/√2 ↓2 = [ 14  -10 ]/√2
[ 5    11    10     8 ]⊗[-1 +1]/√2 ↓2 = [ -6   2 ]/√2
[ 9     7     6    12 ]⊗[-1 +1]/√2 ↓2 = [  2  -6 ]/√2
[ 4    14    15     1 ]⊗[-1 +1]/√2 ↓2 = [-10  14 ]/√2
```

then filter each column with the scaling function, as though it were in the input to a 1D DWT

```
[ 14;  -6;    2;  -10 ]/√2⊗[+1;+1]/√2  = [ 14;  8;  -4; -8;  -10]/2
[-10;   2;   -6;   14 ]/√2⊗[+1;+1]/√2  = [-10; -8;  -4;  8;   14]/2
```

```
[ 14;   8; -4; -8; -10]/2 ↓2  = [ 8;  -8]/2 = [ 4; -4]
[-10;  -8; -4;  8;  14]/2 ↓2  = [-8;   8]/2 = [-4;  4]
```

The result is actually transverse of the real result.

The computation on the horizontal coefficients proceeds in more or less the same way, except the filtering is done on the columns first:

```
[16;    5;    9;     4 ]⊗[-1; +1]/√2 ↓2 = [11;  5]/√2
[ 2;   11;    7;    14 ]⊗[-1; +1]/√2 ↓2 = [-9; -7]/√2
[ 3;   10;    6;    15 ]⊗[-1; +1]/√2 ↓2 = [-7; -9]/√2
[13;    8;   12;     1 ]⊗[-1; +1]/√2 ↓2 = [ 5; 11]/√2

[11 -9 -7  5]√2⊗[+1 +1]/√2 ↓2 = [ 1 -1]
[ 5 -7 -9 11]√2⊗[+1 +1]/√2 ↓2 = [-1  1]
```

Note: this result is not transverse.

The diagonal coefficients use the highpass coefficients in both directions:

```
[16     2     3    13 ]⊗[-1 +1]/√2 ↓2 = [ 14 -10]/√2
[ 5    11    10     8 ]⊗[-1 +1]/√2 ↓2 = [ -6   2]/√2
[ 9     7     6    12 ]⊗[-1 +1]/√2 ↓2 = [  2   6]/√2
[ 4    14    15     1 ]⊗[-1 +1]/√2 ↓2 = [-10  14]/√2

[ 14;  -6;    2;  -10 ]/√2⊗[-1;+1]/√2 ↓2  = [10;   6]
[-10;   2;   -6;   14 ]/√2⊗[-1;+1]/√2 ↓2  = [-6; -10]
```

The result is again the transverse of what we obtained from dwt2 (with zero padding extension mode).

Its interesting that the magic matrix produces approximations and details in H and V that are essentially scaled versions of the wavelet and scaling functions. It does this for other magic matrices (e.g. magic(8). This would suggest that the magic matrix is some sort of eigenmatrix, i.e., a matrix of eigenvectors, the vectors being the wavelet and the scaling functions, and the scale values being the eigenvalues.

At this point, it would be useful to automate this process in MATLAB. I'll show two methods. The first method uses the function `convn`. This works very much like regular convolution (`conv`) except the arguments can be matrices rather than vectors. Assume that the first argument is a matrix of data, and the second argument is a filter kernel (a row vector). Then `convn` convolves each row in the matrix with the filter kernel, thus shortening the separate convolution process done above. If the kernel is a column vector, then the convolution is done along columns.

We also use `dyaddown` as a short cut to taking every other point out of the filtered data (to down sample).

```
phi=[1 1];                    % define the scaling coefficient
psi=[1 -1];                   % and the mother wavelet
c = fliplr(phi);              % define the analysis lowpass filter
d = fliplr(psi);              % define the analysis highpass filter

yAa = convn(Y,c);             % LPF across image
yAb = dyaddown(yAa);          % downsample
yAc = convn(yAb,c');          % LPF down image
yA  = dyaddown(yAc')'/2;      % downsample, transpose, and scale

yHa = convn(Y',d);            % HPF down image
yHb = dyaddown(yHa);          % downsample
yHc = convn(yHb,c');          % LPF across image
yH  = dyaddown(yHc')/2;       % downsample and scale

yVa = convn(Y,d)              % HPF across image
yVb = dyaddown(yVa)           % downsample
yVc = convn(yVb,c')           % LPF down image
yV  = dyaddown(yVc')'/2       % downsample, transpose, and scale

yDa = convn(Y,d);             % HPF across image
yDb = dyaddown(yDa);          % downsample
yDc = convn(yDb,d');          % HPF down image
yD  = dyaddown(yDc')'/2;      % downsample, transpose, and scale

% note, transposing using ' assumes all data are real. For complex
data, use transpose.
```

The second method combines the two filtering operations into a single n-dimensional convolution by calculating a 2D filter kernel as the outer product of the horizontal and vertical kernels:

```
yA = dyaddown(dyaddown(convn(Y, c'*c))')'/2;
yH = dyaddown(dyaddown(convn(Y, d'*c))') /2;
yV = dyaddown(dyaddown(convn(Y',d'*c))') /2;
yD = dyaddown(dyaddown(convn(Y, d'*d))')'/2
```

At this point, if you know how to implement convolution, then you know how to do 2D wavelet decomposition. The next thing we need to know is image reconstruction. Then we will look at some other forms of image extension (dealing with borders). MATLAB, by default, assumes a symmetricization of the data as reflected about the border. This will produce different results from the hand calculations because these generally assume zero padding beyond the border. More on this later.

Dealing with edges, and odd numbers of data points.

First we deal with the issue of data points with specific reference to the dB2 wavelet. Given an input data set of length n, the resulting number of coefficients in cA is $\lfloor (n+1)/2 \rfloor + 1$, and the number of coefficients in cD is the same. For large n, this is approximately n/2 in each set, making for a total of approximately n coefficients total, but we see that the result is somewhat different for small n.

For example, suppose we start with 8 data points and do a dB2 decomposition. We end up with $\lfloor (8+1)/2 \rfloor + 1 = \lfloor 4.5 \rfloor + 1 = 4+1 = 5$ coefficients for cA and 5 for cD (making a total of 10). If we do a further decomposition on cA we get cAA with $\lfloor (5+1)/2 \rfloor + 1 = \lfloor 3 \rfloor + 1 = 4$ data points. So cAA, cAD, and cD would have 4+4+5=13 data points. Clearly we are not conserving data when the number of input points becomes small, but this is a special case. For a single decomposition, we find that the number of coefficients is equal to the number of input data points plus 2 or 3. For large numbers the 2 or 3 extra points are negligible. Vut where do these 2 or 3 points come from?

To answer that question, we will need to look carefully at how the convolution (or filtering) is done. For the sake of simplicity, let's assume an input string of [1 1 1 1 1] and a db2 scaling function of phi=[0.6830 1.1830 0.3170 -0.1830]. The filter kernel is [-0.1294 0.2241 0.8365 0.4830] (phi flipped and scaled by 1 over root 2. First we compute the approximation coefficients assuming the ends are zero padded:

```
x =  [1 1 1 1 1];
dwtmode('zpd')
[cA1,cD1]=dwt(x,'db2');
cA1 = [0.0947   1.4142   1.5436   0.4830]
```

Then we pad the data at both ends with 3 zeros

```
xe = [0 0 0 1 1 1 1 1 0 0 0];
```

Then convolve with the filter kernel

```
ca1 = conv(xe,[-0.1294 0.2241 0.8365 0.4830]);
ca1 = [0 0 0
      -0.1294 0.0947 0.9313 1.4142 1.4142 1.5436 1.3195 0.4830
       0 0 0]
```

Lets look at this result before we go further. The padded zeros are retained in the MATLAB convolution, so they are shown separately, but the purpose of the zeros is to get the convolution 'started'. Defining the flipped filter kernel as h

```
h= [0.4830    0.8365    0.2241   -0.1294]

xe=[ 0  0  0  1  1  1  1  1  0  0  0 ];

1st [h1 h2 h3 h4]                          = h4 =              -0.1294
2nd    [h1 h2 h3 h4]                       = h3+h4 =            0.0947
3rd       [h1 h2 h3 h4]                    = h2+h3+h4 =         0.9313
4th          [h1 h2 h3 h4]                 = h1+h2+h3+h4 =      1.4142
5th             [h1 h2 h3 h4]              = h1+h2+h3+h4 =      1.4142
6th                [h1 h2 h3 h4]           = h1+h2+h3 =         1.5436
7th                   [h1 h2 h3 h4]        = h1+h2 =            1.3195
8th                      [h1 h2 h3 h4]     = h1 =               0.4830
```

Now when we do downsampling (highlighted in red above), we keep the even data points, assuming an index that starts at 0. We can do this in MATLAB using the dyaddown function with an argument of 1 (which selects the odd MATLAB indices, which correspond to the even DSP indices).

The purpose of doing the extension is to provide a way of dealing with the start and end of the data stream. This is especially important in image processing, because the 'discontinuity' in the image at its edges is an artificial truncation introduced by the image acquisition system. In many cases, it is useful to be able to ameliorate this truncation, at least as it affects filtering, and various methods of data extension are used to facilitate this. The most common of these method is symmetricization. This is simply a reflection of the data at its end points. Its important to understand how this process works. For a filter kernel of length 4, the extension has length 3. Take the first 3 data points, flip them, then add these to the beginning of the array. Repeat this process at the end.

```
x =          [  1  1  1  1  1 ];
x =          [ x1 x2 x3 x4 x5 ];
xe=  [x3 x2 x1 x1 x2 x3 x4 x5 x5 x4 x3];
```

Let's repeat the above example with symmetricization:

```
dwtmode('sym')
[cA2,cD2]=dwt(x,'db2');
cA2 = [1.4142 1.4142 1.4142 1.4142]

xe = [1 1 1 1 1 1 1 1 1 1 1];
ca2= conv(xe,[-0.1294 0.2241 0.8365 0.4830]);
ca2= [-0.1294 0.0947 0.9313
       1.4142 1.4142 1.4142 1.4142 1.4142 1.4142 1.4142 1.4142
       1.5436 1.3195 0.4830]
```

Notice that the coefficients are now uniform – this seems to better reflect the uniform nature of the initial data. We also find that the detail coefficients are all zero when symmetricization is used – whereas with zero padding the detail coefficients are non zero.

The question now arises as to whether extension affects reconstruction. Let's revisit the process of reconstruction as it relates to extension. The process is remarkably simple, but produces an error. First we start with the approximation and detail coefficients

```
cA2 =   [ 1.4142  1.4142  1.4142  1.4142]
cD2 =   [-0.5551 -0.5551 -0.5551 -0.5551]*1.0e-016
```

The detail coefficients are essentially zero, so we ignore them in the reconstruction. Next we zero interpolate:

```
cA2u = dyadup(cA2,0)
cA2u = [1.4142 0 1.4142 0 1.4142 0 1.4142]
xr =   conv(cA2u,phi/sqrt(2))
   =   [0.6830  1.1830
        1.0000  1.0000 1.0000 1.0000 1.0000 1.0000
        0.3170 -0.1830]
```

As indicated earlier, we remove the first 2(k-1) data points (where k is the order of the dBk kernel) and also the last 2(k-1) points. This yields

```
xr =   [1.0000  1.0000 1.0000 1.0000 1.0000 1.0000]
```

Notice now that the recovered signal has one extra data point!. This is because we started with an odd number of data points. Another way of looking at this result is to repeat the process when the input data contains an even number of data points. Let `x=[1 1 1 1 1 1]`. Then

```
cA2 =    [1.4142 1.4142 1.4142 1.4142]
```

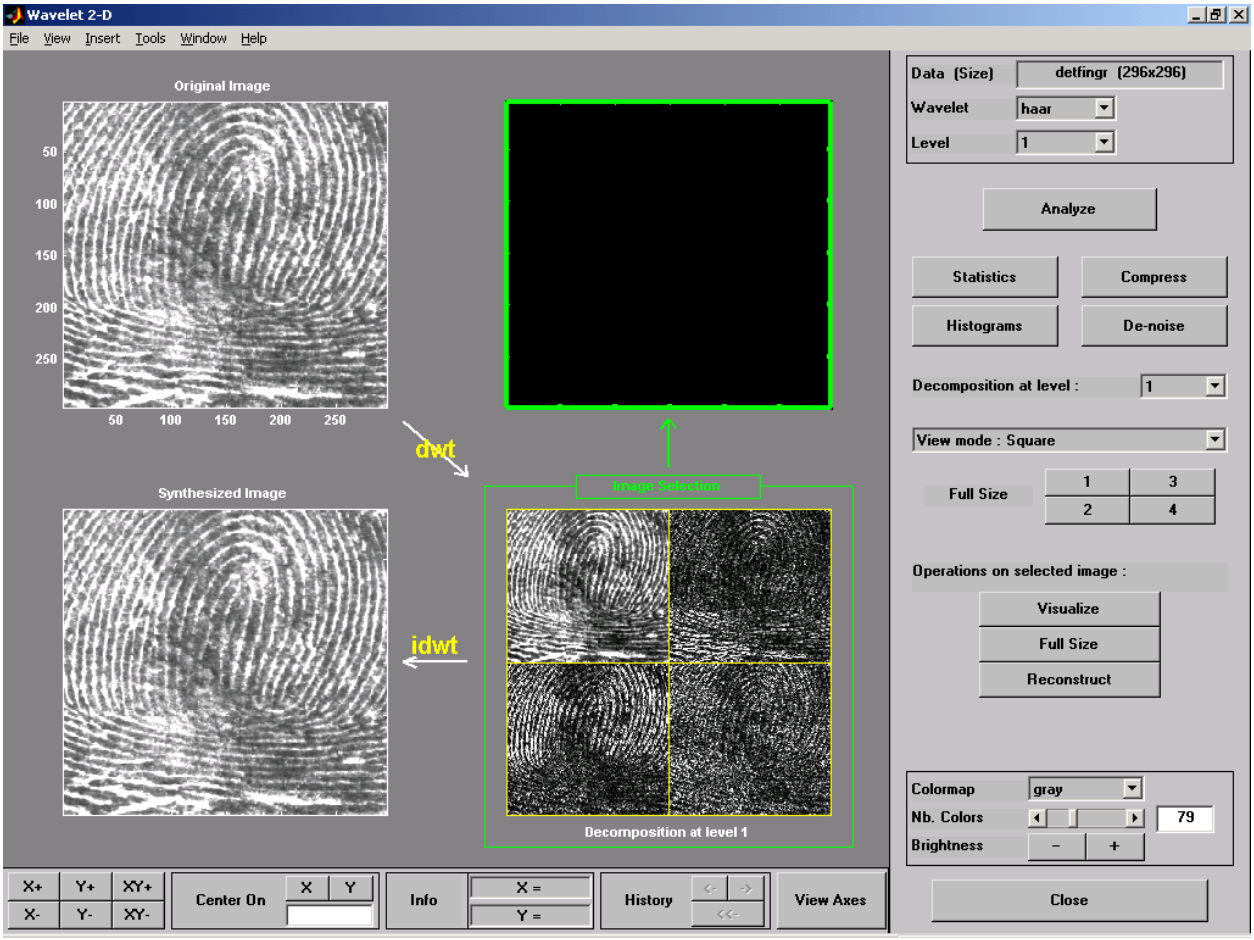
which is the same result when x = [1 1 1 1 1]. Hence the reconstruction result is correct for this input.

So how can we be assured of having the correct result when our input set contains an odd number of datapoints? The answer is to pad the odd set of data with one extra 0, flag this event, then strip out the extra zero when the reconstruction is done

```
dwtmode('sym')
x = [1 1 1 1 1 0 ];

