An Introduction to Wavelet Transforms and Data Compression Using Mathematica.

Alkiviadis G. Akritas

University of Thessaly
Department of Computer and Communication Engineering
GR-38221, Volos
Greece

Wavelet Transforms trace their origin both to Signal Processing and Theoretical Mathematics. Since their introduction they have found applications in many areas — most notably in fingerprinting by the FBI, where they are used to compress fingerprint data before storing it. Using *Mathematica* as a paradigm we present an introduction to these transforms and their application and demonstrate the main ideas with a picture of Pedro. The book by George Nakos and the papers by Colm Mulcahy and Gilbert Strang were inspirational.

Uniform and Adaptive Plotting of Functions

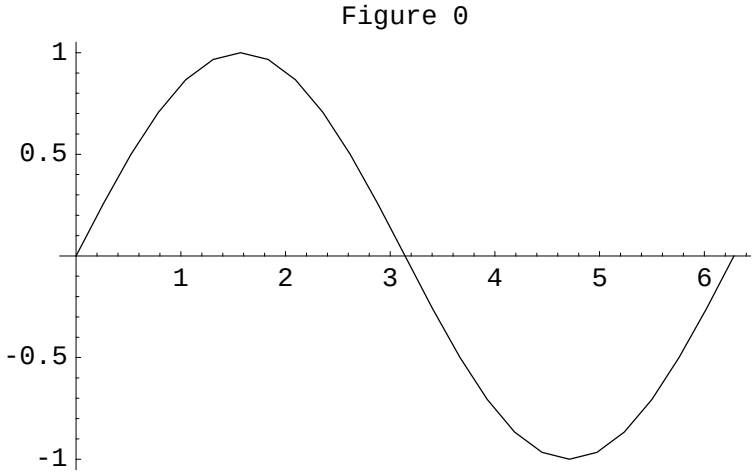
Plotting various functions with a computer algebra system like *Mathematica* is an activity quite similar to signal processing; to wit, in both cases we take samples. In this section we review standard ways of plotting discrete data sets and two dimensional digital images. The inherent difficulties of plotting functions by uniform sampling will lead us to adoptive plotting techniques (the main idea of which is at the heart of wavelet transforms) and to techniques based on wavelets (see the section on Image Compression with the Haar Transform). We also indicate the need for data compression.

■ Uniformly distributed sample points

Suppose we have a set of n planar data points $\{x_i, y_i\}$, i = 1, ..., n, which are samples of the function $y = f(x)$. A common way to plot $f(x)$ from this set of points is to plot these points individually and then join adjacent points with line segments. This can be achieved with the *Mathematica* function ListPlot — used with option PlotJoined→True**.**

Earlier versions of *Mathematica* used — in their Plot function — a fixed number of (just 25) points x_i uniformly distributed in the interval $[a, b]$. As an example, in Figure 0 below we see a "darn good" plot of $\sin(x)$ using ListPlot.

```
xPoints0 = Table[x, {x, 0, 2 π, 2 π / 24}];
yPoints0 = Map[Sin[ #] &, xPoints0];
ListPlot[Transpose[{xPoints0, yPoints0}],
  PlotJoined → True, PlotLabel → "Figure 0"];
Print["Sin[x] in the interval [0, 2π];
    25 sample points were used"];
```

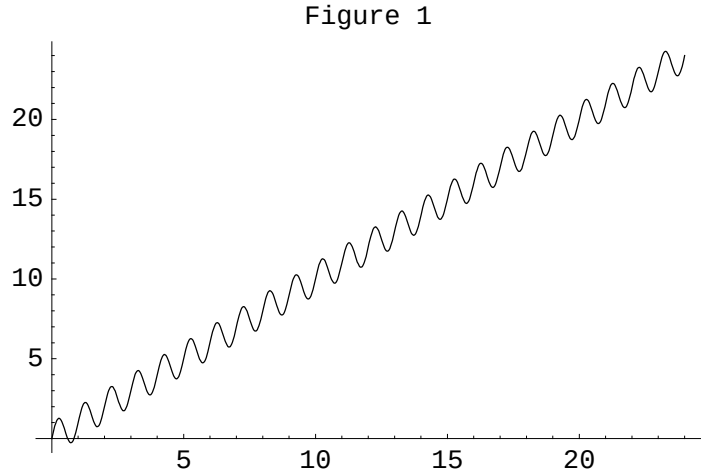



```
Sin[x] in the interval [0, 2π];  25 sample points were used
```

However, this practice of using a fixed number of points x_i uniformly distributed in the interval [a, b] leads to serious problems as we see in the following examples. Please note that the plots of Figures 2 and 4 below (obtained using the function ListPlot) could be obtained using the function Plot of older versions of *Mathematica* — drawing, at the time, a lot of flak.

Consider, first, the function $f(x) = x + \sin(2\pi x)$, in the interval $0 \le x \le 24$. Its plot is shown in Figure 1 below:

```
Plot[x + Sin[2 π x], {x, 0, 24}, PlotLabel → "Figure 1"];
Print["The function x+Sin[2πx] in the interval [0, 24]"];
```

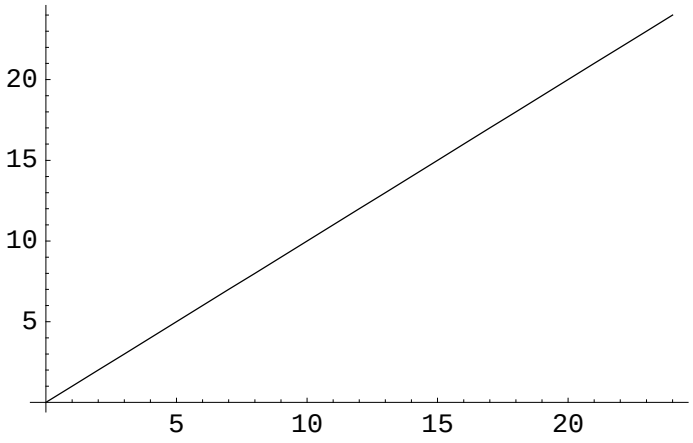



```
The function x+Sin[2πx] in the interval [0, 24]
```

Using the uniform sampling approach mentioned above, and sampling at the points x = 0, 1, 2, ... 24, we get the straight line shown in Figure 2.

```
xPoints1 = Table[x, {x, 0, 24}];
yPoints1 = Map[# + Sin[2 π #] &, xPoints1];
ListPlot[Transpose[{xPoints1, yPoints1}],
  PlotJoined → True, PlotLabel → "Figure 2"];
Print["Aliased version of the function x+
    Sin[2πx]; 25 sample points were used"];
```

Figure 2



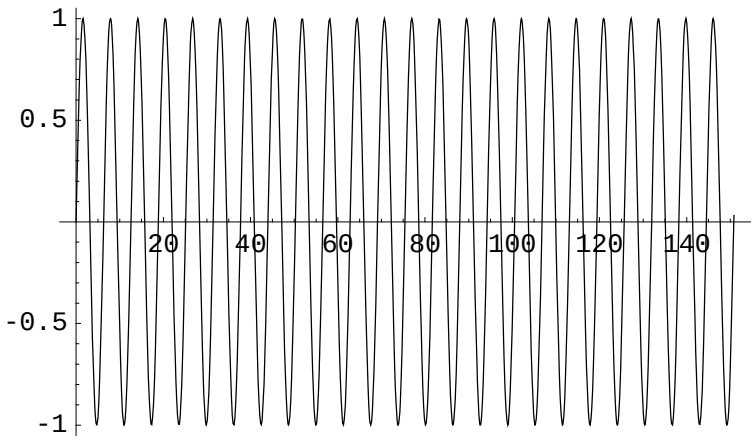

```
Aliased version of the function x+Sin[2πx]; 25 sample points were used
```

In the vernacular of signal analysis this is called *aliasing*, whereby a slow frequency appears instead of a higher frequncy because the discrete samples are the same. In our example we get an "aliased" version of the desired function. The reason for aliasing is that we did *not* use enough sample points (and at the points where we did sample, i.e. x = 0, 1, 2, ... 24, we have $x = x + \sin(2\pi x)$).

As a second example of aliasing consider the function $\sin(x)$, in the (storied) interval $0 \le x \le 48.01\pi$. Its true nature is shown in Figure 3.

```
Plot[Sin[x], {x, 0, 48.01 π}, PlotLabel → "Figure 3"];
Print["The function Sin[x] in the interval [0, 48.01π]"];
```

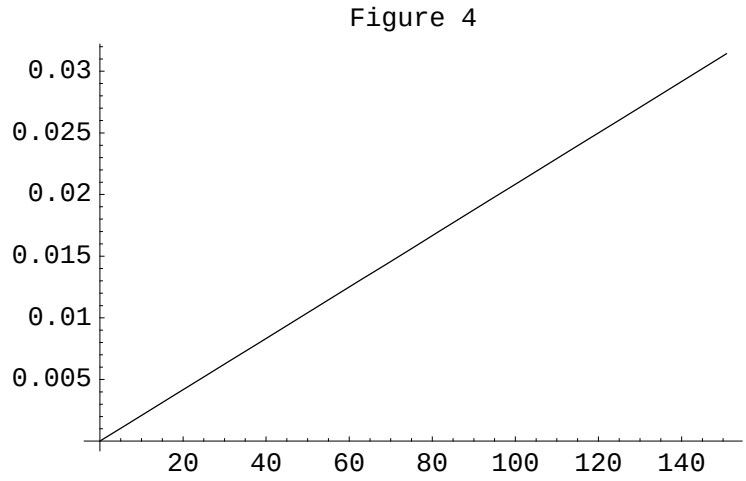
Figure 3




```
The function Sin[x] in the interval [0, 48.01π]
```

However, using only 25 uniformly sampled points in the interval [0, 48.01π] we obtain the alias of Figure 4.

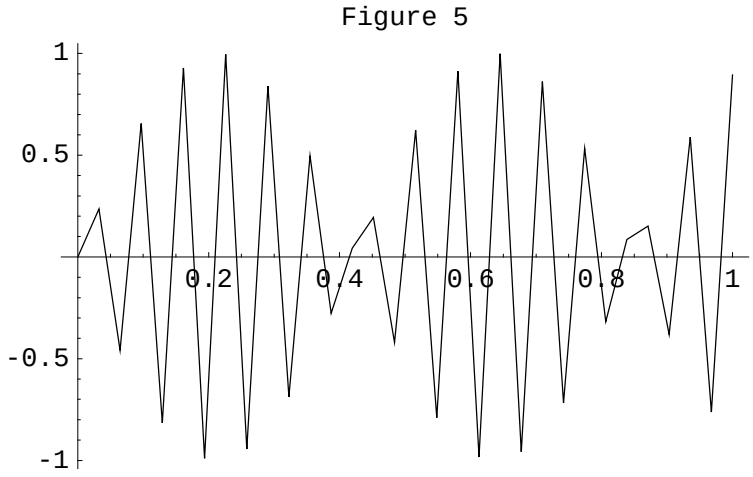
```
xPoints2 = Table[x, {x, 0, 48.01 π, 48.01 π / 24}];
yPoints2 = Map[Sin[ #] &, xPoints2];
ListPlot[Transpose[{xPoints2, yPoints2}],
  PlotJoined → True, PlotLabel → "Figure 4"];
Print["Aliased version of the function Sin[x] in the
    interval [0, 48.01π];  25 sample points were used"];
```




```
Aliased version of the function Sin[x] in
  the interval [0, 48.01π];  25 sample points were used
```