[cA,cD] = dwt(x,'db2');

cA = [ 1.4142  1.4142 1.5436  0.3536 ]
cD = [-0.0000 -0.0000 0.4830 -0.6124 ]

cAu= dyadup(cA,0);
cDu= dyadup(cD,0);

xr = conv(cAu,phi/sqrt(2)) + conv(cDu,psi/sqrt(2));
xr = [0.6830  1.1830
      1.0000  1.0000 1.0000 1.0000 1.0000 0.0000
     -0.4330  0.2500]
```

Using the MATLAB Interface to do 2D Processing

The fingerprint data below are contained in a file called detfingr.mat. The file is located in the wavedemo subdirectory of the wavelet toolbox subdirectory. When you load the data, it first appears in the top left window. Select the Haar wavelet and a 1 level decomposition. Press analyze and the the following results:

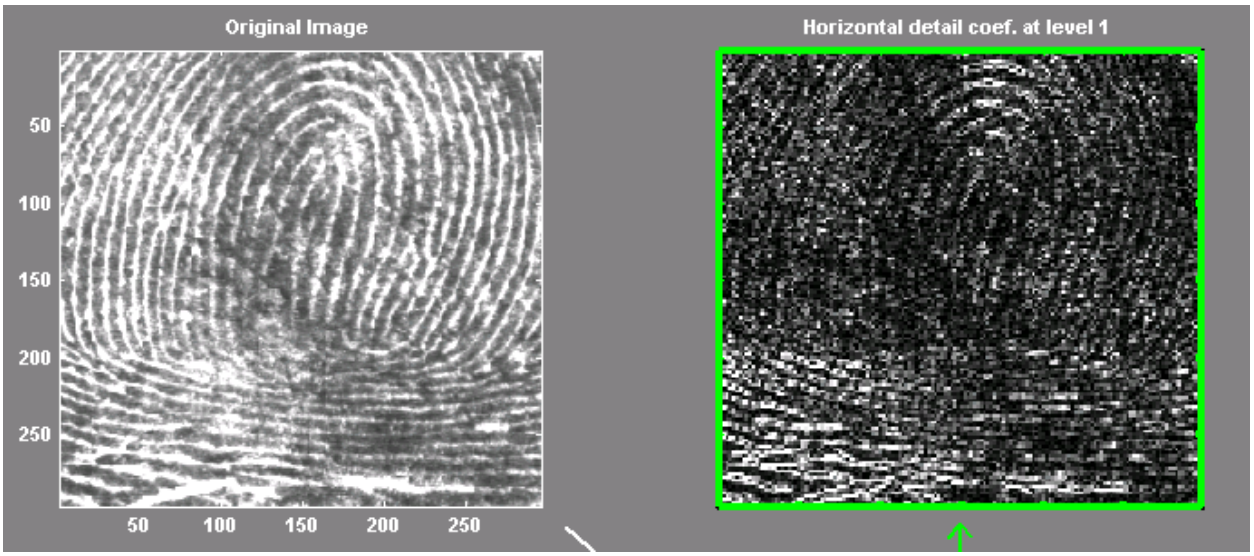


The wavemenu interface automatically reconstructs the image through the idwt2 function. From the command line, the analysis and synthesis procedure would be as follows:

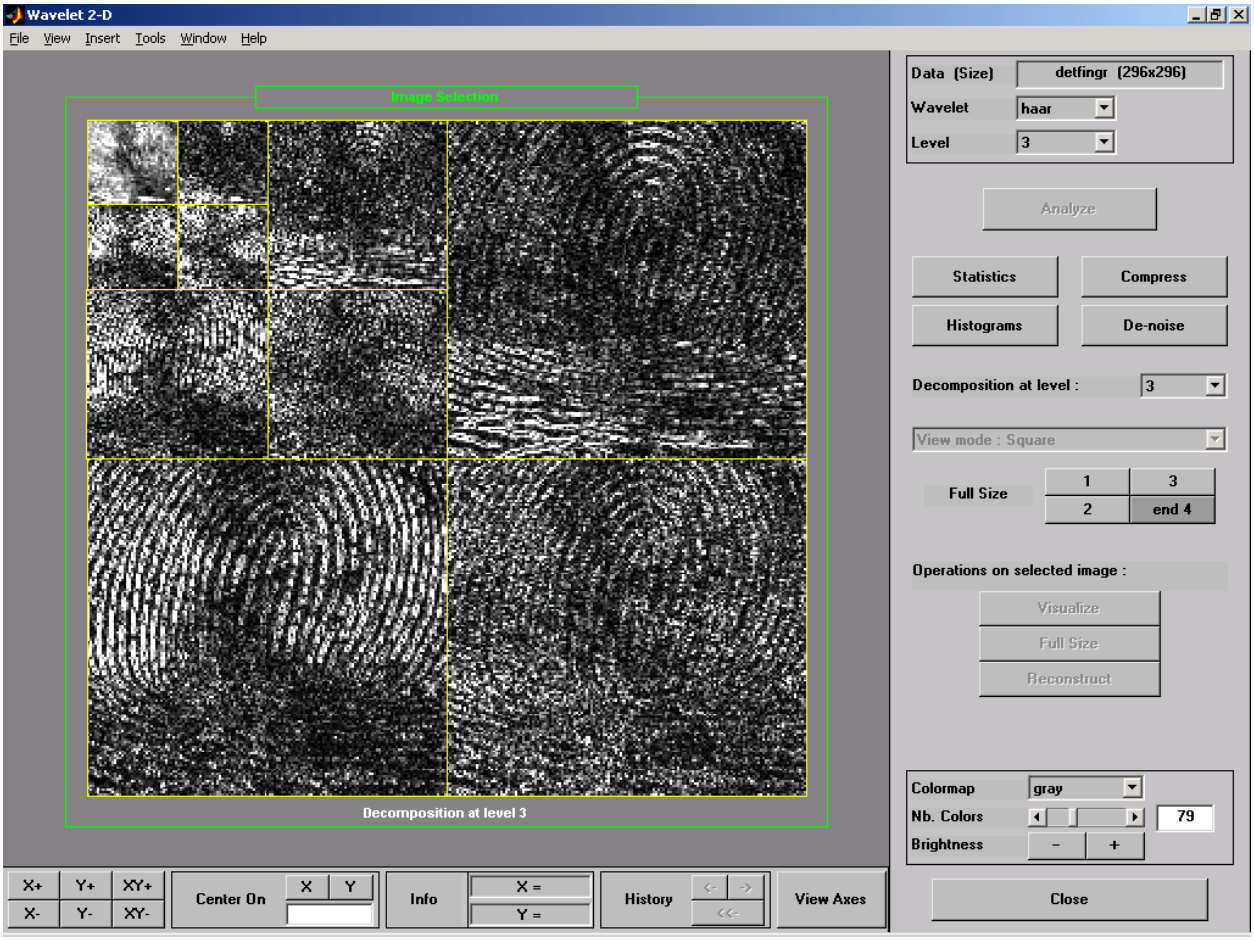
```
[cA,cH,cV,cD] = dwt2(x,'db1');
x_hat = idwt2(cA,cH,cV,cD,'db1');
```

The default color scheme is pink. I changed the above to gray and adjusted the brightness with the N.b. colors slider. This is a very useful feature for assessing the veracity of wavelet processing.

The bottom right axes shows the 1st level decomposition coefficients. Since these can appear as small images, there is a tool for viewing an individual set of coefficients. Click on the appropriate sub-image and then click on the visualize button. The following results when one compares the analysis and horizontal (cH) detail coefficients:

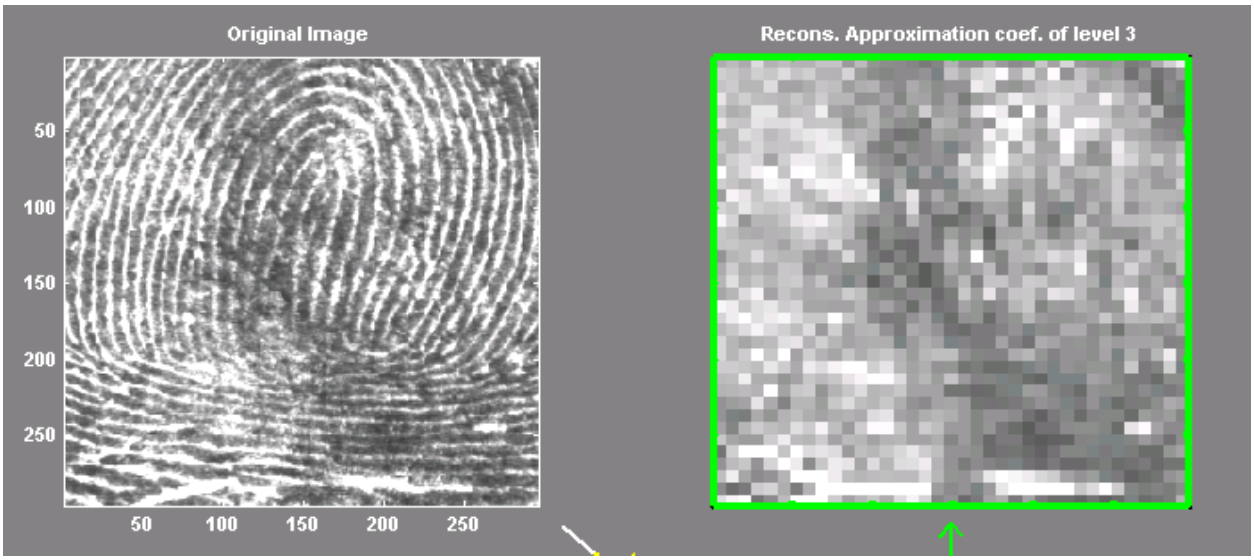


I checked this result by loading the data and doing the dwt2 by hand. Recall the horizontal detail coefficients are obtained by convolving the wavelet filter kernel vertically down the image (and then averaging this result with the scaling kernel). This will highlight differences of discontinuities in the vertical direction, but because of high degree of horizontal continuity in this image (and many others), the resulting image tends to have predominantly horizontal features. Keep this result in mind when doing these decompositions.



The above process can be repeated for higher level decompositions. Like the 1D transform, the 2D transform works solely on the original data set and subsequent analysis coefficients. The figure above shows a 3-level decomposition. You can also select the level of the decomposition that you want to look at. The above looks at all 3.

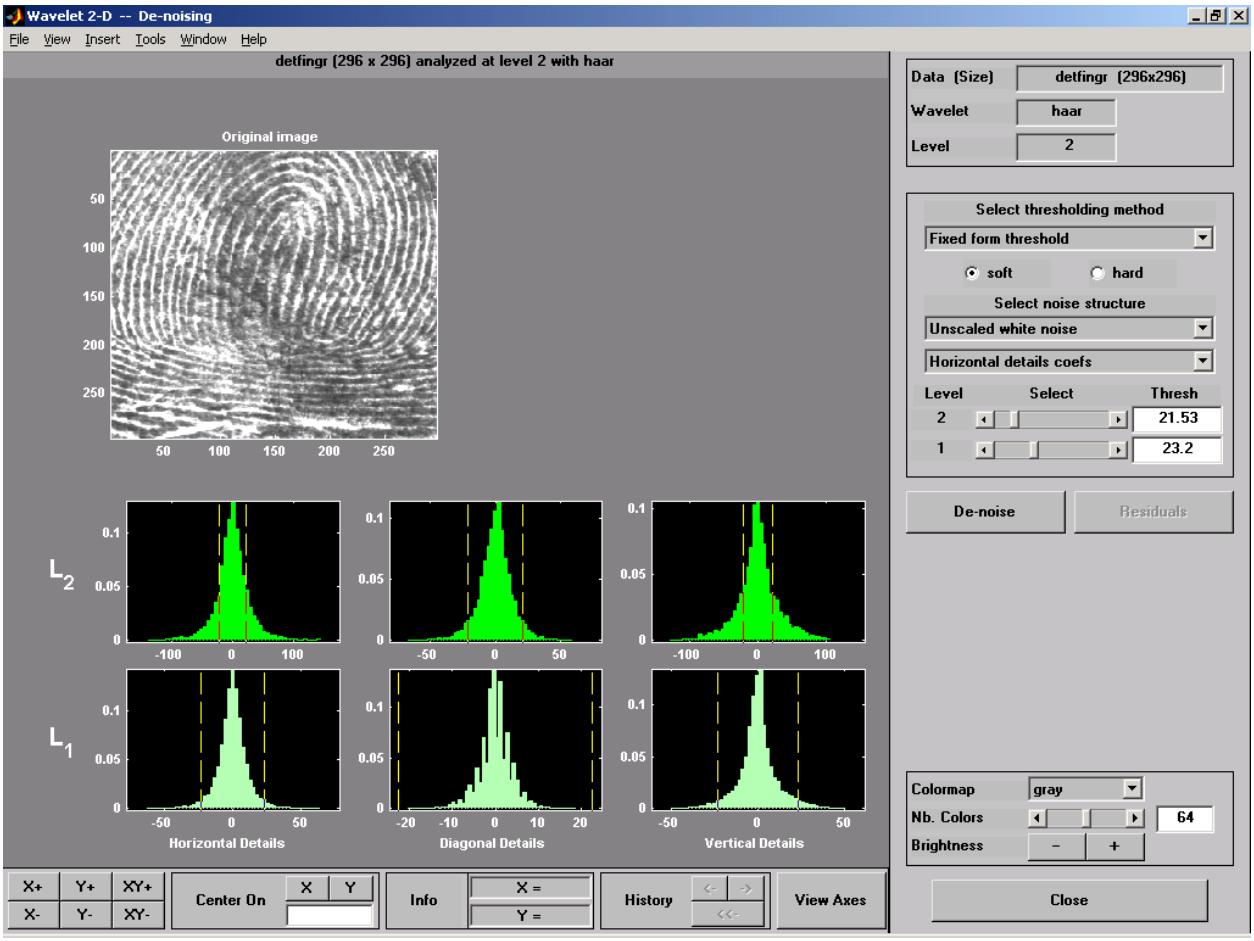
Another useful feature of the interface is that it allows you to reconstruct the image based on various levels of approximations. Of course, you can reconstruct on details, but this generally will not provide meaningful results. To reconstruct on the 3rd level approximations, click on the top left sub-image and press reconstruct. The following results:



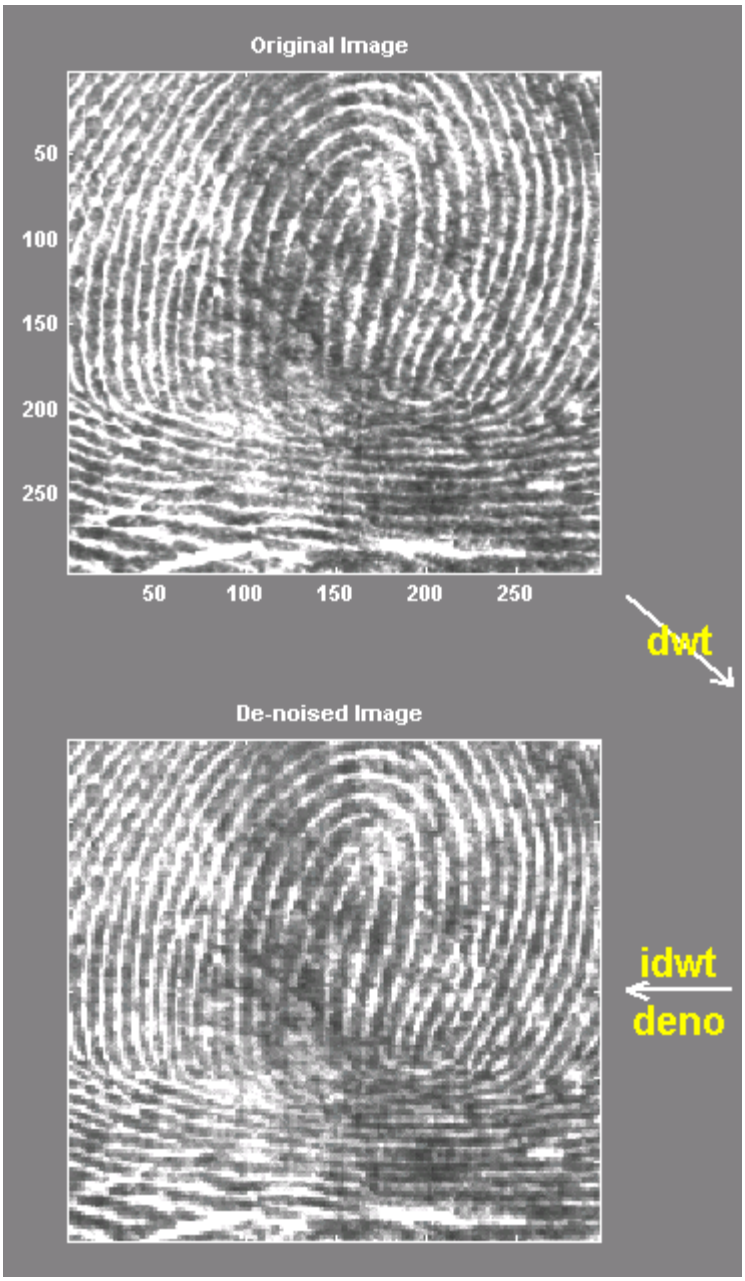
Clearly, a lot of information is lost in the detail coefficients, and without doing any processing on the coefficients, all we are doing here is viewing the 3rd-level approximations as up-sampled and reconstructed.

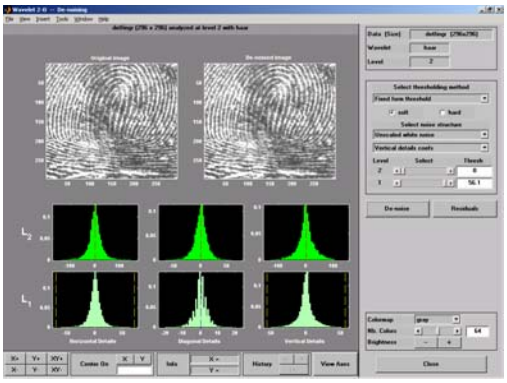
Let's now do a denoising of the image at level 2. In order to do this, you will need to re-analyze the image using a 2-level decomposition (since the prior 3rd level decomposition will carry over into the denoising subroutine). The following results:

Note that in denoising, the detail coefficients are thresholded, not the approximations. Histrograms of the various detail coefficients are shown. On the x-axis is the image intensity. Generally speaking, image data are positive integers, but detail coefficients will contain negative numbers, since the filter kernel contains negative coefficients. Adjust the sliders to the desired level and then select denoise. The signal can then be reconstructed – MATLAB does this automatically by doing the idwt2 on the denoised coefficients, but it keeps the original set of approximation and detail coefficients in the bottom right window.



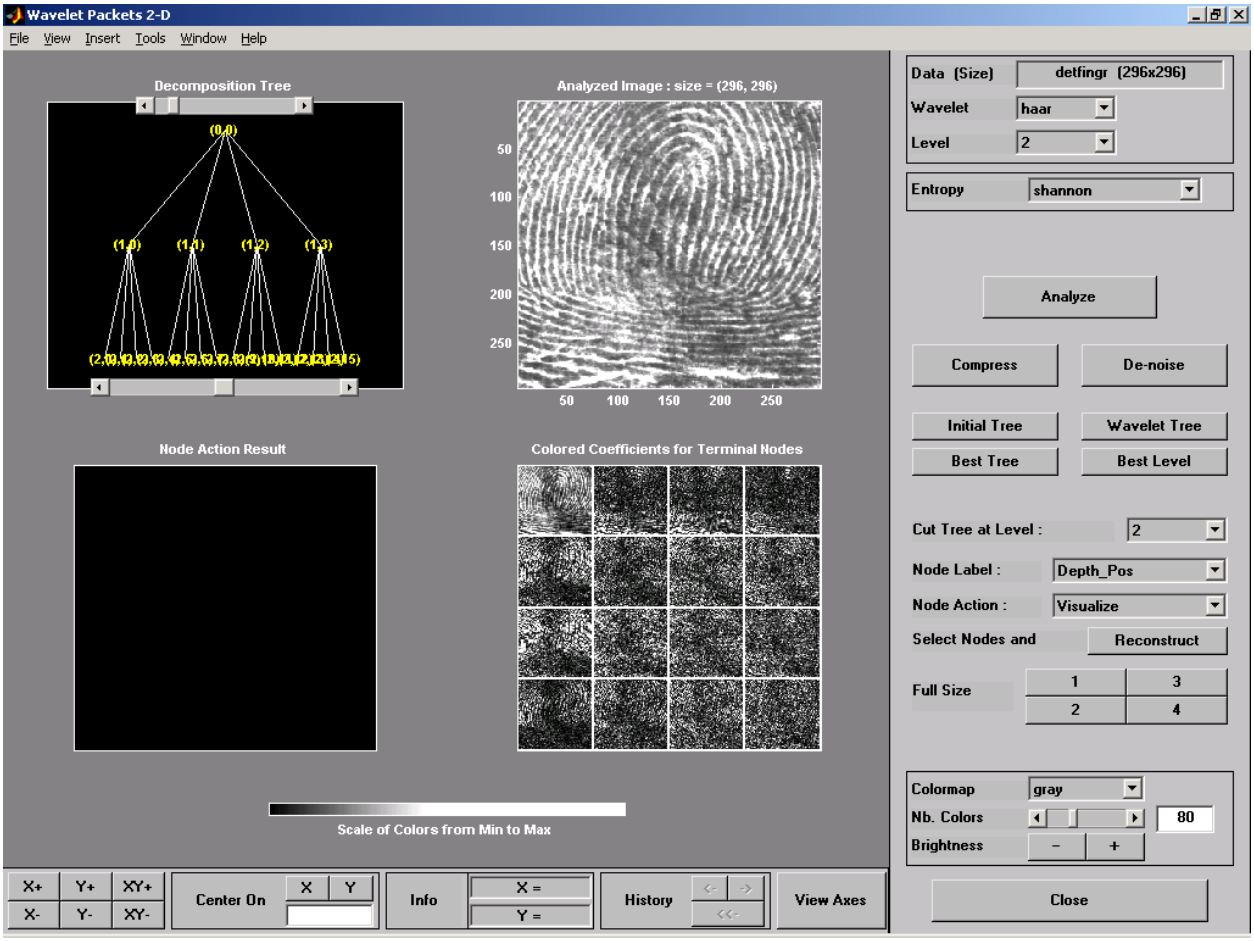
The figure below is denoise using only level 1 horizontal and vertical coefficients. Obviously there is little information in the level 2 coefficients and the level 1 diagonal coefficients.


Original Image
50
100
150
200
250
50
100
150
200
250
dwt
De-noised Image
idwt
deno

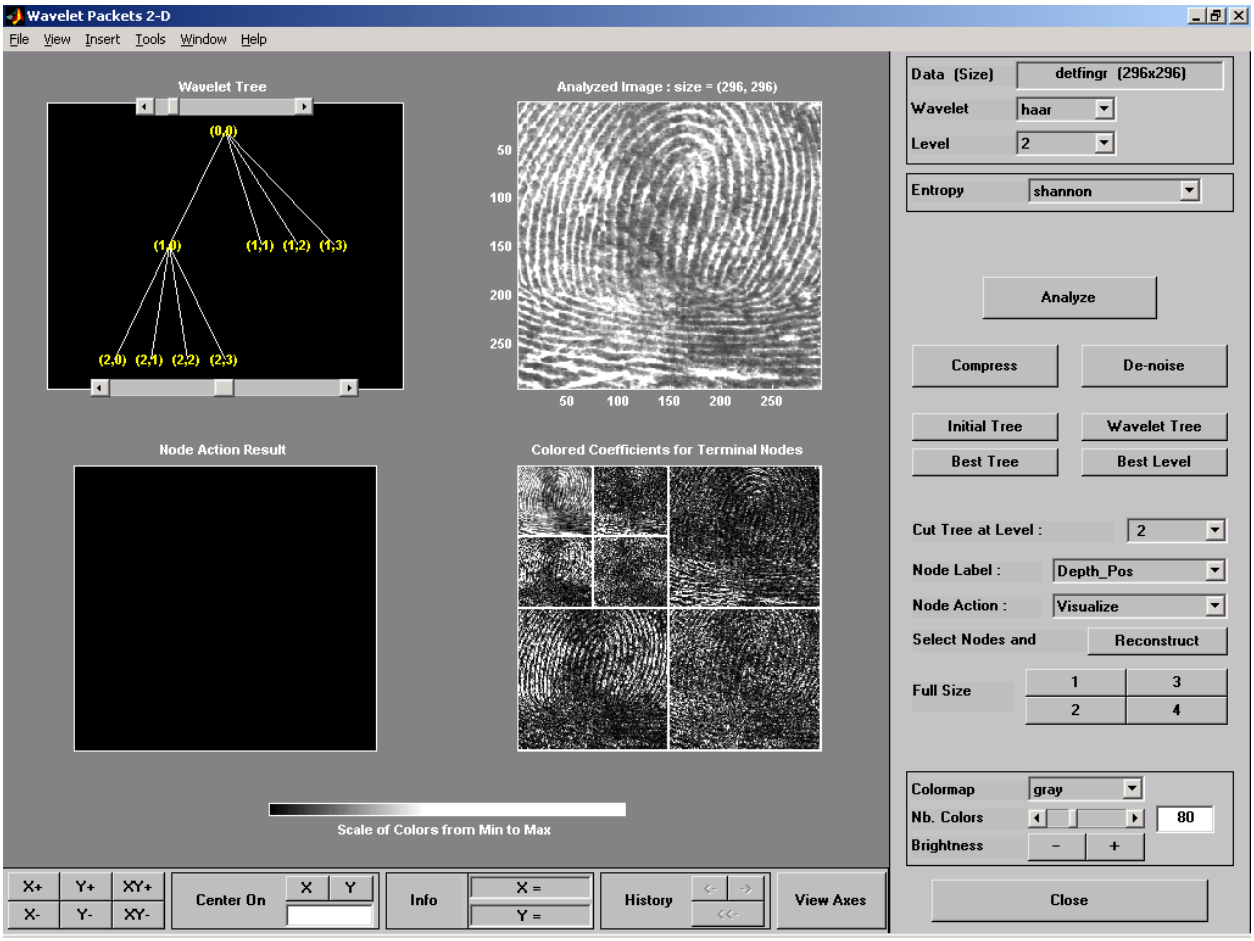


Wave Packet Analysis

Wavelet decomposition works by decomposing subsequent levels of approximations. Wave packet analysis works by decomposing subsequent levels of approximations and details. The tool can be selected from the wavemenu interface. It looks like this:



The previous analysis (based on wavelet decomposition can be obtained by selecting the wavelet tree button. The result looks like this:



Wavelet analysis, as it is called, is really a subset of packet analysis. You can select the nodes in the upper left figure window and the corresponding coefficients appear in the bottom-left window.

The packet analysis tool is good for doing image compression. Image compression is done by first finding an optimal set of decompositions based on image entropy, and then thresholding the resulting decompositions to retain a certain amount of the image energy.

Image Entropy:

Entropy in the classical sense is a measure of the amount of randomness in a set of probabilistic data. Some people associate randomness with information, but this is not necessarily the case. The Shannon Entropy of a random variable is given by

$$H=-\sum_{n=1}^{N} p_n \log_2 p_n \quad \text{(bits per symbol)}$$

where p_n are the discrete probabilities of the random variable. The probabilities p_n ust sum to 1.

However, we don't have to take the log in base 2 (which is what we typically do when analyzing digital communication systems. We can take the log in any base and the result is scaled by a constant, e.g.:

$$H=-\sum_{n=1}^{N} p_n \log_e p_n \quad \text{(nats per symbol)}$$

Suppose we treat an image as a random process, then we might view image intensity as a random variable, and compute a set of probabilities from its histogram. In this case N would be the N levels of quantization that the image intensity might take on. Another approach would be to compute the energy in each pixel and normalize it with respect to the entire picture energy. This would produce a somewhat different distribution, but with the same property that the sum of the elements is 1.

Let s_n be the pixel intensities. Typically these will be integers, but they can be positive or negative. Define e_n as the pixel energy: $e_n=s_n^2$. The according to the formulation for entropy, we would have

$$H_E=-\sum_{n=1}^{N} [e_n/E]\log_e[e_n/E]$$

where E is the total image entropy given by $E=\sum_{n=1}^{N} e_n$

This can be written as

$$H_E=-\left[\frac{1}{E}\sum_{n=1}^{N} e_n \log_e[e_n]\right]+\log_e E$$

If we set things up so that the energy image is unity (this is easily done by scaling each pixel by 1 over the square root of the total energy) then the relation becomes

$$H_E=-\sum_{n=1}^{N} e_n \log_e[e_n]$$

Now let's consider a simple 1-level decomposition of a small 4-by-4 image I=magic(4) using the Haar wavelet