A final example of aliasing is obtained when we try to plot the function sin(90x) using only 32 uniformly spaced sample points. Instead of a horizontally telescoped version of the sin(x) function we obtain the pulsing pattern shown in Figure 5.

```
xPoints3 = Table[x, {x, 0, 1, 1 / 31}];
yPoints3 = Map[Sin[90 #] &, xPoints3];
ListPlot[Thread[{xPoints3, yPoints3}],
  PlotJoined → True, PlotLabel → "Figure 5"];
Print["Aliased version of the function Sin[90x] in the
   interval [0, 1]; 32 uniformly sampled points were used"]
```




```
Aliased version of the function Sin[90x] in the
  interval [0, 1]; 32 uniformly sampled points were used
```

Some definitions are needed now in order to state the famous theorem explaining alias. Consider the sinusoid $x(t) = \sin(\omega t)$. This repeats whenever the time t is increased by $T = \frac{2\pi}{\omega}$, its *period*. Suppose T is measured in seconds. Then the numbers of cycles in one second is $f = \frac{1}{T}$. Thus, the frequency in Hertz is $f = \frac{1}{T} = \frac{\omega}{2\pi}$. We can now state the following:

Shannon's Sampling Theorem: Every analog signal $x(t)$ with frequencies not exceeding $\omega_{\max}$ can be perfectly reconstructed from its discrete samples $x(nT)$, provided the sampling rate $\frac{1}{T}$ exceeds $2 f_{\max} = \frac{\omega_{\max}}{\pi}$ (Nyquist rate).

Note that the critical Nyquist rate has two samples per oscillation! Also, reconstruction requires a strict inequality -- the Nyquist rate cuts it too close.

According to Shannon's theorem, the first two examples fail because we have one sample per oscillation. In the third example, we observe that we have enough sample points for the function $\sin(90x)$, since $32 > 2 f_{\max} = \frac{\omega_{\max}}{\pi} = \frac{90}{\pi} = 28.6479$ (Nyquist rate), but nonetheless aliasing appears. This is not unusual, and using a few more points yields the anticipated plot.

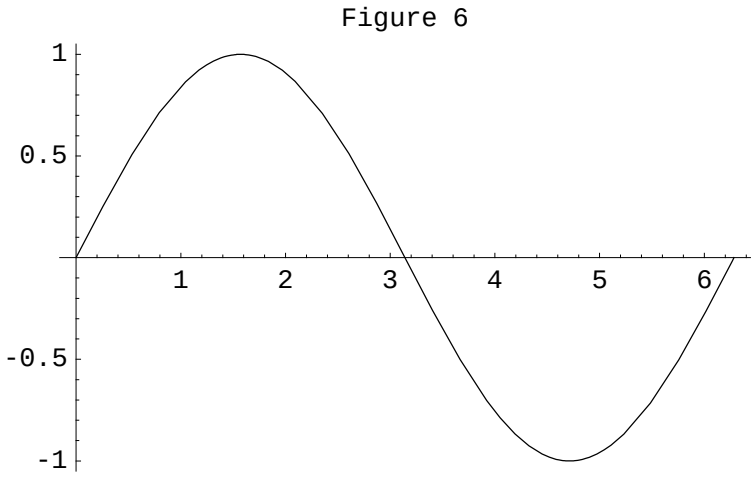
■ Adoptive plotting

From the above examples it is obvious that uniformly spaced sample points is not the right way to go, unless we are prepared to use a lot of points; this is done in the next section where we use a large number of uniformly spaced sample points together with wavelet transforms for "data compression".

To correct the errors described in the above examples, adoptive plotting (or sampling) techniques are used with points clustered where the function exhibits great variation. These routines initially generate an internal plot based on uniform sampling and they examine the angles between the connecting line segments in order to identify regions of great variation. Having identified regions of great variation, they next subdivide the corresponding intervals further before producing the visible plot.

This is precisely what happens when the current version *Mathematica* plots the function sin(x), $0 \le x \le 2\pi$. As we can see 82 sample points were used to make the plot of Figure 6.

```
n = 0;
xlow = 0;
xhigh = 2 π;

Plot[++n; Sin[x], {x, xlow, xhigh}, PlotLabel → "Figure 6"];
Print["For this plot ", n,
 " sample points were used in the interval [",
 xlow, ", ", xhigh, "]"]
```




```
For this plot 82 sample points were used in the interval [0, 2 π]
```

Note that these 82 points are *not* uniformly distributed in the interval [0, 2π]. Letting *xhigh* vary successively from 1 to 6 we see that in the interval (0, 1] there were 25 sample points, in the interval (1, 2] there were 29 sample points, in *each* of the intervals (2, 3], and (3, 4] there was 1 sample point, in the interval (4, 5] there were 24 sample points and finally, in the interval (5, 6] there were 2 sample points.

We can also look at these sample points. For example to plot sin(x) in the interval [0, 0.5] the following points were used:

```
n = 0;
xlow = 0;
xhigh = 0.5;

Plot[Print["n = ", ++n, ", x[", n, "] = ", x]; Sin[x],
  {x, xlow, xhigh}, PlotPoints → 5, DisplayFunction → Identity];
```

```
n = 1, x[1] = 1.25×10^-7

n = 2, x[2] = 0.121701

n = 3, x[3] = 0.254426

n = 4, x[4] = 0.379078

n = 5, x[5] = 0.498955

n = 6, x[6] = 0.5
```

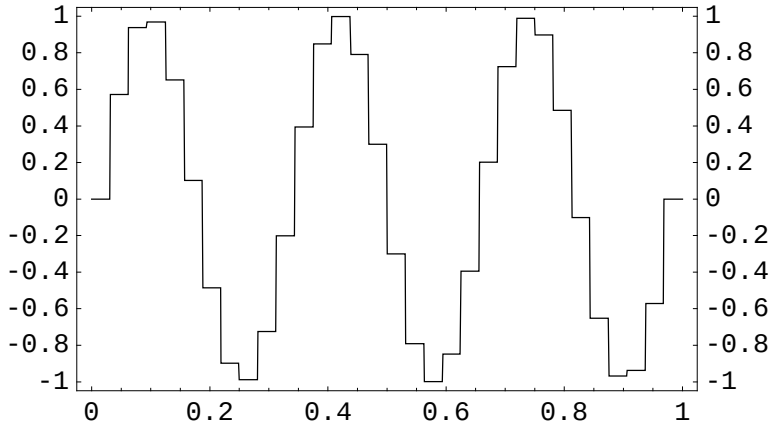
To recap: When there is no variation we need few points to recover the plot of a function and, conversly, when there is variation we need many points. This point will be used in the wavelet transforms; to wit, if a digital image is "smooth" (mostly black or white) then we can recover the image keeping very few of the "details" nonzero.

■ Step functions

Finally, as a prelude to the discussion on wavelets we mention that linear interpolation of uniformly sampled points is *not* the only way of plotting; instead of joining the points with line segments, we could use the *y*-values as step levels for a staicase effect. The function $\sin(6\pi x)$ is plotted this way in Figure 7.

```
fi[x_] := Which[0 ≤ x < 1, 1, True, 0]; fi[x_, i_, k_] := fi[2^i x - k];
xPoints4 = Table[x, {x, 0, 1, 1 / 31}];
yPoints4 = Map[Sin[6 π #] &, xPoints4];
yStep4 = Table[fi[t, 5, k], {k, 0, 31}];
Plot[yPoints4.yStep4, {t, 0, 1}, PlotLabel → "Figure 7",
 Axes → {False, False}, Frame → True, FrameTicks → {Automatic,
   {-1, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1}}];
Print["Step function version of the function Sin[6πx] in
    the interval [0, 1]; 32 sample points were used"];
```

Figure 7




```
Step function version of the function Sin[6
  πx] in the interval [0, 1]; 32 sample points were used
```

This should come as no surprise since we know from Calculus that continuous functions on [0, 1] can be approximated to any degree of precision with linear functions or by step functions. These staircase approximations lead to better and better plots as the number of sample points increases, although a lot more are needed to obtain the continuous effect. In this context questions of data storage and transmission arise. These questions become crucial when we explore digital images which are higher-dimensional analogues of data points in the plane.

Black and white digital images are derived from two-dimensional arrays of *pixels.* A pixel is the smallest unit of space in a digital image. Associated with each pixel there is a number, in the range [0, 1], representing gray levels ranging from black (0) to white (1). So now our data points are triplets $\{x, y, z\}$, where z measures the gray level at pixel position $\{x, y\}$. Viewed from above we have two-dimensional step functions, where the steps are shaded according to their height. (Color images can be dealt with by decomposing them into their red, green and blue components and treating each of these like grayscales.) When 256 ($= 2^8$) levels of gray are used, we are talking about 8-*bit images.*

Digital images come in various resolutions. An image with resolution 1600×1200 pixels is derived from a matrix of 1600×1200 = 1,920,000 pieces of data, each representing a gray level. Clearly, images with such high resolution are difficult to store and time-consuming to transmit over low band line. Therefore, we should try to come up with a more economical way to store the matrices that represent these images. This is achieved by taking advantage of regions in the image where there is little or no variation, (for example, a black coat in the picture).

Signals and Wavelet Transforms

Most signals start their lives in analog form. They become digital by sampling at equal time intervals, $x_{\text{digital}}(n) = x_{\text{analog}}(nT)$, where $n = 0, \pm1, \pm2, \ldots$ and T is the *sampling interval.*

The Discrete Time Fourier Transform turns the samples $x_{\text{digital}}(n) = x(n)$ into the coefficients of a 2π -periodic function $\chi(\omega) = \sum_{n=-\infty}^{\infty} x(n)\, e^{-i n \omega}$. We refer to ω as the *frequency*, and we plot the transform $\chi(\omega)$ in the interval $\omega = -\pi$ and $\omega = \pi$. Then "low frequencies" refer to frequencies near zero, and "high frequencies" have $|\omega| \approx \pi$.

Notice that the signal $x(n) = \{\ldots, 1, 1, 1, 1, 1, \ldots\}$ has exactly zero frequency ($\omega = 0$, the lowest frequency), whereas the alteration between 1 and -1 gives the signal $x(n) = \{\ldots, 1, -1, 1, -1, 1, \ldots\}$ with the highest frequency $\omega = \pm\pi$. In between these two extremes we have the family of pure sinusoidal signals $x(n) = e^{i n \omega}$ with frequency $0 < |\omega| < \pi$.

■ Lowpass and Highpass Filters

Sampling, *transforming* and *filtering* constitute the basic signal processing operations. Of the three, filtering (and filters) will be of importance to us.

Filters are also called *linear time-invariant* systems. When presented with the input $x_\omega(n)$ these systems produce the output $H(\omega)\, x_\omega(n)$, where the amplifying factor $H(\omega)$ is the *frequency response.* Hence, filters act on signals to produce modified signals and this is the most important operation in signal processing. Below we examine filters that are: (a) *finite impulse response* (FIR), meaning each output is a linear combination of a *finite* number of input samples, (b) *time-invariant*, meaning their coefficients are constant for all time, and (c) *causal*, meaning they involve no future samples like $x(n+1)$. (Note that for images the situation is different because n refers to position, not time. Therefore, once the complete image is available, filters in image processing need not be causal.)

Lowpass Filter: The first example is the moving average filter

```
yLP[n_] := 1/2 x[[n - 1]] + x[[n]]/2
```

We are going to consider three special inputs, the impulse, the constant and the alternating signal. For the impulse $x(n) = \{\ldots, 0, 0, 1, 0, 0, \ldots\}$ the output is $y_{\text{LP}}(n) = \{\ldots, 0, 0, \frac{1}{2}, \frac{1}{2}, 0, \ldots\}$; that is, the impulse response contains the filter coefficients.

```
Clear[x]; x = Insert[Table[0, {n, 1, 19}], 1, 10]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

```
Table[yLP[n], {n, 2, 20}]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 1/2, 1/2, 0, 0, 0, 0, 0, 0, 0, 0}
```

For the constant signal $x(n) = \{\ldots, 1, 1, 1, 1, 1, \ldots\}$ the output is $y_{\text{LP}}(n) = \{\ldots, 1, 1, 1, 1, 1, \ldots\}$. That is, the response exactly equals the input; the frequency $\omega = 0$ is in the *passband*. Taking a look at a finite number of samples we have:

```
x = Table[1, {n, 1, 20}]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

```
Table[yLP[n], {n, 2, 20}]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

On the other hand, for the alternating signal $x(n) = \{..., 1, -1, 1, -1, 1, ...\}$ the output is $y_{LP}(n) = \{..., 0, 0, 0, 0, 0, ...\}$. In other words, the response is zero; the frequency $\omega = \pm\pi$ is in the *stopband*.