```
I = [16     2     3    13
      5    11    10     8
      9     7     6    12
```

```
        4    14    15     1]
```

First we compute the energy of the image `EI = sum(I(:).^2)` = 1496 and normalize I with respect to the square root (38.6782);

```
In = [16     2     3    13
       5    11    10     8
       9     7     6    12
       4    14    15     1]/36.6782
```

To calculate the image entropy, you can do something like

```
H1 = -sum(In(:).^2.*log(In(:).^2))= 2.3720
```

Note the columnization of the image matrices – this is necessary because sum will produce a row vector when it operates on a matrix. Of course, MATLAB has a built in function that will do this, but you have to work with a properly normalized image

```
HI = wentropy(In,'shannon') = 2.3720
```

You will get a scaled and shifted version of this result if you don't have an image with unity energy. Next we do a 1st level decomposition on the normalized image

```
[cA,cH,cV,cD]=dwt2(In,'db1');

HA = wentropy(cA,'shannon') = 1.2705
HH = wentropy(cH,'shannon') = 0.0195
HV = wentropy(cV,'shannon') = 0.1941
HD = wentropy(cD,'shannon') = 0.5411

Σ  =                          2.0252
```

We have the result that the entropy (which is additive) has decreased from 2.3720 to 2.0252. IS this good or bad? How do we interpret the above result. To answer that question, lets take two images that are somewhat different in terms of their intensity distributions:

```
I1=ones(3,3)/3;
I2=zeros(3,3); I2(5)=1;

I1 = [ 1 1 1
       1 1 1
       1 1 1 ]/3

I2 = [ 0 0 0
       0 1 0
       0 0 0 ]

wentropy(I1,'shannon') = 2.1972
wentropy(I2,'shannon') = 0
```

The first result is `log(9)` – we see that a highly uniform image provides a high entropy, whereas a highly non-uniform image has low entropy. Random images have high entropy too: not as high as the case where the pixels are uniform, but asymptotically approaching that value when the number of pixels is large enough. We thus associate a reduction in entropy of the wavelet coefficients with an increase in

the meaningful structure of the coefficients, and therefore a desirable thing. If, on the other hand we find that a decomposition at a certain point results in more entropy, then that decomposition has less value.

Next we look at the issue of computing entropy based on unscaled images. This means that the image energy is not unity. As a general rule, energy is not conserved in the wavelet decomposition, and this has an important impact on the optimal decomposition.

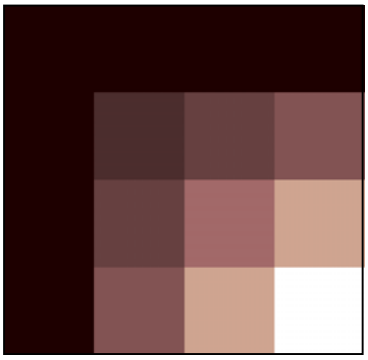
We illustrate with a rather striking example: a Pascal 4x4 matrix. This is generated using the command `pascal` is follows:

```
X = pascal(4) =   [ 1  1  1  1
                    1  2  3  4
                    1  3  6 10
                    1  4 10 20 ]
```

Next we normalized the image to have unity energy: `EX=sum(X(:).^2) = 697:`

```
Xn = X/sqrt(EX) = [0.0379 0.0379 0.0379 0.0379
                   0.0379 0.0758 0.1136 0.1515
                   0.0379 0.1136 0.2273 0.3788
                   0.0379 0.1515 0.3788 0.7576]
```

The image looks something like this:



We do a 2-level packet decompsoition on the image, starting with the first level

```
[cA,cH,cV,cD]=dwt2(Xn,'db2')

cA = [ 0.0805  0.0878  0.1278
       0.0878  0.1079  0.1807
       0.1278  0.1807  1.1647 ]
cH = [-0.0082 -0.0208 -0.0902
      -0.0000  0.0062 -0.1014
       0.0082  0.0102  0.2788 ]
cV = [-0.0082 -0.0000  0.0082
      -0.0208  0.0062  0.0102
      -0.0902 -0.1014  0.2788 ]
cD = [ 0.0142  0.0000 -0.0142
       0.0000  0.0230 -0.0396
      -0.0142 -0.0396  0.0852
```

and then compute the next level

```
[cAA,cAH,cAV,CAD]=dwt2(cA,'db2')
cAA = [ 0.1699    0.1849    0.2775
        0.1849    0.2159    0.4439
        0.2775    0.4439    2.0927]
cAH = [-0.0091   -0.0155   -0.0449
        0.0260    0.0646    0.4725
       -0.0170   -0.0491   -0.4276]
cAV = [-0.0091    0.0260   -0.0170
       -0.0155    0.0646   -0.0491
       -0.0449    0.4725   -0.4276]
cAD = [ 0.0048   -0.0081    0.0033
       -0.0081    0.1171   -0.1090
        0.0033   -0.1090    0.1057]
```

```
[cHA,cHH,cHV,cHD]=dwt2(cH,'db2')
cHA = [-0.0163   -0.0333   -0.1769
       -0.0048   -0.0145   -0.1350
        0.0188    0.0542    0.4732]
cHH = [-0.0112   -0.0156    0.0060
        0.0059    0.0186    0.1761
        0.0052   -0.0030   -0.1821]
cHV = [ 0.0068   -0.0409    0.0341
        0.0004   -0.0348    0.0343
       -0.0034    0.1207   -0.1173]
cHD = [ 0.0070    0.0068   -0.0138
       -0.0006    0.0454   -0.0448
       -0.0065   -0.0522    0.0586]
```

```
[cVA,cVH,cVV,cVD]=dwt2(cV,'db2')
cVA = [-0.0163   -0.0048    0.0188
       -0.0333   -0.0145    0.0542
       -0.1769   -0.1350    0.4732]
cVH = [ 0.0068    0.0004   -0.0034
       -0.0409   -0.0348    0.1207
        0.0341    0.0343   -0.1173]
cVV = [-0.0112    0.0059    0.0052
       -0.0156    0.0186   -0.0030
        0.0060    0.1761   -0.1821]
cVD = [ 0.0070   -0.0006   -0.0065
        0.0068    0.0454   -0.0522
       -0.0138   -0.0448    0.0586]
```

```
[cDA,cDH,cDV,cDD]=dwt2(cD,'db2')
cDA = [ 0.0189    0.0113   -0.0385
        0.0113    0.0126   -0.0394
       -0.0385   -0.0394    0.1312]
cDH = [ 0.0042   -0.0061    0.0162
       -0.0141   -0.0155    0.0505
        0.0098    0.0216   -0.0667]
cDV = [ 0.0042   -0.0141    0.0098
       -0.0061   -0.0155    0.0216
        0.0162    0.0505   -0.0667]
cDD = [ 0.0140    0.0075   -0.0215
        0.0075    0.0196   -0.0272
       -0.0215   -0.0272    0.0487]
```

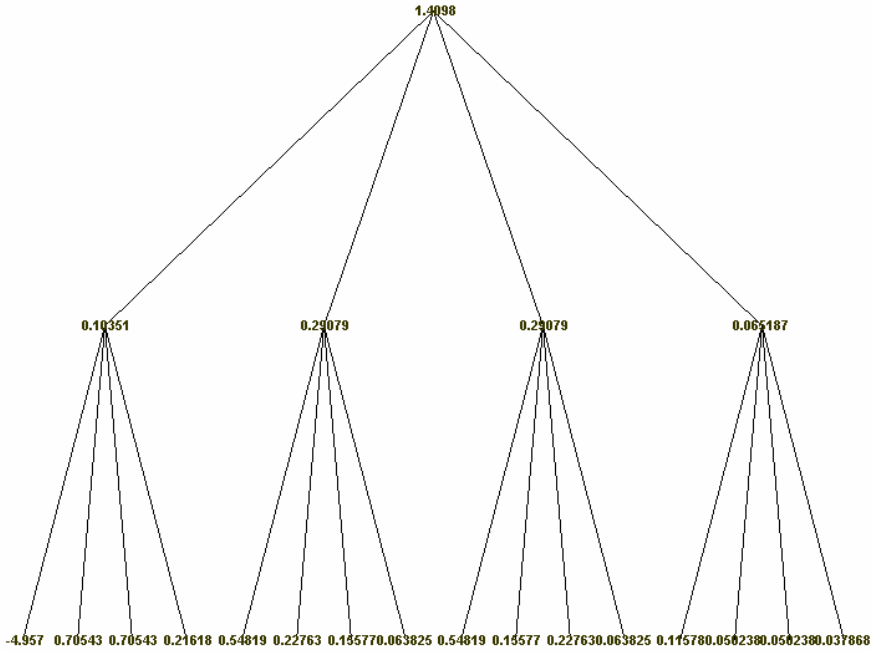
Next we compute the entropies of the various coefficients, staring with the image X:

```
-sum(log(Xn(:).^2).*Xn(:).^2) = 1.4098

-sum(log(cA(:).^2).*cA(:).^2) = 0.1035
-sum(log(cH(:).^2).*cH(:).^2) = 0.2908
-sum(log(cV(:).^2).*cV(:).^2) = 0.2908
-sum(log(cD(:).^2).*cD(:).^2) = 0.0652

                            Σ = 0.7503
```

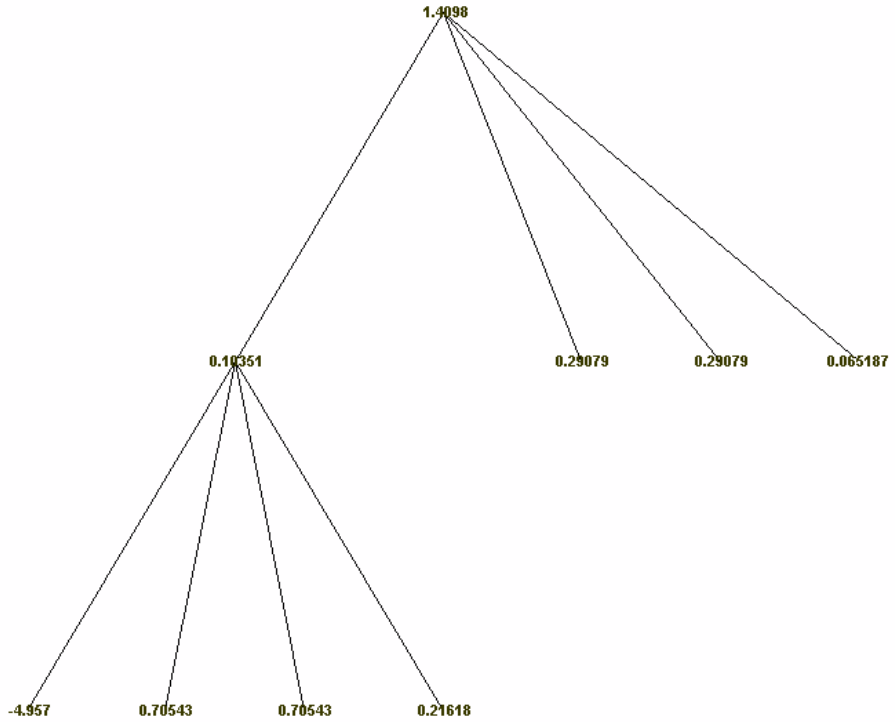
The entropy has reduced from the image to the first level, therefore we keep the entire first level of decompositions. Now we repeat the process on the first level decompositions:

```
[cA,cH,cV,cD]     = dwt2(Xn,'db2');
[cAA,cAH,cAV,cAD]= dwt2(cA,'db2');
[cHA,cHH,cHV,cHD]= dwt2(cH,'db2');
[cVA,cVH,cVV,cVD]= dwt2(cV,'db2');
[cDA,cDH,cDV,cDD]= dwt2(cD,'db2');

h1 = wentropy(cAA,'shannon')    -4.9570
h2 = wentropy(cAH,'shannon')     0.7054
h3 = wentropy(cAV,'shannon')     0.7054
h4 = wentropy(cAD,'shannon')     0.2162
                              Σ -3.3333  <
Entropy at parent node                    0.1035
h5 = wentropy(cHA,'shannon')     0.5482
h6 = wentropy(cHH,'shannon')     0.2276
h7 = wentropy(cHV,'shannon')     0.1558
h8 = wentropy(cHD,'shannon')     0.0638
                              Σ  0.9954  >
Entropy at parent node                    0.2908
h9 = wentropy(cVA,'shannon')     0.5482
h10= wentropy(cVH,'shannon')     0.1558
h11= wentropy(cVV,'shannon')     0.2276
h12= wentropy(cVD,'shannon')     0.0638
                              Σ  0.9954  >
Entropy at parent node                    0.2908
h13= wentropy(cDA,'shannon')     0.1158
h14= wentropy(cDH,'shannon')     0.0502
h15= wentropy(cDV,'shannon')     0.0502
h16= wentropy(cDD,'shannon')     0.0379
                              Σ  0.2451  >
Entropy at parent node                    0.0652
```

The process is nicely summarized in the MATLAB wavemenu interface: below we see the initial tree with node entropies.



Only the cA coefficients have 2nd level entropy that is lower than the node entropy at the previous level, so only these coefficients are expanded in the next level. The best tree is the tree that has least entropy. This is shown as follows:



However, we notice that the entropy of the cAA coefficient is negative! This is because the dB2 wavelet transform does not (generally) conserve energy. This is easy to show by example.

As such, we should first normalize the energies of each group of coefficients before contemplating further decomposition. At the first level decomposition, we start with an energy of 1 in the original image, and end up with an energy of

```
sum(cA(:).^2)     1.4881
sum(cH(:).^2)     0.0969
sum(cV(:).^2)     0.0969
sum(cD(:).^2)     0.0115
            Σ     1.6934
```

Since we want the transform to conserve energy, we divide each coefficient by 1.3013 so the total energy in the transform is now 1.

```
cAn=cA/1.3013;
cHn=cH/1.3013;
cVn=cV/1.3013;
cDn=cD/1.3013;
```

and repeat the computation of entropy at the first layer

```
wentropy(cAn,'shannon')    0.5240
wentropy(cHn,'shannon')    0.2019
wentropy(cVn,'shannon')    0.2019
wentropy(cDn,'shannon')    0.0421
                       Σ   0.9698
```

We note that the net entropy decreases (from 1.4098 to 0.9698) therefore the first level decomposition is worth doing. Therefore we consider second level decompositions. However, the first level coefficients do not individually have normalized energy (they collectively have normalized energy), so we must do a second normalizing on each set of coefficients as follows:

```
cAnn=cAn/sqrt(sum(cAn(:).^2));
cHnn=cHn/sqrt(sum(cHn(:).^2));
cVnn=cVn/sqrt(sum(cVn(:).^2));
cDnn=cDn/sqrt(sum(cDn(:).^2));
```

The compute the next layer of coefficients

```
[cAA,cAH,cAV,cAD]= dwt2(cAnn,'db2');
[cHA,cHH,cHV,cHD]= dwt2(cHnn,'db2');
[cVA,cVH,cVV,cVD]= dwt2(cVnn,'db2');
[cDA,cDH,cDV,cDD]= dwt2(cDnn,'db2');
```

The scale each of these so they have unity group energy

```
cAAn=cAA/sqrt(sum(cAA(:).^2+cAH(:).^2+cAV(:).^2+cAD(:).^2));
cAHn=cAH/sqrt(sum(cAA(:).^2+cAH(:).^2+cAV(:).^2+cAD(:).^2));
cAVn=cAV/sqrt(sum(cAA(:).^2+cAH(:).^2+cAV(:).^2+cAD(:).^2));
cADn=cAD/sqrt(sum(cAA(:).^2+cAH(:).^2+cAV(:).^2+cAD(:).^2));

cHAn=cHA/sqrt(sum(cHA(:).^2+cHH(:).^2+cHV(:).^2+cHD(:).^2));
cHHn=cHH/sqrt(sum(cHA(:).^2+cHH(:).^2+cHV(:).^2+cHD(:).^2));
cHVn=cHV/sqrt(sum(cHA(:).^2+cHH(:).^2+cHV(:).^2+cHD(:).^2));
cHDn=cHD/sqrt(sum(cHA(:).^2+cHH(:).^2+cHV(:).^2+cHD(:).^2));

cVAn=cVA/sqrt(sum(cVA(:).^2+cVH(:).^2+cVV(:).^2+cVD(:).^2));
cVHn=cVH/sqrt(sum(cVA(:).^2+cVH(:).^2+cVV(:).^2+cVD(:).^2));
cVVn=cVV/sqrt(sum(cVA(:).^2+cVH(:).^2+cVV(:).^2+cVD(:).^2));
cVDn=cVD/sqrt(sum(cVA(:).^2+cVH(:).^2+cVV(:).^2+cVD(:).^2));

cDAn=cDA/sqrt(sum(cDA(:).^2+cDH(:).^2+cDV(:).^2+cDD(:).^2));
cDHn=cDH/sqrt(sum(cDA(:).^2+cDH(:).^2+cDV(:).^2+cDD(:).^2));
cDVn=cDV/sqrt(sum(cDA(:).^2+cDH(:).^2+cDV(:).^2+cDD(:).^2));
cDDn=cDD/sqrt(sum(cDA(:).^2+cDH(:).^2+cDV(:).^2+cDD(:).^2));
```

Then compute the entropy

```
h1 = wentropy(cAAn,'shannon')       0.6870
h2 = wentropy(cAHn,'shannon')       0.2432
h3 = wentropy(cAVn,'shannon')       0.2432
h4 = wentropy(cADn,'shannon')       0.0510
                            Σ       1.2243

h5 = wentropy(cHAn,'shannon')       0.7336
h6 = wentropy(cHHn,'shannon')       0.4285
h7 = wentropy(cHVn,'shannon')       0.3198
h8 = wentropy(cHDn,'shannon')       0.1389
                            Σ       1.6208

h9 = wentropy(cVAn,'shannon')       0.7336
h10= wentropy(cVHn,'shannon')       0.4285
h11= wentropy(cVVn,'shannon')       0.3198
h12= wentropy(cVDn,'shannon')       0.1389
                            Σ       1.6208

h13= wentropy(cDAn,'shannon')       0.9062
h14= wentropy(cDHn,'shannon')       0.5340
h15= wentropy(cDVn,'shannon')       0.5340
h16= wentropy(cDDn,'shannon')       0.4565
                            Σ       2.4306
```

We compare the above sums to the normalized coefficients cAnn, cHnn, cVnn, cDnn

```
wentropy(cAnn,'shannon')      0.4671
wentropy(cHnn,'shannon')      0.6674
wentropy(cVnn,'shannon')      0.6674
wentropy(cDnn,'shannon')      1.1879
```

and see that in each case the second level entropy has increased.

In summary, it is apparent that the Shannon Entropy, as formulated in terms of energy, is particularly sensitive to scaling. The approach we have taken here is to start with an image of unity energy, do the first level decomposition on the image to obtain the first level coefficients. Normalize this group of coefficients so that they have unity energy (as a group), then compute the entropy of each set of coefficients, sum, and compare to the entropy of the image. If the sum entropy of the first level is smaller, then keep the first level coefficients and consider do a second level. Before computing the second level coefficients, each set of first level coefficients is renormalized to have unity energy, and the process is repeated. However, in subsequent iterations, we must compare the sum energy of the next level of coefficients with the renormalized entropy of the parent node. This is not the procedure that is followed in MATLAB.

A Further Note on the Use of Shannon Entropy

Shannon entropy, as previously mentioned, is highly sensitive to scaling. This example shows how Let $x(n)$ be a series of image data with energy and entropy given by

$$E(x) = \sum_{n=1}^{N} x^2(n)$$

$$H_E(x) = -\sum_{n=1}^{N} x^2(n)\log(x^2(n))$$

Now define a scaled version of the image $y(n) = \lambda x(n)$. Then

$$E(y) = \sum_{n=1}^{N} y^2(n) = \sum_{n=1}^{N} \lambda^2 x^2(n) = \lambda^2 E(x)$$

$$H_E(y) = -\sum_{n=1}^{N} y^2(n)\log(y^2(n)) = -\sum_{n=1}^{N} \lambda^2 x^2(n)\log(\lambda^2 x^2(n))$$

$$H_E(y) = -\sum_{n=1}^{N} \lambda^2 x^2(n)[\log(\lambda^2) + \log(x^2(n))] = -\lambda^2 \log(\lambda^2) \sum_{n=1}^{N} x^2(n) - \lambda^2 \sum_{n=1}^{N} x^2(n)\log(x^2(n))]$$

$$H_E(y) = -\lambda^2 \log(\lambda^2) E(x) + \lambda^2 H_E(x)$$

$$H_E(y) = \lambda^2 H_E(x) - \log(\lambda^2) E(y)$$

$$\frac{H_E(y)}{E(y)} = \frac{H_E(x)}{E(x)} - \log\left(\frac{E(y)}{E(x)}\right) \quad \text{or}$$

$$H_E(y) = \frac{E(y) H_E(x)}{E(x)} - E(y)\log\left(\frac{E(y)}{E(x)}\right)$$

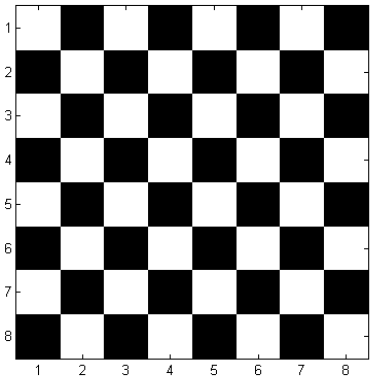
The transformation, although appearing to be linear (ax+b) is not really linear, it is affine. We can end up with negative entropies when $E(y) > E(x)$. It is straightforward to compute a condition that prevents negative entropies, but it is not clear if this would be useful or not.

Compression of Regularized Images

To start this study in compression, we consider regularized images, which (my definition) are images with a high degree of symmetry or periodicity. The first of such is an 8-by-8 checkerboard. The following section of code generates such a matrix:

```
N=8;
K=((N+1)^2+1)/2;
X=[ones(1,K); zeros(1,K)];
X=X(:);
X=X(1:end-1);
X=reshape(X,N+1,N+1);
X=X(1:N,1:N);
```

The image can be viewed as follows:




```
imagesc(X)
axis('image')
colormap('bone')
set(gcf,'color', [1 1 1])
```

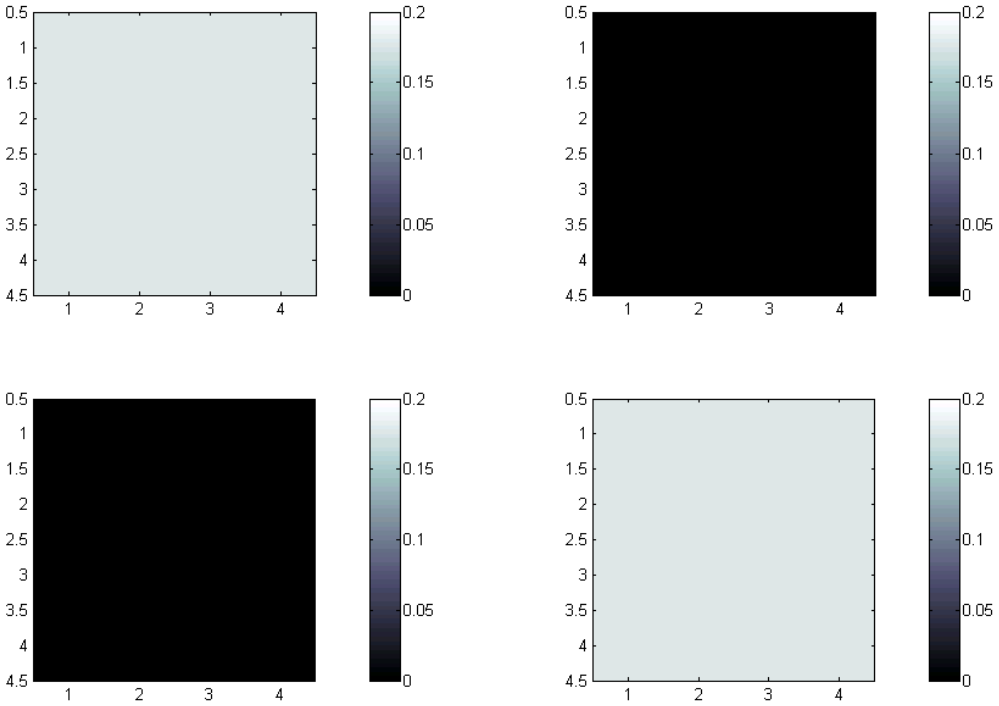
Note, it is important to use the function imagesc rather than image, because image is set up to display indexed images. You can use image(X*256) or change the colormap and color limits to use image directly. For this reason, the images displayed in the wavelet GUI are not visitible, but the mathematical operations are correct.

We perform a 2D dwt on a normalized image:

```
[cA,cH,cV,cD]=dwt2(X/sqrt(sum(X(:).^2)),'db1');
```

and plot

```
subplot(2,2,1), imagesc(cA), axis('image'), caxis([0 0.2]), colorbar
subplot(2,2,2), imagesc(cH), axis('image'), caxis([0 0.2]), colorbar
subplot(2,2,3), imagesc(cV), axis('image'), caxis([0 0.2]), colorbar
subplot(2,2,4), imagesc(cD), axis('image'), caxis([0 0.2]), colorbar
```



Both cA and cD are uniform with values equal to the original non-zero (normalized) image intensities 0.1768. The cH and cV coefficients are zero. This is a little counter intuitive, but adjacent columns on the vertical convolution are negative, so they average to zero. We automatically have compression with any further work, since half of the coefficients are zero. Can we do any better?

If we do an entropy calculation, we find the entropy of the image to be 3.47 (using the Shannon metric). Because we have used the Haar wavelet, we have conserved energy, and it turns out that entropy in the cA and cD coefficients are equal and equals the original entropy. This would suggest that the original decomposition is of minimal value, but as we have seen, we already have a compression of 50%. Let's go ahead and do a second level decomposition on cA and cD:

```
[cAA,cAH,cAV,cAD]=dwt2(cA,'db1');
[cDA,cDH,cDV,cDD]=dwt2(cD,'db1');
```

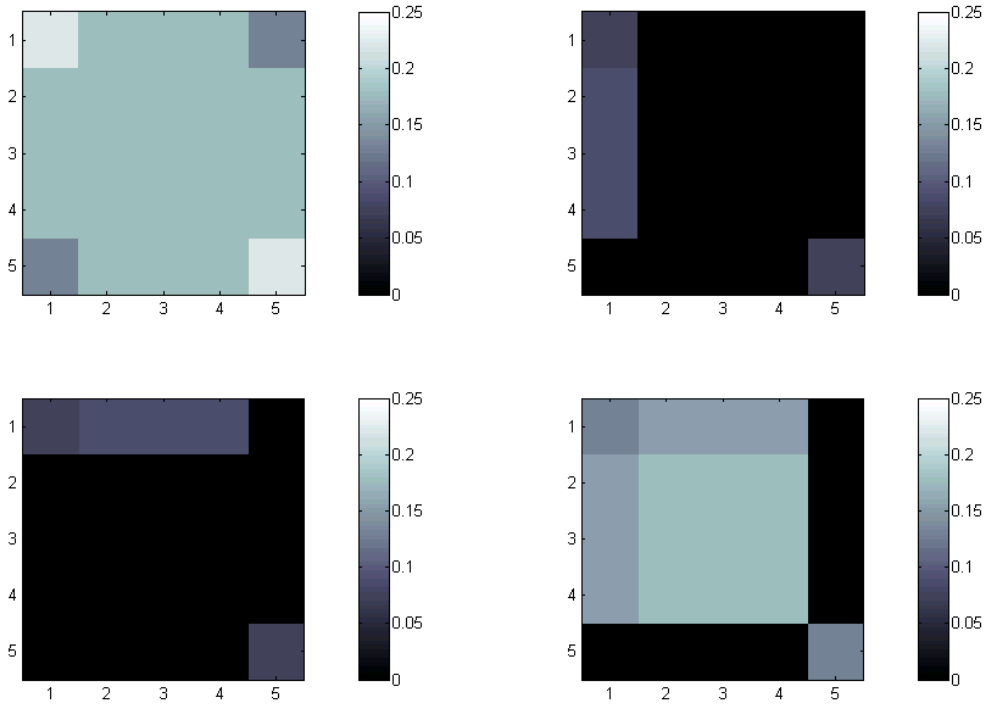
In this case, the only non-zero coefficients are cAA and cDA = [0.3536 0.3536; 0.3536 0.3536]. So we have gone from 64 image pixels to 2*4=8 coefficients. Can we do any better? Obviously yes, we do a third level decomposition on cAA and cDA, and end up with only 1 non-zero coefficient each (both with value 0.7071. So a wavelet packet decomposition on the checkerboard and an appropriate thresholding results in a compression down to two non-zero coefficients: cAAA and cDAA. We have gone from 64 pixels to 2, and this result will always result in two pixels regardless of the original size of the image. If we normalize the pixels to unity, then we can think of the two pixels as being encoded with 2 bits of information, and our original image started with 1 bit of information from a Shannon probabilitic point of view. This is an interesting result. Is the compression obtained above the maximum possible compression. I believe it is, because the 1 bit of Shannon information gives us no information about the horizontal and vertical true Shannon entropies. In other words, if we concatenate all of the pixels end-to-end and calculate their Shannon entropy based on their historgram, we find 50% white and 50% black

pixels, making for an entropy of 1 bit. If we calculate the entropy down the columns, and average, we find 1 bit, and across the rows, we find 1 bit. I claim that these results are orthogonal. For example, the following image

```
1     0     1     0
1     0     1     0
1     0     1     0
1     0     1     0
```

Has 1 bit of true Shannon entropy per row (on average) and zero bits of Shannon entropy per column (on average), but as a whole it has 1 bit per pixel. Thus, I conjecture that the Wavelet packet decomposition of the checkerboard using the Haar wavelet is the best possible, in the sense that it obtains the 2D true Shannon entropy. Do we get the same result for the db2 wavelet?

```
[cA,cH,cV,cD]=dwt2(X,'db2');

subplot(2,2,1), imagesc(cA), axis('image'), caxis([0 0.25]), colorbar
subplot(2,2,2), imagesc(cH), axis('image'), caxis([0 0.25]), colorbar
subplot(2,2,3), imagesc(cV), axis('image'), caxis([0 0.25]), colorbar
subplot(2,2,4), imagesc(cD), axis('image'), caxis([0 0.25]), colorbar
```



We se a result that is similar: we see that cA and cD are predominantly uniform, and cH and cV are predominantly zero, but not quite. Lets calculate Shannon entropy. First we note that db2 wavelet does not conserve energy – the energy in the first level coefficients is 1.5625, so we normalize the 1st level coefficients by the square root and calculate the entropy:

```
wentropy(cA/sqrt(e1),'shannon')     1.9606
wentropy(cH/sqrt(e1),'shannon')     0.2427
wentropy(cV/sqrt(e1),'shannon')     0.2427
wentropy(cD/sqrt(e1),'shannon')     1.6620
                                    4.1081
```

Shannon entropy has increased over 3.47, but this is really a function of the Shannon metric. Both cH and cD have low energy and low entropy:

```
sum(cA(:).^2)     0.7891
sum(cH(:).^2)     0.0703
sum(cV(:).^2)     0.0703
sum(cD(:).^2)     0.6328
                  1.5625
```

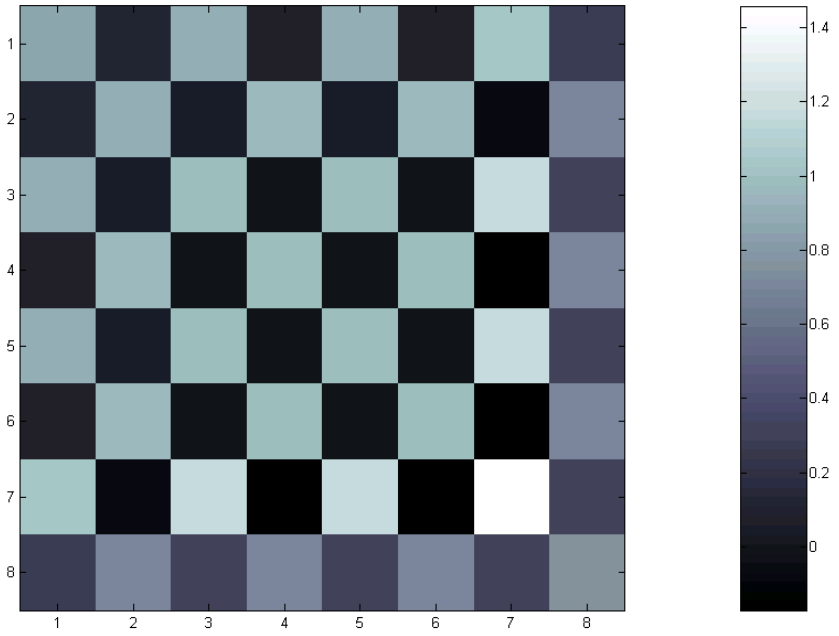
so we might threshold a lot of these values down to zero

```
cH =
    0.0765   -0.0000   -0.0000   -0.0000   -0.0765
    0.0884   -0.0000   -0.0000   -0.0000   -0.0884
    0.0884   -0.0000   -0.0000   -0.0000   -0.0884
    0.0884   -0.0000   -0.0000   -0.0000   -0.0884
   -0.0765   -0.0000   -0.0000   -0.0000    0.0765

cV =
    0.0765    0.0884    0.0884    0.0884   -0.0765
    0.0000    0.0000    0.0000    0.0000   -0.0000
    0.0000    0.0000    0.0000    0.0000   -0.0000
    0.0000    0.0000    0.0000    0.0000   -0.0000
   -0.0765   -0.0884   -0.0884   -0.0884    0.0765
```

We could easily remove the 'zero' terms because they are on the order of e-18. Or we could remove everything, because the largest terms in cH and cV are about 10 times smaller than the smallest terms in cA and cD. Let's look at the reconstruction if we assume cH and cV are all set to zero:

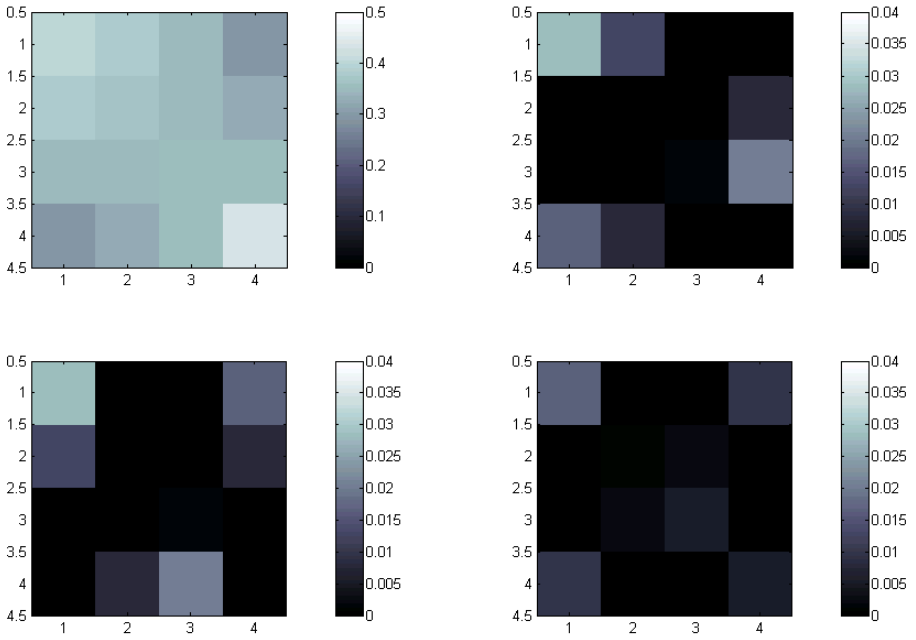
```
Xr=idwt2(cA, zeros(5,5), zeros(5,5), cD,'db2')
imagesc(Xr*5.65685)
colormap('bone')
axis('image')
colorbar
```

We get something approximating the original image, but clearly there are some errors. The small values removed in cH and cV are important, and the thresholding was too severe.

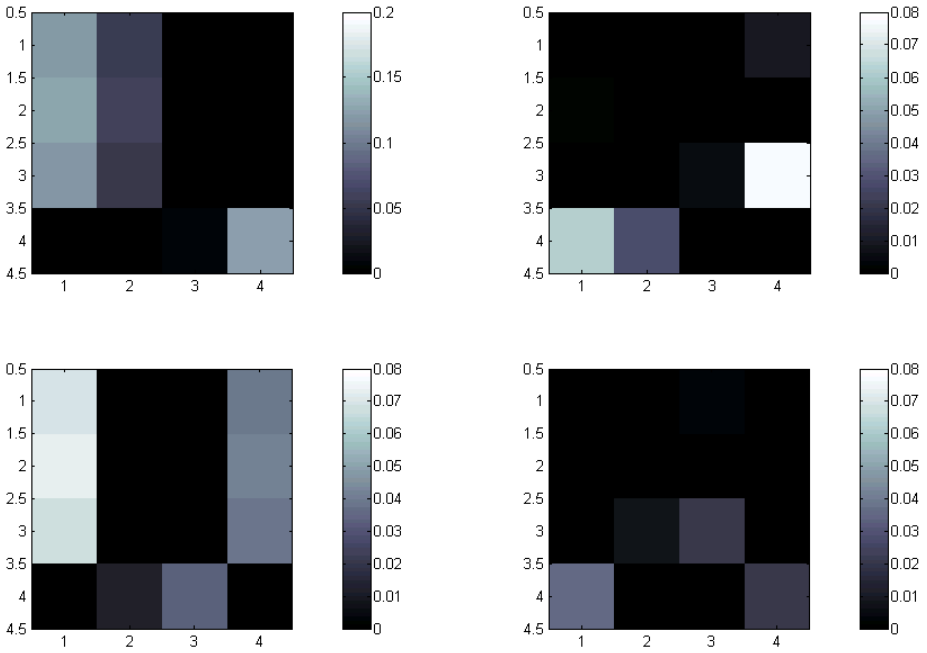


Subsequent levels of decomposition may provide further compression:

```
[cAA,cAH,cAV,cAD]=dwt2(cA,'db2');

subplot(2,2,1), imagesc(cAA), axis('image'), caxis([0 0.5]),  colorbar
subplot(2,2,2), imagesc(cAH), axis('image'), caxis([0 0.04]), colorbar
subplot(2,2,3), imagesc(cAV), axis('image'), caxis([0 0.04]), colorbar
subplot(2,2,4), imagesc(cAD), axis('image'), caxis([0 0.04]), colorbar
```



```
[cHA,cHH,cHV,cHD]=dwt2(cH,'db2');
subplot(2,2,1), imagesc(cHA), axis('image'), caxis([0 0.2]),  colorbar
subplot(2,2,2), imagesc(cHH), axis('image'), caxis([0 0.08]), colorbar
subplot(2,2,3), imagesc(cHV), axis('image'), caxis([0 0.08]), colorbar
subplot(2,2,4), imagesc(cHD), axis('image'), caxis([0 0.08]), colorbar
```



We see again large numbers of coefficients becoming small. Appropriate thresholding will provide a lossy compression of the image, but the obvious clear result is that selection of the wavelet basis is critical to obtaining the optimal compression.

Biorthogonal Wavelets

When we developed the theory of wavelet decomposition based on the Daubechies maxflat filters, we started with a polynomial in z that represented a low-pass power spectrum, and factored it into two 'identical' low-pass amplitude responses

$$P_k(z) = H(z)H(z^{-1})$$

where k is the order of the Daubechies wavelet, as in dbk. $P_k(z)$ is obtained by multiplying a binomial series by $B_k(x)$ by $(1-x)^k$ and substituting

$$x = \frac{1-\cos(\pi f/f_N)}{2} = \left(\frac{1-z}{2}\right)\left(\frac{1-z^{-1}}{2}\right)$$

Now what we are doing here is dividing the decomposition (analysis) filter and the reconstruction (synthesis) filter into two 'equal' parts, H(z) and its time reverse $H(z^{-1})$. Let's illustrate this process for db2. The scaling function for db2 is

$$\begin{aligned}\varphi(n) &= [0.6830 \quad 1.1830 \quad 0.3170 \quad -0.1830] \\ &= [1+\sqrt{3},\ 3+\sqrt{3},\ 3-\sqrt{3},\ 1-\sqrt{3}]/4\end{aligned}$$

and the low-pass filter kernel is given by

$$\begin{aligned}h(n) &= \text{fliplr}(\varphi(n))/\sqrt{2} \\ &= [-0.1294 \quad 0.2241 \quad 0.8365 \quad 0.4830] \\ &= 0.1294\delta(0) + 0.2241\delta(n-1) + 0.8365\delta(n-2) - 0.4830\delta(n-3) \\ &= [(1-\sqrt{3})\delta(0) + (3-\sqrt{3})\delta(n-1) + (3+\sqrt{3})\delta(n-2) + (1+\sqrt{3})\delta(n-3)]/(4\sqrt{2})\end{aligned}$$

$$\text{So} \quad H(z) = [(1-\sqrt{3}) + (3-\sqrt{3})/z + (3+\sqrt{3})/z^2 + (1+\sqrt{3})/z^3]/(4\sqrt{2})$$

$$\text{and} \quad H(z^{-1}) = [(1-\sqrt{3}),\ (3-\sqrt{3})z,\ (3+\sqrt{3})z^2,\ (1+\sqrt{3})z^3]/(4\sqrt{2})$$

Now let us try to recover $P_2(z)$. To do this we must multiply H(z) and $H(z^{-1})$, but in order to do so, we must have a consistent change in the power of z, so we flip $H(z^{-1})$ so that it descends in z, as does H(z)

$$\begin{aligned}H(z)H(z^{-1}) = {} & [0z^3 + 0z^2 + 0z + (1-\sqrt{3}) + (3-\sqrt{3})/z + (3+\sqrt{3})/z^2 + (1+\sqrt{3})/z^3]. \\ & [(1+\sqrt{3})z^3 + (3+\sqrt{3})z^2 + (3-\sqrt{3})z + (1-\sqrt{3}) + 0/z + 0/z^2 + 0/z^3]/32\end{aligned}$$

We can evaluate this easily in MATLAB using the conv function:

```
r3 = sqrt(3);
h1 = [0 0 0 1-r3 3-r3 3+r3 1+r3]
h2 = [1+r3  3+r3  3-r3 1-r3  0 0 0]
```

P2 = conv(h1,h2)/32
= [0 0 0 -2 0 18 32 18 0 -2 0 0 0]/32

So $P_2(z) = [0z^6+0z^5+0z^4-2z^3+0z^2+18z^1+32z^0+18z^{-1}+0z^{-2}-2z^{-3}+0z^{-4}+0z^{-5}+0z^{-6}]/32$

$P_2(z) = [-2z^3 + 18z^1 + 32z^0 + 18z^{-1} -2z^{-3}]/32$

Now in practice, we evaluate the two filtering operations separately, so $P_2(z)$ will start at z^0 rather than z^3 so we make P_2 causal by dividing by z^3 and the following results:

$P_2(z) = [-1z^0 + 9z^{-2} + 16z^{-3} + 9z^{-4} -1z^{-6}]/16$

Let's try to summarize what we have done. For the lowpass (approximation coefficients) filter, we start with a power spectrum and factor this in to two 'equal' filters. It turns out these two 'equal' filters correspond to decomposition and reconstruction, and their product results in the original power spectrum. The 'flipping' of the reconstruction kernel is equivalent to the transformation $H(z^{-1})$, not including a factor of z^n to make the kernel causal. For highpass, the process is just the same.

Now comes the big point. Notice that the coefficients of $P_2(z)$ is just integers divided by an integer power of 2. It would be very easy to implement this filter in DSP, because the division by powers of 2 is easily implemented by right shifts. However, we have to implement this filter using the kernels that contain roots of 3, which means we need floating point, or some sort of scaled integer arithmetic. It would be very nice if we could implement the filter efficiently in integer arithmetic. By the way, the above coefficients – integers divided by inger powers of two, are known as binary coefficients, and stem directly from the binomial series that was used to form the polynomial. It turns out there is a way, and the result leads to the formulation of biorthogonal wavelets.

We generalize the previous factorization that results in orthogonal wavelets

$$P_k(z) = H(z)H(z^{-1})/z^n$$

to one that results in biorthogonal wavelets

$$P_k(z) = H(z)F(z)$$

We can think of the H filter as the analysis (decomposition) filter and the F filter as the synthesis (reconstruction) filter, although as we work through some examples, the roles can be interchanged. In the orthogonal case we have $F(z)=H(z^{-1})/z^n$.

The simplest division is to make $H(z) = 1$ and $F(z) = P(z)$. This doesn't make for very interesting decomposition, because the input signal is now filtered. However, it gives us a starting point about which can generate new filters, so we start as follows using the power spectrum polynomial in z for db2:

$$P_2(z) = H_1(z)F_7(z) = 1.[-1z^0 + 9z^{-2} + 16z^{-3} + 9z^{-4} - 1z^{-6}]/16$$

The corresponding filter kernels are $h_1=1$ and $f_7 = [-1\ 0\ 9\ 16\ 9\ 0\ -1]/16$

You can see where the subscript comes from - it's the number of elements in the filter kernel. The filters as constructed are unbalanced, the construction part is doing all the work, so we remedy this situation by shifting some of the zeros in F to H. Recall that the zeros of $P_2(z)$ are factors of the form $(1+z^{-1})/2$. So we multiply H by this zeros, and divide F by the same:

$$H_2(z) = H_1(z)[1 + z^{-1}]/2$$

$$h_2(n) = [h_1(n) + h_1(n-1)]/2 = \left[1 \quad 1\right]/2$$

$$F_6(z) = F_7(z)/([1 + z^{-1}]/2)$$

$$f_6(n) = 2f_7(n) - f_6(n-1) = \left[-1 \quad 1 \quad 8 \quad 8 \quad 1 \quad -1\right]/8$$

The second filter is a little difficult to calculate, so we go through it step by step:

```
n  = -1  0  1  2  3  4  5  6
f7 = [0 -1  0  9 16  9  0 -1]/16
f6 = [0 -2  2 16 16  2 -2  0]/16

        -2 = 2(-1)-( 0)
         2 = 2(0) -(-2)
        16 = 2(9) -( 2)
        16 = 2(16)-(16)
         2 = 2( 9)-(16)
        -2 = 2( 0)-( 2)
         0 = 2(-1)-(-2)
```

The process can be repeated to move yet another zero from F to H. We iterate on h2 and f6, noting

$$h_{new}(n) = [h_{old}(n) + h_{old}(n-1)]/2$$

$$fnew(n) = 2f_{old}(n) - f_{new}(n-1)$$

So $h_3(n) = [h_2(n) + h_2(n-1)]/2 = \left[1 \quad 2 \quad 1\right]/4$

$f_5(n) = 2f_6(n) - f_5(n-1)] = \left[-1 \quad 2 \quad 6 \quad 2 \quad -1\right]/4$

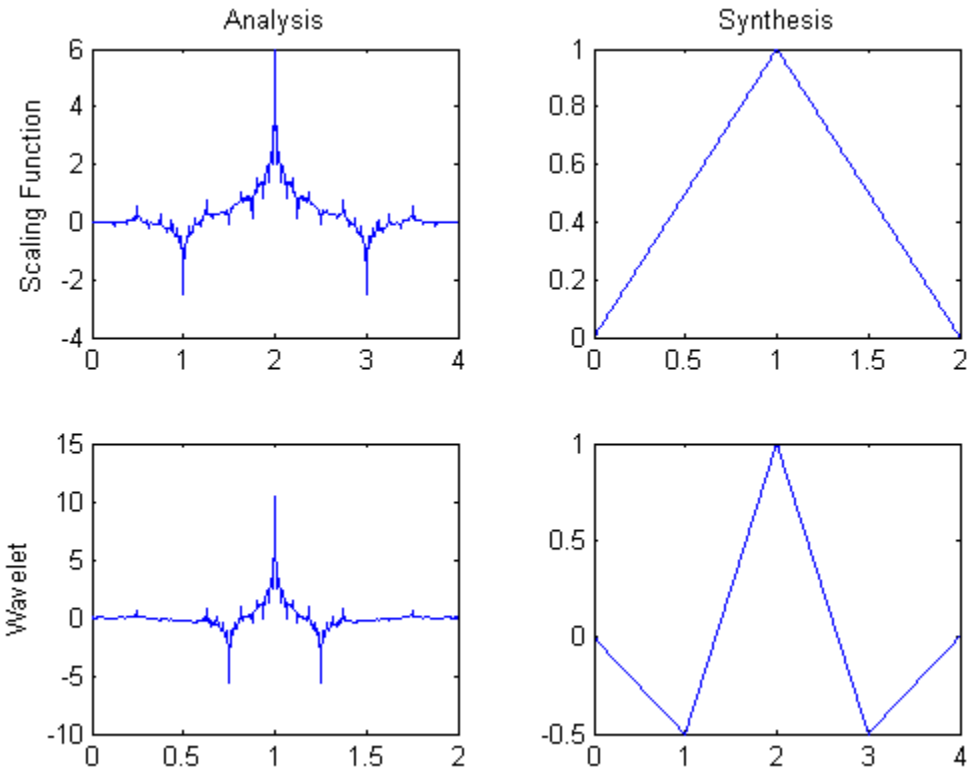
Note that `conv([1 2 1],[-1 2 6 2 -1])= [-1 0  9 16 9 0 -1]`

As mentioned before the F and H filters may be apportioned to the decomposition and reconstruction filter banks, or vice versa. It is common practice to place the f5 filter in the decomposition block, and h3 in the reconstruction block. This leads to a 5/3 filter.

However, the construction of the wavelet filter is a little different than before. The coefficients of the synthesis wavelet filter are essentially negated. Strang has both of the wavelet functions negated from what is shown below, which is fine since the net result is no sign change.

The following scaling functions and wavelets are obtained by iteration (10 iterations):

```
phi1f = 2*[1    2    1 ]/4;          phi1h = [-1 2 6 2 -1]/4;
phimh = phi1h;                       phimf = phi1f;
psi1h = fliplr(phi1f);               psi1f = fliplr(phi1h);

psi1h = psi1h.*((-1).^(1:(length(phi1h)-0)));
psi1f = psi1f.*((-1).^(0:(length(phi1f)-1)));

psimh = psi1h;                       psimf = psi1f;

for m=2:10
    phimuh = dyadup(phimh,0);        phimh  = conv(phimuh,phi1h);
    phimuf = dyadup(phimf,0);        phimf  = conv(phimuf,phi1f);
    psimuh = dyadup(psimh,0);        psimh  = conv(psimuh,phi1h);
    psimuf = dyadup(psimf,0);        psimf  = conv(psimuf,phi1f);
end

x1 = linspace(0,4,length(phimh));    x2 = linspace(0,2,length(phimf));
x3 = linspace(0,2,length(phimh));    x4 = linspace(0,4,length(phimf));

subplot(2,2,1), plot(x1,phimh), title('Analysis'), ylabel('Scaling Function')
subplot(2,2,2), plot(x2,phimf), title('Synthesis')
subplot(2,2,3), plot(x3,psimh), ylabel('Wavelet')
subplot(2,2,4), plot(x4,psimf)
set(gcf,'color', [1 1 1])
```



With more iterations, the synthesis filter becomes more spiky, in fact, it is infinite at binary values (integers divided by an integer power of 2).

You might also notice that in going from the low-pass filter kernel to the scaling function, we have multiplied the analysis coefficients by 2, but left the synthesis coefficients unscaled. Previously, we had distributed root 2 equally between each filter. Conversely, starting from the scaling functions, we make the analysis filter equal to the analysis scaling function, and divide the synthesis scaling function by 2. This saves having to carry factors of root 2 in the decomposition, which we want to avoid if we are doing integer math. The above biorthogonal scaling and wavelet functions can be derived from MATLAB using:

```
[phih,psih,phif,psif,xva1]=wavefun('bior2.2',1)

phih =   [-0.25  0.50  1.50  0.50 -0.25]
psih =   [-0.50  1.00 -0.50  ]

phif =   [ 0.50  1.00  0.50  ]
psif =   [-0.25 -0.50  1.50 -0.50 -0.25]

xva1 =   [ 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5]
```

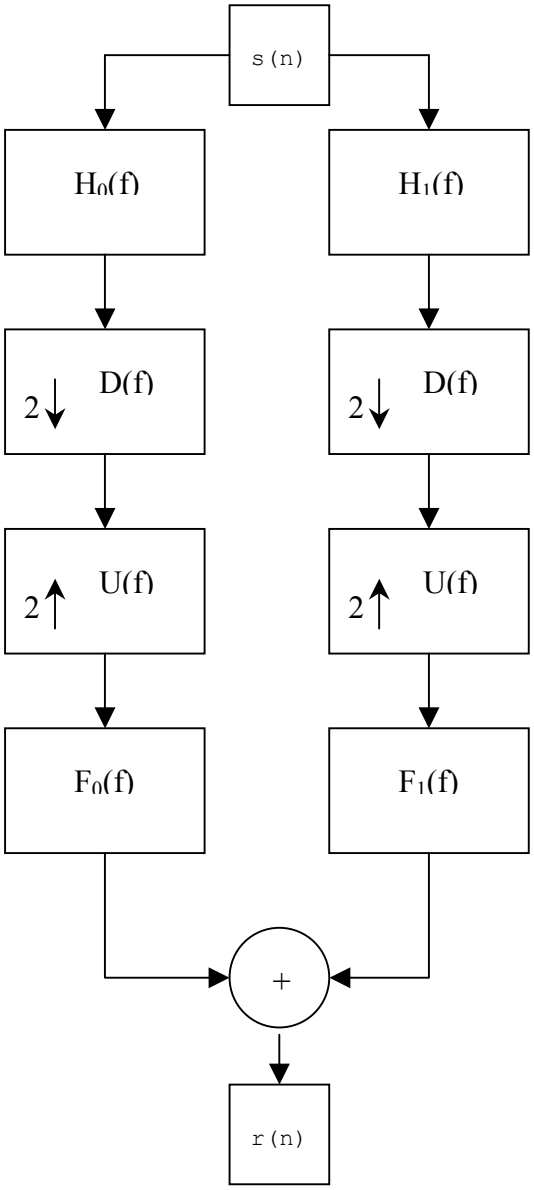
Compare these values to the 'hand' computations above (the 1s indicate first iteration, not decomposition or reconstruction):

```
phi1h =  -0.25    0.50    1.50    0.50   -0.25
psi1h =  -0.50    1.00   -0.50
phi1f =   0.50    1.00    0.50
psi1f =  -0.25   -0.50    1.50   -0.50   -0.25
```

In MATLAB, the biorthogonal wavelet is annotated by the number of zeros at z=-1 (which corresponds to $f=f_N$). The filter f7 starts out with 2k = 4 zeros at z=-1 for db2. f6 keeps three and shifts one to h2, resulting in bior1.3. f5 keeps two zeros and shifts two to h3, hence bior2.2. Other methods for forming biorthogonal wavelets will be examined next.

Biorthogonal Wavelets from Lifting

In order to understand lifting, we need to take a step back and review the process of perfect reconstruction. In the lieterature, this is abbreviated as PR. I'm going to use the same notation as Strang, so the lowpass filters have a subscript 0, and the highpass filters have a subscript 1 (instead of L and H), and the analysis filters are H, whereas the synthesis filters are F. Down sampling and upsampling are represented as D and U. Refer to previous notes on these frequency responses.



Imagine at first that the down-sampling and upsampling are not present, then in the z-domain, we would have

$$R(z) = S(z)[H_0(z)F_0(z) + H_1(z)F_1(z)]$$
$$R(z)/S(z) = [H_0(z)F_0(z) + H_1(z)F_1(z)]$$

Previously, we had set the right hand side to be 2. Since this is a constant, there is no amplitude distortion. We note that the products HF are analogous to power spectra. We also know that real filters have delay, so we expect y(n) to be the same as x(n) a certain number of time samples later (determined by the filter kernel lengths). Hence we will write

$$R(z)/S(z) = [H_0(z)F_0(z) + H_1(z)F_1(z)] = 2/z^m$$

For no distortion. We have, however, removed the downsampling followed by upsampling. If you recall from our previous notes, when we take a signal x(n), down sample it and then upsample it, we end up with a new signal y(n) whose spectrum is calculated as follows:

$$Y(f)=[X(f)+X(f-1)]/2$$

We have normalized frequency with respect to Nyquist frequency, so we could also write:

$$Y(f/f_N)=[X(f/f_N)+X((f-f_N)/f_N)]/2$$

In the z-domain, we can write things more simply because

$$\exp(j\pi(f-f_N)/f_N)=\exp(j\pi(f)/f_N)\exp(-j\pi)=-z$$

So

$$Y(z)=[X(z)+X(-z)]/2$$

So lets trace the signal through the lowpass leg of the filter in the z-domain. It starts out as $S(z)$ and is lowpass filtered to become

$$S(z)H_0(z)$$

It is downsampled and upsampled, so it becomes

$$[S(z)H_0(z)+S(-z)H_0(-z)]/2$$

At this point, we accept that the signal is probably aliased: no matter, we can compensate for this later.

Then it is filtered by the synthesis filter, so it becomes

$$[S(z)H_0(z)F_0(z)+S(-z)H_0(-z)F_0(z)]/2$$

Likewise, in the highpass leg, we have

$$[S(z)H_1(z)F_1(z)+S(-z)H_1(-z)F_1(z)]/2$$

So when these are summed, the result becomes

$$R(z)=[S(z)H_0(z)F_0(z)+S(-z)H_0(-z)F_0(z)]/2 + [S(z)H_1(z)F_1(z)+S(-z)H_1(-z)F_1(z)]/2$$

$$R(z) = S(z)[H_0(z)F_0(z) + H_1(z)F_1(z)]/2 + S(-z)[H_0(-z)F_0(z) + H_1(-z)F_1(z)]/2$$

So now we get both results in 1, and you can see where the factor of 2 comes in: we make

$$[H_0(z)F_0(z) + H_1(z)F_1(z)] = 2/z^m \qquad (1)$$

and

$$[H_0(-z)F_0(z) + H_1(-z)F_1(z)] = 0 \qquad (2)$$

and we end up with

$$R(z)/S(z) = 1/z^m$$
$$r(n) = s(n-m)$$

The second condition is known as the condition for no-aliasing. The filters we have looked at all meet these criteria (no distortion and no aliasing). We can combine the two equations into a single equation for the design of the lowpass filter, by eliminating F_1:

From (2) above $F_0(z) = -H_1(-z)$ or $H_1(z) = -F_0(-z)$ and $F_1(z) = +H_0(-z)$ or $H_0(z) = +F_1(-z)$

so

$$[H_0(z)F_0(z) + (-F_0(-z))(H_0(-z))] = 2/z^m$$

$$[H_0(z)F_0(z) - H_0(-z)F_0(-z)] = 2/z^m$$

We define the product $P_0(z) = H_0(z)F_0(z)$ as the lowpass power spectrum. We need:

$$P_0(z) - P_0(-z) = 2/z^m$$

Not just any power spectrum will do, it has to have this symmetric property. For example, here is a power spectrum (it happens to be the db2 power polynomial) that does meet the condition:

$$P_0(z) = [-1z^0 + 0z^{-1} + 9z^{-2} + 16z^{-3} + 9z^{-4} + 0z^{-5} - 1z^{-6}]/16$$
$$P_0(-z) = [-1z^0 - 0z^{-1} + 9z^{-2} - 16z^{-3} + 9z^{-4} - 0z^{-5} - 1z^{-6}]/16$$
$$P_0(z) - P_0(-z) = [0z^0 + 0z^{-1} + 0z^{-2} + 32z^{-3} + 0z^{-4} + 0z^{-5} + 0z^{-6}]/16 = 2/z^3$$

The odd powers of z have to be zero, except the 'center' one at z^{-m}. The result is stated differently when the polynomial is made symmetric about z^0, which makes the filter kernel symmetric about n=0 and non-causal

$$z^m P_0(z) - z^m P_0(-z) = P(z) + P(-z) = 2$$

We are defining P(z) as essentially the same polynomial, only z^0 occurs in the middle, not at the start. The change in sign on P(-z) is not so obvious. Let's work the above example again to make sure it is correct.

$$P(z) = [-1z^{3} + 0z^{2} + 9z^{1} + 16z^{0} + 9z^{-1} + 0z^{-2} - 1z^{-3}]/16$$

$$P(-z) = [+1z^{3} + 0z^{2} - 9z^{1} + 16z^{0} - 9z^{-1} + 0z^{-2} + 1z^{-3}]/16$$

$$P(z) + P(-z) = [0z^{3} + 0z^{2} + 0z^{1} + 32z^{0} + 0z^{-1} + 0z^{-2} + 0z^{-3}]/16 = 2$$

Once we have $P_0(z)$ we can factor it into $H_0(z)$ and $F_0(z)$ and from there derive the high-pass filters $H_1(z)$ and $F_1(z)$ as follows:

$$[H_0(-z)F_0(z) + H_1(-z)F_1(z)] = [-H_0(-z)H_1(-z) + H_1(-z)H_0(-z)] = 0$$

At this point, we will do well to check this against our previous known result of db2.

```
phi =  [ 0.6830   1.1830   0.3170  -0.1830]
psi =  [-0.1830  -0.3170   1.1830  -0.6830]
```

Starting with $P_0(z) = [-1z^{0} + 9z^{-2} + 16z^{-3} + 9z^{-4} - 1z^{-6}]/16$, we factor this into

$$P_0(z) = H_0(z)F_0(z) = H_0(z)H_0(z^{-1})$$

Note, we are following Strang notation, here, previously we used the subscript in P to identify order of db, now we are saying 0 represents lowpass.

$$H_0(z) = [-0.1830 + 0.3170/z + 1.1830/z^{2} + 0.6830/z^{3}]/\sqrt{2} = \mathrm{fliplr}(\Phi(z))/\sqrt{2}$$

$$F_0(z) = H_0(z^{-1}) = [-0.1830 + 0.3170z + 1.1830z^{2} + 0.6830z^{3}]/\sqrt{2}$$

$$F_0(z) = z^{3}[0.6830 + 1.1830/z + 0.3170/z^{2} - 0.1830/z^{3}]/\sqrt{2}$$

$$F_1(z) = H_0(-z) = [-0.1830 - 0.3170/z + 1.1830/z^{2} - 0.6830/z^{3}]/\sqrt{2}$$

$$H_1(z) = -F_0(-z) = -[0.6830 - 1.1830/z + 0.3170/z^{2} + 0.1830/z^{3}]/\sqrt{2}$$

$$H_1(z) = [-0.6830 + 1.1830/z - 0.3170/z^{2} - 0.1830/z^{3}]/\sqrt{2}$$

Compare this result to

```
[H0,H1,F0,F1] = wfilters('db2')

H0 = -0.1830    0.3170    1.1830    0.6830
H1 = -0.6830    1.1830   -0.3170   -0.1830

F0 =  0.6830    1.1830    0.3170   -0.1830
F1 = -0.1830   -0.3170    1.1830   -0.6830
```

We are now in a position to do lifting. First we restate the condition for perfect reconstruction (PR)

$$[H_0(z)F_0(z) - H_0(-z)F_0(-z)] = 2/z^{m}$$

m is an odd integer. (I'll try to prove this later).

We keep F fixed and form a new filter $H^{\#}$ as follows:

$$H_0^{\#}(z) = H_0(z) + F_0(-z)S(z^2)$$

This is the lifting theorem, originally introduced by William Sweldens et al. We need to prove that this new lowpass analysis filter satisfies the PR condition, so we substitute the expression back into the PR condition:

$$[H_0^{\#}(z)F_0(z) - H_0^{\#}(-z)F_0(-z)] \stackrel{?}{=} 2/z^m$$

$$[(H_0(z) + F_0(-z)S(z^2))F_0(z) - (H_0(-z) + F_0(+z)S((-z)^2))F_0(-z)] \stackrel{?}{=} 2/z^m$$

$$H_0(z)F_0(z) + F_0(-z)F_0(z)S(z^2) - H_0(-z)F_0(-z) + F_0(+z)F_0(-z)S(z^2) \stackrel{?}{=} 2/z^m$$

$$H_0(z)F_0(z) - H_0(-z)F_0(-z) \stackrel{yes}{=} 2/z^m$$

The tricky part to doing lifting is finding $S(z^2)$ – it seems that any $S(z^2)$ will work: I will show one way of finding a permissible function.

Next we show how to lift $P_0(z) = [-1z^3 + 9z^1 + 16z^0 + 9z^{-1} - 1z^{-3}]/16$. We start by factoring this into $H_0(z)F_0(z)$ but make the analysis (decomposition) be lazy by setting $H_0(z) = 1$. This means

$$F_0(z) = [-1z^3 + 9z^1 + 16z^0 + 9z^{-1} - 1z^{-3}]/16$$

In terms of filter kernels, we have

$$h_1(n) = [1]$$

$$f_7(n) = [-1\ 0\ 9\ 16\ 9\ 0\ -1]/16$$

Note here that the subscripts on the kernels indicates the number of terms (including zero values) in the kernel. This combination is known as a 1/7 filter – it is not useful in and of itself. We lift H, keeping F fixed:

$$H_0^{\#}(z) = H_0(z) + F_0(-z)S(z^2)$$

$$H_0(z) = 1$$

$$F_0(z) = [-1z^3 + 9z^1 + 16z^0 + 9z^{-1} - 1z^{-3}]/16$$

$$F_0(-z) = [1z^3 - 9z^1 + 16z^0 - 9z^{-1} + 1z^{-3}]/16$$

$S(z^2) = [z + z^{-1}]/4$ (we show where this comes from later)

$$F_0(-z)S(z^2) = [1z^4 + 0z^3 - 8z^2 + 16z^1 - 18z^0 + 16z^{-1} - 8z^{-2} + 0z^{-3} + 1z^{-4}]/64$$

The coefficients above can be found as follows:

```
conv([1 0 -9 16 -9 0 1],[1 0 1]) = [1 0 -8 16 -18 16 -8 0 1]
```

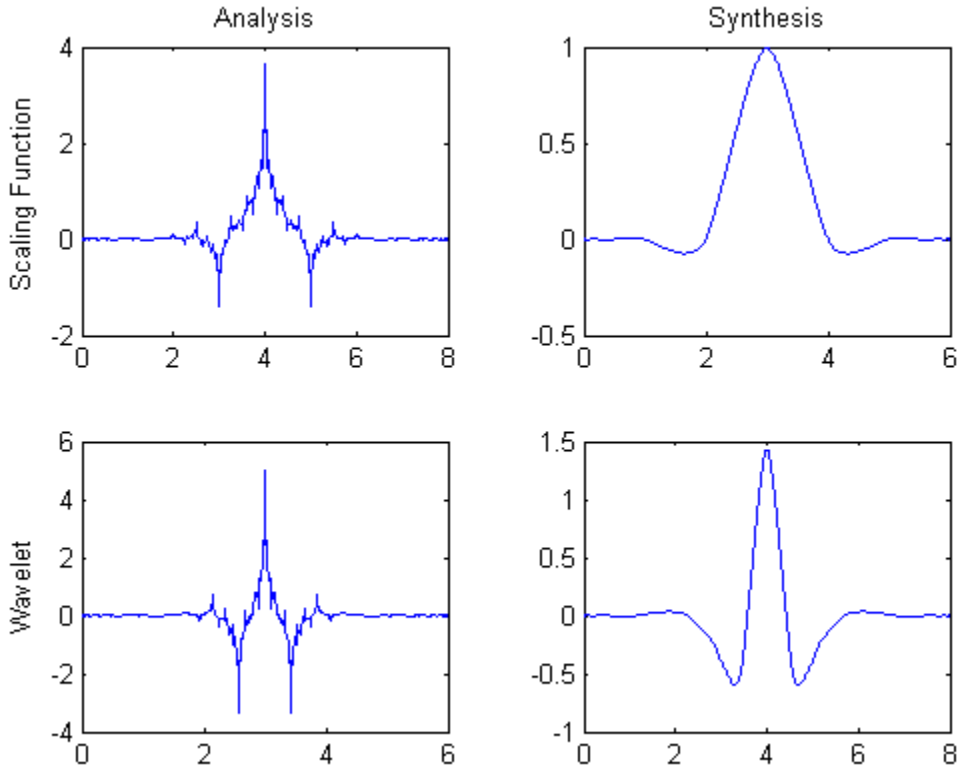
$$H_0^{\#}(z) = 64z^0/64 + [1z^4 + 0z^3 - 8z^2 + 16z^1 - 18z^0 + 16z^{-1} - 8z^{-2} + 0z^{-3} + 1z^{-4}]/64$$

$$H_0^{\#}(z) = [1z^4 + 0z^3 - 8z^2 + 16z^1 + 46z^0 + 16z^{-1} - 8z^{-2} + 0z^{-3} + 1z^{-4}]/64$$

$$h_9(n) = [1\ 0\ -8\ 16\ 46\ 16\ -8\ 0\ 1]/64$$

This method of lifting has expanded the filter kernel rather than just distributing the power spectrum between analysis and synthesis. Longer filter kernels produce more zeros when signals are input, thus producing better compression. Note this 9/7 filter is not the same has the 9/7 filter produced by MATLAB's bior4.4. f7 has 4 roots at z=-1 and h9 has 2 roots at z=-1. This can easily be checked using the MATLAB `roots` function (just substitute the kernel coefficients into the function).

How did we figure $S(z^2)$? You need something that is a function of z^2. The simplest is $S(z^2)=1$, but this is not very interesting, so the next choice would be $S(z^2)=1+1/z^2$, which when balanced becomes $S(z^2)=z+0+1/z$. Other choices are possible – it's not clear that they lead to good wavelets. The 9/7 biorthogonal scaling functions and wavelets look like this



An important property of the 9/7 filter is that it is reversible with integer inputs. This means that there will be no quantization due to round off – making for completely lossless compression of integer signals and images.

Formulation of the JPEG 2000 9/7 Filter.

The 9/7 filter that is used in JPEG 2000 is different from the 9/7 filter outlined above, in that it has irrational coefficients. The Construction of these coefficients is described very briefly in a paper entitled "Image Coding Using the Wavelet Transform" by Marc Antonini and other authors (including Ingrid Daubechies). It was published in the IEEE Transactions on Image Processing in April 1992. Using Strang's notation rather than that of the paper, the lowpass power function in the frequency domain is taken from a general class of functions that satisfy the PR conditions:

$$P_0(\Omega) = H_0(\Omega)\,F_0(\Omega) = \cos(\Omega/2)^{2m}\left[\sum_{p=0}^{m-1}\binom{m-1+p}{p}\sin(\Omega/2)^{2p}\right]$$

where $\Omega = \pi f / f_N$

For the 9/7 filter, m=4, so the above equation becomes

$$P_0(\Omega) = H_0(\Omega)\,F_0(\Omega) = \cos(\Omega/2)^{8}\left[\sum_{p=0}^{3}\binom{3+p}{p}\sin(\Omega/2)^{2p}\right]$$

Noting that $z = e^{j\pi f/f_N} = e^{j\Omega}$, we see that

$$\cos(\Omega/2) = [e^{j\Omega/2} + e^{-j\Omega/2}]/2 = [z^{0.5} + z^{-0.5}]/2$$

$$\cos(\Omega/2)^2 = [z + 2 + z^{-1}]/4$$

$$\sin(\Omega/2) = [e^{j\Omega/2} - e^{j\Omega/2}]/2j = [z^{0.5} - z^{-0.5}]/2j$$

$$\sin(\Omega/2)^2 = [-z + 2 - z^{-1}]/4$$

So

$$H_0(\Omega)\,F_0(\Omega) = \frac{[z+2+z^{-1}]^4}{256}\left[\sum_{p=0}^{3}\binom{3-p}{p}\frac{[-z+2-z^{-1}]^p}{4^p}\right]$$

The products can be done using convolution. For example

$[z+2+z^{-1}] = [1\ 2\ 1]$

$[z+2+z^{-1}]^2 = \text{conv}([1\ 2\ 1],[1\ 2\ 1]) = [1\ 4\ 6\ 4\ 1]$

$[z+2+z^{-1}]^4 = \text{conv}([1\ 4\ 6\ 4\ 1],[1\ 4\ 6\ 4\ 1]) = [1\ 8\ 28\ 56\ 70\ 56\ 28\ 8\ 1]$

The summation terms above wok out as follows:

```
P=0    [  0   0    0    1    0   0   0]1/1
P=1    [  0   0   -1    2   -1   0   0]4/4
P=2    [  0   1   -4    6   -4   1   0]10/16
P=3    [ -1   6  -15   20  -15   6  -1]20/64
Σ      [-10  80 -262  416 -262  80 -10]/32
```

So we see $H_0(\Omega)F_0(\Omega)$ as the product of two polynomials:

```
p1=[1 8 28 56 70 56 28 8 1]
p2=[-10  80 -262  416 -262  80 -10]
```

The goal is to distribute the roots of the second polynomial evenly between `H` and `F`.

```
Q=roots(p2)= 3.0407
             2.0311 + 1.7390i
             2.0311 - 1.7390i
             0.2841 + 0.2432i
             0.2841 - 0.2432i
             0.3289

poly(Q)= [1.0 -8.0 26.2 -41.6 26.2 -8.0 1.0]
```

We note that we have lost a factor of –10; the minus must be re-included in one of the factorizations.

The complex roots are put in the decomposition filter, and the real roots go in the reconstruction filters

```
poly([Q(1),Q(6)]) = [1.0000  -3.3695   1.0000]
poly(P(2:5)) =      [1.0000  -4.6305   9.5975  -4.6305  1.0000]
```

The polynomials `[1 4 6 4 1]` are shared equally between the two filters, so the filter coefficients become

```
f7 = conv([1.000 -3.3695 1.000],[1 4 6 4 1])
h9 = conv([1.000 -4.6305 9.5975 -4.6305 1.000],[1 4 6 4 1])

f7 = [1.0  0.6305 -6.4780 -12.2170 -6.478 0.6305  1.00]
h9 = [1.0 -0.6305 -2.9245   9.9765 22.541 9.9765 -2.9245 -0.6305 1.0]
```

In the Antonini paper the filter coefficients are normalized to 1. We reinsert the missing –1 back into the f7 polynomial by dividing by its sum, which is negative:

```
f7 = f7/sum(f7)
   = [-0.0456 -0.0288  0.2956  0.5575 0.2956 -0.0288 -0.0456]

h9 = h9/sum(h9)
   [0.0267 -0.0169 -0.0782 0.2669 0.6030 0.2669 -0.0782 -0.0169 0.0267]
```

The Antonini paper lists the above numbers as the filter coefficients scaled by $1\sqrt{2}$, and hence the actual filter kernels are the same as those produced by the MATLAB function `wfilters`.

```
[LO_D,HI_D,LO_R,HI_R] = wfilters('bior4.4') ;

[LO_D',    HI_D',    LO_R',    HI_R']

     0         0         0         0
 0.0378   -0.0645   -0.0645   -0.0378
-0.0238    0.0407   -0.0407   -0.0238
-0.1106    0.4181    0.4181    0.1106
 0.3774   -0.7885    0.7885    0.3774
 0.8527    0.4181    0.4181   -0.8527
 0.3774    0.0407   -0.0407    0.3774
-0.1106   -0.0645   -0.0645    0.1106
-0.0238         0         0   -0.0238
 0.0378         0         0   -0.0378
```

Our results, h9 and f7, when multiplied by $\sqrt{2}$ become LO_D and LO_R respectively. HI_D and HI_R can be constructed using the PR criterion stated earlier. HI_D is simply LO_R with the even MATLAB indices negated, and HI_R is simply LO_D with the odd MATLAB indices negated.

In the JPEG 2000 standard, the approximation filtering is done with h9 and 2f7 rather than $\sqrt{2}$h9 and $\sqrt{2}$f7. This filter has good compression properties, even though its irrational coefficients are a little harder to deal with.

DWT by Lifting

In this section, we look specifically at the implementation of the 5/3 DWT in JPEG 2000, which is done using lifting. We obtained the 5/3 binary coefficient filter by balancing the lowpass power polynomial for db2 `[-1 0 9 16 9 0 -1]/16`, which resulted in `f3` = [`1 2 1]/4 and h5=[-1 2 6 2 -1]/4`. These are the synthesis and analysis filter kernels of the low-pass section. The corresponding highpass kernels are `[1 -2 1]/4` and `[-1 -2 6 -2 -1]/4` respectively.

JPEG200 has this a little bit different: in MATLAB notation, they have:

```
[LO_D',HI_D',LO_R',HI_R'] = [-1   -4   4  -1
                              2    8   8  -2
                              6   -4   4   6
                              2    0   0  -2
                             -1    0   0  -1 ]/8
```

whereas in MATLAB