```
x = Table[(-1)^n, {n, 1, 20}]
```

```
{-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1}
```

```
Table[yLP[n], {n, 2, 20}]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

For the in between frequencies, i.e. $0 < \omega < \pi$, consider the input signal $x(n) = e^{i n \omega}$. The output then is

$$y_{LP}(n) = \frac{1}{2} e^{i n \omega} + \frac{1}{2} e^{i(n-1)\omega} = (\frac{1}{2} + \frac{1}{2} e^{-i\omega}) e^{i n \omega}$$

In this case, the output frequency is the same ω, but the filter multiplies each frequency component of the input by the *frequency response* function $H(\omega) = \frac{1}{2} + \frac{1}{2} e^{-i\omega}$.

As we have seen,, at $\omega = 0$ the frequency response is $H = 1$; i.e. the constant signal passes unchanged through the filter. At $\omega = \pi$ the frequency response is $H = 0$; that is, the alternating signal is completely blacked out. Writing the response function as

$$H(\omega) = \frac{1}{2} + \frac{1}{2} e^{-i\omega} = e^{-i\omega/2}(\frac{1}{2} e^{i\omega/2} + \frac{1}{2} e^{-i\omega/2}) = e^{-i\omega/2} \cos \frac{\omega}{2}.$$

we see that the amplitude is $|H(\omega)| = \cos \frac{\omega}{2}$, which drops from one at $\omega = 0$ to zero at $\omega = \pi$. This is the *transition band* for the filter.

Highpass Filter: The second example is the moving difference filter

```
yHP[n_] := 1/2 x[[n]] - 1/2 x[[n - 1]]
```

For the impulse $x(n) = \{..., 0, 0, 1 ,0 , 0, ...\}$ the output is $y_{HP}(n) = \{..., 0, 0, \frac{1}{2}, -\frac{1}{2}, 0, ...\}$; that is, the impulse response again contains the filter coefficients.

```
Clear[x]; x = Insert[Table[0, {n, 1, 19}], 1, 10]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

```
Table[yHP[n], {n, 2, 20}]
```

$$\left\{0, 0, 0, 0, 0, 0, 0, 0, \frac{1}{2}, -\frac{1}{2}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right\}$$

For the constant signal $x(n) = \{..., 1, 1, 1, 1, 1, ...\}$ the output is $y_{HP}(n) = \{..., 0, 0, 0, 0, 0, ...\}$. That is, the lowest frequency $\omega = 0$ is stopped. Taking a look at a finite number of samples we have:

```
x = Table[1, {n, 1, 20}]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

```
Table[yHP[n], {n, 2, 20}]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

On the other hand, for the alternating signal $x(n) = (-1)^n$ the output is $y_{HP}(n) = (-1)^n$. In other words, the frequency $\omega = \pi$ passes without any change.

```
x = Table[(-1)^n, {n, 1, 20}]
```

```
{-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1}
```

```
Table[yHP[n], {n, 2, 20}]
```

```
{1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1}
```

For the in between frequencies, i.e. $0 < \omega < \pi$, consider again the input signal $x(n) = e^{i n \omega}$. The output then is

$$y_{HP}(n) = \frac{1}{2} e^{i n \omega} - \frac{1}{2} e^{i(n-1)\omega} = (\frac{1}{2} - \frac{1}{2} e^{-i\omega}) e^{i n \omega}.$$

The multiplying factor is the frequency response function $H(\omega) = \frac{1}{2} - \frac{1}{2} e^{-i\omega}$. Again we write as

$$H(\omega) = \frac{1}{2} - \frac{1}{2} e^{-i\omega} = e^{-i\omega/2}(\frac{1}{2} e^{i\omega/2} - \frac{1}{2} e^{-i\omega/2}) = e^{-i\omega/2}\, i \sin \frac{\omega}{2},$$

in which case the amplitude is $|H(\omega)| = \sin \frac{\omega}{2}$, which rises from zero at $\omega = 0$ to one at $\omega = \pi$. This is the transition band for the filter.

In summary, a lowpass filter preserves the *smooth* part of a signal, whereas a highpass filter preserves the *rough and noisy* part. In wavelet language, a lowpass filter gives averages whereas a highpass filter gives details. In some applications those details are important to keep (like edges in an image) whereas in other applications the high frequencies are mostly noise from measurements.

■ Convolutions

A good lowpass filter has many uses, so we look now at the (FIR, linear time-invariant and causal) filter with N+1 coefficients $h(0)$, $h(1)$, ..., $h(N)$. This is a filter of *order N* with $N+1$ coefficients. At each time step the $N+1$ coefficients multiply $N+1$ samples from the input signal – starting with the current sample $x(n)$ and going back to the sample $x(n-N)$. Hence, the output is:

$$y(n) = h(0)\,x(n) + h(1)\,x(n-1) + \cdots + h(N)\,x(n-N).$$

We may write the action of the filter in the time domain compactly as the sum:

$$y(n) = \sum_{k=0}^{N} h(k)\,x(n-k).$$

Convolution is the fundamental operation of discrete time-invariant systems and *filters are discrete convolutions!* This convolution is easily implemented in hardware using three building blocks: *unit delays, multipliers and adders.*

Convolution is also easily implemented in softwrae. For example, the filter $h = \{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}$ is lowpass, since the alternating signal $\omega = \pi$ is eliminated