```
[LO_D',HI_D',LO_R',HI_R'] = [-1    2   2   1
                              2   -4   4   2
                              6    2   2  -6
                              2    0   0   2
                             -1    0   0   1]/sqrt(2)/4
```

the difference is a sign change in the highpass coefficients and the distribution of the factor of root 2. We should keep this in mind when we compare results with MATLAB.

First we write the recursion equations that occur in the JPEG 2000 Still Image Coding System paper

$$y(2n+1)=x_{ext}(2n+1)-\left\lfloor \frac{x_{ext}(2n)+x_{ext}(2n+2)}{2} \right\rfloor$$

$$y(2n)=x_{ext}(2n)+\left\lfloor \frac{y(2n-1)+y(2n+1)+2}{4} \right\rfloor$$

In a slightly different form. First we lift the restriction that we are dealing with integers, so there is no need to round down, and as a result, the +2 in the numerator of the second term in the second equation (a compensating factor for rounding down) can be removed. Further, we invoke the notation used by Daubechies and Sweldens in their 1996 paper on factoring and lifting, and drop the 'ext' subscript, understanding that the x data are appropriately extended in some way.

$$d(n)=y(2n+1)=x(2n+1)-\left\lfloor \frac{x(2n)+x(2n+2)}{2} \right\rfloor$$

$$s(n)=y(2n)=x(2n)+\left\lfloor \frac{d(n-1)+d(n)}{4} \right\rfloor$$

Recognizing that $x(2n)$ is the even points in $x(n)$, which we denote as $x_e(n)=x(2n)$, and that $x(2n+1)$ is the (the set of) odd points, which we denote $x_o(n)=x(2n+1)$, then we could also write

$$d(n) = y_o(n) = x_o(n) - \left[\frac{x_e(n)+x_e(n+1)}{2}\right]$$

$$s(n) = y_e(n) = x_e(n) + \left[\frac{d(n-1)+d(n)}{4}\right]$$

Notice that `d` and `s` are simply odd and even points in the output vector `y`. It turns out that d is the detail coefficients and `a` the approximations.

At this point, we do a numerical example to illustrate the computation, and then try to illustrate the significance of the result.

Let `x(n) = [0 2 3 1 2 1]` in which case we have $x_e(n) = [0\ 3\ 2]$ and $x_o(n) = [2\ 1\ 1]$

Note we are using DSP indexing, so `n=0,1,2,3,…` d and s are computed in a leap-frogging manner – in fact we are using previous computations of d to help us compute s. Lets start with n=0:

$$d(0) = x_o(0) - \left[\frac{x_e(0)+x_e(1)}{2}\right] = 2 - \frac{0+3}{2} = \frac{1}{2}$$

To compute `s(0)`, we need to know `d(-1)`

$$d(-1) = x_o(-1) - \left[\frac{x_e(-1)+x_e(0)}{2}\right]$$

We have to assume something about $x_o(-1)$ and $x_e(-1)$ in order to solve this, so we assume a zero-padded extension, and they are both zero. In this case

$$d(-1) = 0 - \frac{0+x_e(0)}{2} = 0 - \frac{0+0}{2} = 0$$

(Note that the result is zero here because the first point in `x`, `x(0)` happens to be zero.) Hence

$$s(0) = x_e(0) + \left[\frac{d(-1)+d(0)}{4}\right] = 0 + \frac{0+0.5}{4} = \frac{1}{8}$$

Thus we have two points in our output vector y(0) and y(1). We compute the next 2 points:

$$d(1) = x_o(1) - \left[\frac{x_e(1)+x_e(2)}{2}\right] = 1 - \frac{3+2}{2} = \frac{-3}{2}$$

$$s(1) = x_e(1) + \left[\frac{d(0)+d(1)}{4}\right] = 3 + \frac{0.5-1.5}{4} = \frac{11}{4}$$

And the next two

$$d(2) = x_o(2) - \left[\frac{x_e(2)+x_e(3)}{2}\right] = 1 - \frac{2+0}{2} = 0$$

$$s(2) = x_e(2) + \left[\frac{d(1)+d(2)}{4}\right] = 2 - \frac{-1.5+0}{4} = \frac{13}{8}$$

Notice that we again have to assume something about the nature of x beyond the given data, so we assume zeros.

Now this result is different from what we would get in MTALAB, because of the sign inversion in the high-pass filter, and because of an amplitude scaling factor of 2, so we delay a comparison until later.

To summarize, the input and output vectors x and y are, respectively, as follows:

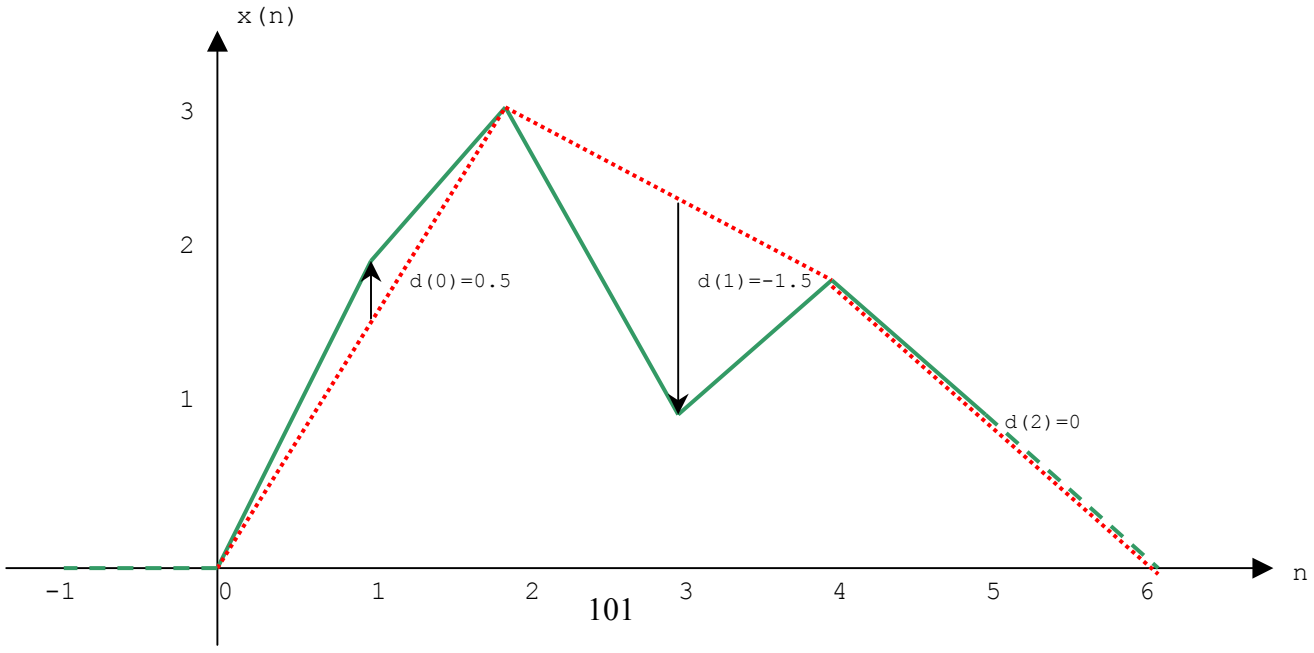
```
x(n) = [ 0   2   3    1    2    1]
y(n) = [ 1/8 1/2 11/4 -3/2 13/8 0]
```

The even points are approximations and the odd points are differences. The differences are smaller than the approximations – and very nicely the last point happens to be zero, which makes for good compression.

Now we attempt to explain what the filter is doing in terms of prediction and updating. It turns out that the what we are trying to do is predict the odd values from the even values, recording the prediction error as the difference, and at the same time to compute a moving average (the approximations) in order to compensate the prediction and provide perfect reconstruction. If this sounds a little confusing, cast your mind back to the Haar wavelet: the high pass filter [1 –1] computes differences, and the lowpass filter [1 1] computes a moving average. In the 5/3 filter, we use the computation of the differences to help us 'update' the computation of the moving average.

The original data sequence Let `x(n) = [0 2 3 1 2 1]` is plotted as the solid line:

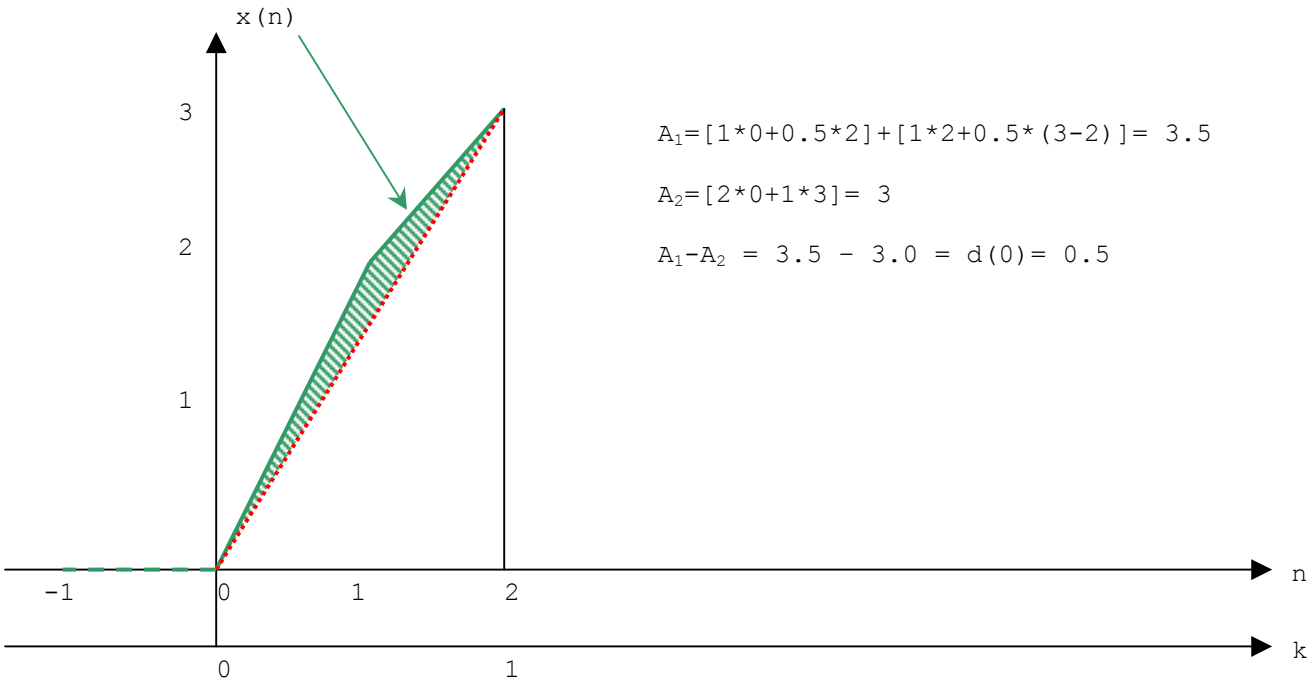
The detail coefficients (along with adjacent even data points) are plotted as a dotted (red) line, the original data are plotted as the solid (green) line (the extension of x(n) is plotted as a dashed (green) line). Note that d(-1)=0 is included,



We see that d(0)=0.5 is the difference between x(1)=2 and what we would predict x(1) to be based on the nearest even coefficients. A simple estimator of x(1) based on even coefficients is (x(0)+x(2))/2=1.5. Likewise, d(1) is the difference between x(3)=1, and its estimator based on x(2) and x(4), i.e., (x(2)+x(4))/2=2.5. The same applies to d(2): we were lucky that in this case that the difference happens to be zero, but this is the general trend; to try to drive the differences down to zero.

What about the approximations? Details are differences (high pass filtering), approximations are moving averages (low-pass filtering) and averages correspond to integration, or area finding.

```
y(n) = [ 1/8 1/2 11/4 -3/2 13/8 0]
```



A_1=[1*0+0.5*2]+[1*2+0.5*(3-2)]= 3.5

A_2=[2*0+1*3]= 3

A_1-A_2 = 3.5 - 3.0 = d(0)= 0.5

The difference in area corresponds to d(0) as measured on the original data index n=0,1,2. However, the averages (approximations) are calculated every other point. Therefore we scale the above result by a factor of ½ and compute the change in area as d(0)/2 = 0.5/2 = 0.25. To 'compensate' this change, we add half of d(0)/2, i.e. d(0)/4 to the adjacent even data points. This has the effect of making the area under the s curve the same as the area under the original data.

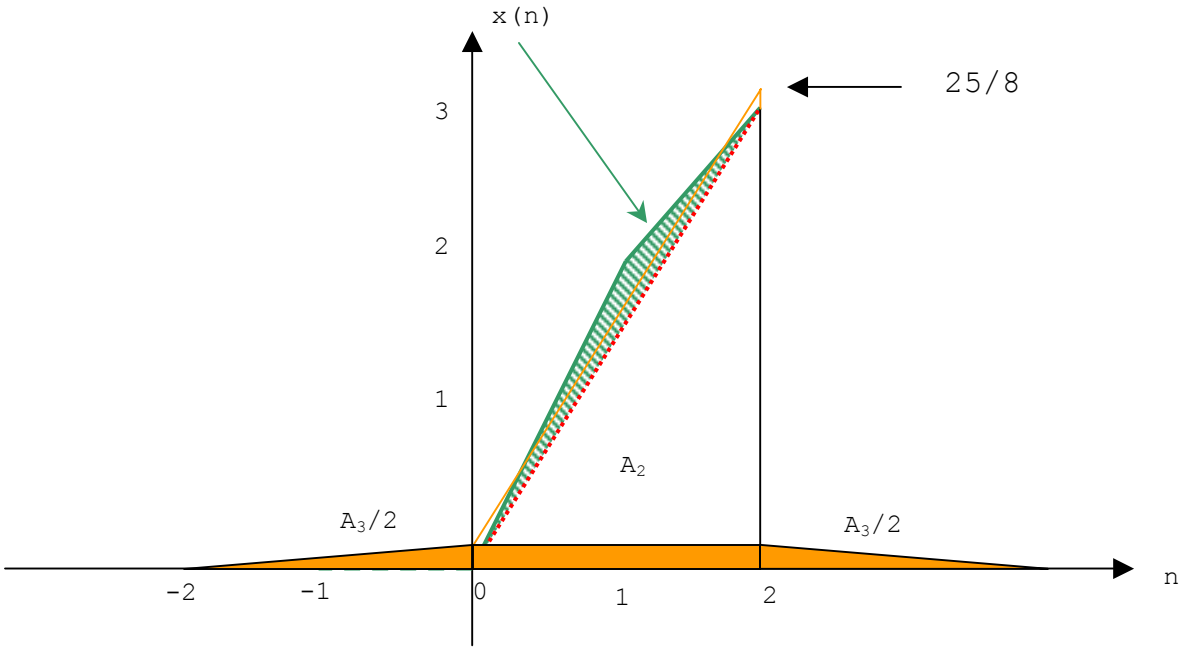
This is a little difficult to see, so first we note that by adding 1/8 to each of the even points adjacent to x(1), we add area = 2(1/8)=1/4 immediately to the interval [0,2]. To the next set of adjacent even points x(-2) and x(4), we add nothing, but we do affect the area [-2,0] and [2,4], with a linear taper from 1/8 to each end point. This adds an extra 1/4, so we have added 1/4+1/4=1/2

A_1=[[1*0+0.5*2]+[1*2+0.5*(3-2)]]= 3.5

A_2=[2*0.125+2*0.5*(3.125-0.125)]= 3.25

A_3=2*(2*0.5*0.125) = 0.25

A_1 – (A_2 + A_3) = 0.



After the first iteration

d(0) = y(1) = 1/2
s(0) = y(0) = 0 → 0 + 1/8 = 1/8
s(1) = y(2) = 3 → 3 + 1/8 = 25/8

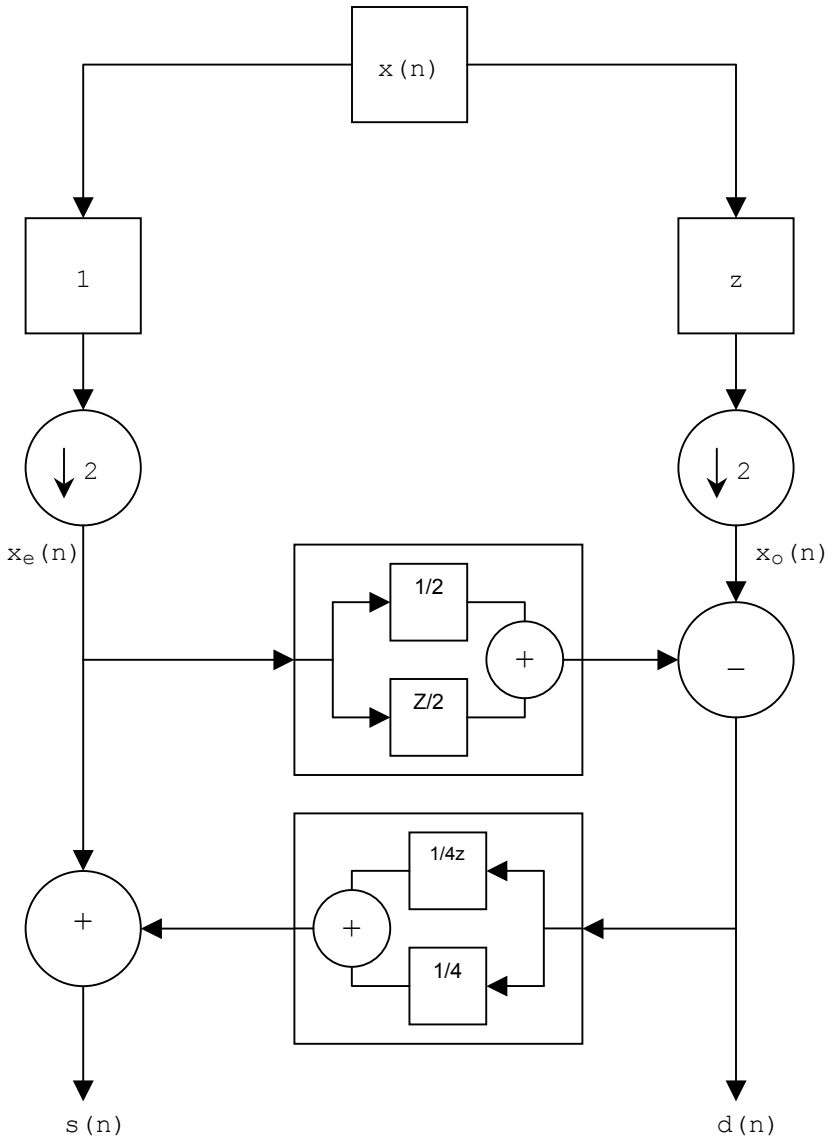
After the second iteration

d(1) = y(3) = -3/2
s(1) = y(2) = 25/8 → 25/8 - 3/8 = 22/8
s(2) = y(4) = 2 → 2 - 3/8 = 13/8

After the third iteration

d(2) = y(5) = 0
s(2) = y(4) = 13/8 → 13/8 + 0 = 13/8
s(3) = y(6) = 0 → 0 + 0 = 0

The decomposition filter architecture for the 5/3 lifting scheme can be summarized as follows:



The reconstruction proceeds in a similar manner.