```
yLP[n_] := 1/4 x[[n]] + 1/2 x[[n - 1]] + 1/4 x[[n - 2]]
```

```
Clear[x]; x = Table[(-1)^n, {n, 1, 20}]
```

```
{-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1}
```

```
Table[yLP[n], {n, 3, 20}]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

whereas the constant signal is preserved. (Note that the filter coefficients sum up to 1.)

```
x = Table[1, {n, 1, 20}]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

```
Table[yLP[n], {n, 3, 20}]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

Note again how the impulse response contains the filter coefficients.

```
x = Insert[Table[0, {n, 1, 19}], 1, 10]
```

```
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

```
Table[yLP[n], {n, 3, 20}]
```

$$\left\{0, 0, 0, 0, 0, 0, 0, \frac{1}{4}, \frac{1}{2}, \frac{1}{4}, 0, 0, 0, 0, 0, 0, 0, 0\right\}$$

The interesting part starts when the input is the linear signal x = {0, 1, 2, 3, 4}. Padding the signal with two zeros on both ends (since the filter is of order $N = 2$) we have:

```
x = {0, 0, 0, 1, 2, 3, 4, 0, 0}; Table[yLP[n], {n, 3, 9}]
```

$$\left\{0, \frac{1}{4}, 1, 2, 3, \frac{11}{4}, 1\right\}$$

Note that the signal is preserved at the center!

We can also easily implement lowpass filters using the ListConvolve function of *Mathematica* (see also the Help Browser for this function). As expected, the result is the same as the one obtained above.

```
ListConvolve[{1/4, 1/2, 1/4}, {0, 1, 2, 3, 4}, {1, -1}, 0]
```

$$\left\{0, \frac{1}{4}, 1, 2, 3, \frac{11}{4}, 1\right\}$$

Observe that a unit *delay* is introduced between input and output. Notice that five terms convolved with three terms produce seven terms. We convolved an input signal of length $L = 5$, with a second order filter, $N = 2$. The output is of length $L + N = 7$, which is reasonable given that a fourth degree polynomial, $0 + x + 2\,x^2 + 3\,x^3 + 4\,x^4$, times a second degree polynomial, $\frac{1}{4} + \frac{1}{2}\,x + \frac{1}{4}\,x^2$ yields a sixth degree polynomial, $0 + \frac{1}{4}\,x + x^2 + 2\,x^3 + 3\,x^4 + \frac{11}{4}\,x^5 + x^6$.

It is extermely interesting to see how ListConvolve works. The linear signal is *inverted* and padded with $N = 2$ zeros at both ends to obtain the list {0, 0, 4, 3, 2, 1, 0, 0, 0}. Starting from the right end we form the dot products $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}.\{0, 0, 0\} = 0$, $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}.\{1, 0, 0\} = \frac{1}{4}$, $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}.\{2, 1, 0\} = 1$, $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}.\{3, 2, 1\} = 2$, $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}.\{4, 3, 2\} = 3$, $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}.\{0, 4, 3\} = \frac{11}{4}$, and $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}.\{0, 0, 4\} = 1$ which result in the coefficients of the sixth degree polynomial. The thing to note here is that inversion is associated with convolution!

Note the end effects at both ends because we do not have the samples $x(-2)$ and $x(-1)$ for $y(0)$ and $y(1)$ on the left and $x(5)$ and $x(6)$ for $y(5)$ and $y(6)$ on the right.

The above remarks apply when the input is the sinusoidal signal x = {0, 1, 0, -1, 0} with frequency $\omega = \frac{\pi}{2}$. We observe that it, too, is preserved with a unit delay between input and output; additionally, the amplitude of the output has changed to $\frac{1}{2}$.

```
x = {0, 0, 0, 1, 0, -1, 0, 0, 0}; Table[yLP[n], {n, 3, 9}]
```

$$\left\{0, \frac{1}{4}, \frac{1}{2}, 0, -\frac{1}{2}, -\frac{1}{4}, 0\right\}$$

or

```
ListConvolve[{1/4, 1/2, 1/4}, {0, 1, 0, -1, 0}, {1, -1}, 0]
```

$$\left\{0, \frac{1}{4}, \frac{1}{2}, 0, -\frac{1}{2}, -\frac{1}{4}, 0\right\}$$

```
Clear[x];
```

To see that the amplitude is reduced by a half let us look at the frequency response function $H(\omega)$. For the filter under consideration we have

$$H(\omega) = \frac{1}{4} + \frac{1}{2} e^{-i\omega} + \frac{1}{4} e^{-2i\omega}.$$

Evaluating at $\omega = \frac{\pi}{2}$ we obtain $H = \frac{-1}{2} i$, from which we have that the magnitude of the frequency is $\frac{1}{2}$. (Recall that from Fourier transforms we have that convolutions in the time domain correspond to multiplications in the frequency domain.)

■ Wavelet Transforms

As we said, lowpass filters greatly reduce the high frequency components; when these components represent noise in signals, this reduction is the right thing to do. However, if we want to *reconstruct* the signal, then we use two filters, a highpass as well as a lowpass filter. This generates a *filter bank* structure, which leads to a *Discrete Wavelet Transform* (DWT) – in itself a *lossless* transform. Its inverse, (IDWT), is another transform of the same type – two filters that are fast to compute. Between the DWT and the IDWT compression and transmission of the signal can take place, and this is the key to more and more applications.

Wavelet transforms are associated with *multiresolution into different scales.* The simplest change of scale comes from *downsampling* a signal, to wit keeping only its even-numbered components $y(2n)$. This operation is denoted by the symol $\downarrow 2$:

$$(\downarrow 2)\begin{pmatrix} y(1) \\ y(2) \\ y(3) \\ y(4) \end{pmatrix} = \begin{pmatrix} y(2) \\ y(4) \end{pmatrix}.$$

In *Mathematica* this is easily programmed:

```
downSample[x_List] :=
 Module[{i = 1, y = {}}, While[i <= Length[x],
   If[Mod[i, 2] == 0, AppendTo[y, x[[i]]]; i = ++i, i = ++i]]; y]

downSample[{2, 3, 4, 5}]

{3, 5}
```

Obviously, with downsampling information is lost. However, we can *double* the length of the input signal by using two filters, and then *halve* each output to obtain a lossless transform. The input is at one time scale and the two half-length outputs are at another scale (an octave lower).

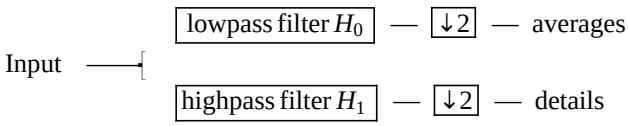


Figure 8a: Filter Bank Structure — Averages and details in the Discrete Wavelet Transform

If the input signal is of length L, the output of both H_0 and H_1 is originally of length L (provided we deal with the samples at the ends). The redundancy from $2\,L$ outputs is removed by (↓2), since after downsampling the length of each output becomes $\frac{L}{2}$.

```
(*  !for the next diagram expand to full screen !*)
```

The main idea of wavelet transforms is to *repeat* the filter bank structure. The *lowpass output* in the Filter Bank Structure above becomes the *input* to a second filter bank. The computation is cut in half because this second input is half the original length. Typically, this process is repeated four or five times.

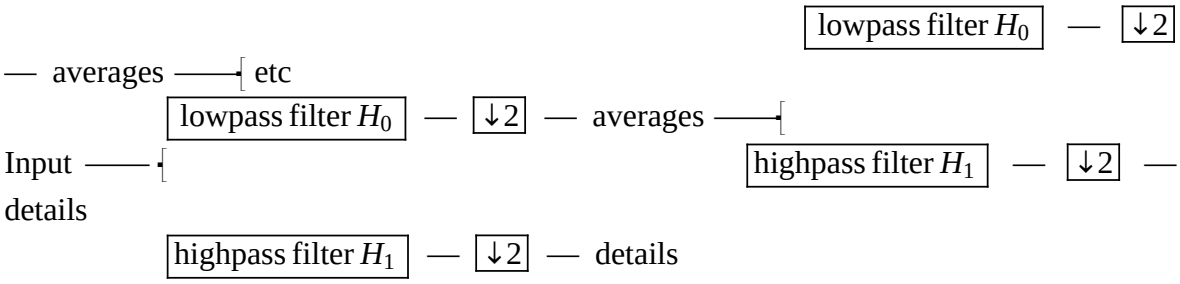


Figure 8b: Wavelet Transform Structure

Notice that the basic wavelet transform structure assumes that the downsampled coefficients of the highpass filter are small enough and not worth the additional effort to run them through another filter bank structure. They are candidates for compression.

The Haar Wavelet Transform and Its *Mathematica* Implementation

Below we choose the simplest of the wavelet transforms, named after Alfred Haar. It naturally combines the simplest examples of lowpass and highpass filters that we have previously seen (to wit, the moving average filter and the moving difference filter) into the most basic example of a filter bank.

■ Analysis

We denote the two filters by H_0 (lowpass) and H_1 (highpass). For convenience we reverse the sign of H_1 and we have:

$y_0 = H_0\, x$ is the averaging filter $\quad y_0(n) = \frac{1}{2}\, x(n-1) + \frac{1}{2}\, x(n)$

$y_1 = H_1\, x$ is the differencing filter $\quad y_1(n) = \frac{1}{2}\, x(n-1) - \frac{1}{2}\, x(n)$.

or as we have already defined in *Mathematica*:

```
yLP[n_] := 1/2 x[[n - 1]] + 1/2 x[[n]]
```

```
yHP[n_] := 1/2 x[[n - 1]] - 1/2 x[[n]]
```

Let the signal be the vector $x = \{7, 5, 6, 2\}$. To take care of the left end, we prepend 0 to the vector x.

```
Clear[x]; x = {0, 7, 5, 6, 2};
```

We next feed this vector to the "first stage" of the Wavelet Transform Structure to obtain the following outputs. From the lowpass filter we obtain the averages:

```
y0 = Table[yLP[n], {n, 2, 5}]
```

$\left\{\frac{7}{2}, 6, \frac{11}{2}, 4\right\}$

whereas from the highpass filter we obtain the differences:

```
y1 = Table[yHP[n], {n, 2, 5}]
```

$\left\{-\frac{7}{2}, 1, -\frac{1}{2}, 2\right\}$

Notice that the differences tend to be smaller than the averages. For smooth inputs this is even more true.

At this point we have eight coefficients in the vectors y_0 and y_1. They are redundant, since they came from only four samples of the vector x. We now downsample and keep the even-numbered components.

```
{y0[[2]], y0[[4]]} = downSample[y0]
```

```
{6, 4}
```

```
{y1[[2]], y1[[4]]} = downSample[y1]
```

```
{1, 2}
```

This is the first step in our analysis. We have found the four "first-level" wavelet coefficients of the signal, or we have separated the signal into wavelets. As we will see in the synthesis, the inverse transform uses these coefficients (that is, adds up wavelets times coefficients) to reconstruct the signal vector x.

The thing to note here is that, appart from downsampling, the coefficients {6, 4} can be obtained in yet another way: namely, by splitting the signal input vector into pairs $x = \{\{7, 5\},\{6, 2\}\}$ and taking the

average of each pair; i.e. $6 = \frac{7+5}{2}$ and $4 = \frac{6+2}{2}$. Similarly the coefficients {1, 2} can be obtained from these pairs by subtraction; to wit, $1 = \frac{7-5}{2}$ and $2 = \frac{6-2}{2}$. These observations are fully exploited in programming the Haar transform.

Now the downsampled output {6, 4} of the lowpasss filter becomes our new input vector *x* into the "second stage" of the Wavelet Transform Structure. Proceeding in a similar fashion we obtain the following outputs.

```
x = {0, 6, 4};

z0 = Table[yLP[n], {n, 2, 3}]

{3, 5}

z1 = Table[yHP[n], {n, 2, 3}]

{-3, 1}

Clear[x];
```

The "second-level" coefficients of the signal are

```
z0[[2]] = downSample[z0]

{5}

z1[[2]] = downSample[z1]

{1}
```

Just as in the first stage, we have $5 = \frac{6+4}{2}$ and $1 = \frac{6-4}{2}$. This completes the iteration of the Haar analysis bank, which is graphically represented below:

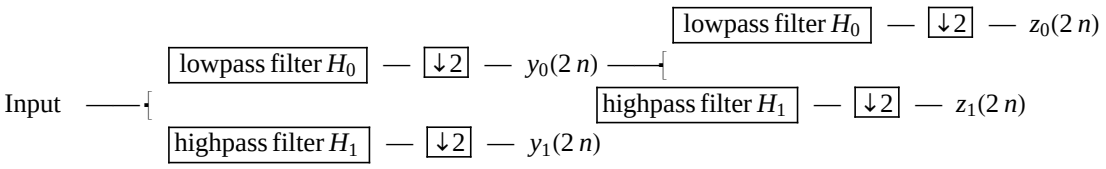


Figure 9: Haar analysis bank.

Summarizing: For the input vector $x = \{7, 5, 6, 2\}$ the wavelet coefficients are $\{5, 1, 1, 2\} = \{z_0(2\,n), z_1(2\,n), y_1(2\,n)\}$. To wit, $z_0(2\,n) = 5$ is the overall average and the rest are the details, $z_1(2\,n) = 1$, and $y_1(2\,n) = \{1, 2\}$.

■ Synthesis

Notice that in the analysis above we obtained two outputs from two successive inputs, to wit:

$$y_0(2\,n) = \tfrac{1}{2}\,x(2\,n-1) + \tfrac{1}{2}\,x(2\,n), \text{ and}$$
$$y_1(2\,n) = \tfrac{1}{2}\,x(2\,n-1) - \tfrac{1}{2}\,x(2\,n).$$

It is now easy to recover $x(2\,n-1)$ and $x(2\,n)$ by the sums and differences of the outputs:

$$x(2\,n-1) = y_0(2\,n) + y_1(2\,n), \text{ and}$$
$$x(2\,n) = y_0(2\,n) - y_1(2\,n).$$

Hence, in the synthesis the same operations are used as in the analysis. The sequence of inverse operations is *upsampling* (↑2) followed by filtering:

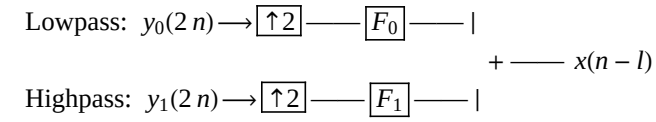


Figure 10: A stage of the inverse wavelet transform

With upsampling we double the length of a list by inserting zeros at its even positions. From the above equations we conclude that the coefficients of the lowpass filter F_0 are {1, 1} and those of the highpass filter F_1 are {1, -1}.

```
xLP[n_] := y[[n]] + y[[n - 1]]
```

```
xHP[n_] := y[[n]] - y[[n - 1]]
```

Moreover, the program to upsample is straightforward:

```
upSample[x_List] :=
 Module[{i = 1, y = x}, While[i <= 2 Length[x],
   If[Mod[i, 2] == 0, y = Insert[y, 0, i]; i = ++i, i = ++i]]; y]
```

```
upSample[{5}]
```

```
{5, 0}
```

To recover (with a unit delay) the input vector x = {7, 5, 6, 2} from the wavelet coefficients {5, 1, 1, 2} we proceed as follows: In the first stage the inputs are the coefficients {5} and {1}, which were computed last in the analysis process; we upsample them, take care of the left end of each vector, and run them through the lowpass and highpass filters F_0 and F_1 respectively. The output vectors are: *lp*, obtained from {5}, and *hp*, obtained from {1}. Then, the inputs to the second stage are: *lp*+*hp* and the last two of the wavelet coefficients {5, 1, 1, 2}. They are subjected to the same treatment.

The first stage going back is

```
Clear[y]; y = upSample[{5}]; PrependTo[y, 0];
```

```
lp = Table[xLP[n], {n, 2, 3}]
```

```
{5, 5}
```

```
Clear[y]; y = upSample[{1}]; PrependTo[y, 0];
hp = Table[xHP[n], {n, 2, 3}]
```

```
{1, -1}
```

```
lp + hp
```

```
{6, 4}
```

Here also, appart from upsampling, the coefficients {6, 4} can be recovered from the first two of the wavelet coefficients {5, 1, 1, 2} in yet another way: namely, by adding 1 to 5 and by subtracting 1 from 5. In the second stage we recover the input vector x = {7, 5, 6, 2} from {6, 4} and the last two of the wavelet coefficients {5, 1, 1, 2} by adding 1 to 6 and by subtracting 1 from 6 and by adding 2 to 4 and by subtracting 2 from 4. These observations are again fully exploited in programming the Haar transform.

For the second stage we have:

```
Clear[y]; y = upSample[{6, 4}]; PrependTo[y, 0];
lp = Table[xLP[n], {n, 2, 5}]
```

```
{6, 6, 4, 4}
```

```
Clear[y] ; y = upSample[{1, 2}] ; PrependTo[y, 0] ;
hp = Table[xHP[n], {n, 2, 5}]
```

```
{1, -1, 2, -2}
```

```
lp + hp
```

```
{7, 5, 6, 2}
```

■ *Mathematica* Implementation of the Haar Transform

Image Compression with the Haar Transform

We saw that in adoptive plotting of functions few samples are taken when there is little or no variation. The same idea will be used in image processing to take advantage of regions of little or no color variation (as in example in the color of a shirt, pants, hair etc). The Haar wavelet transform with the averaging and differencing scheme is ideally suited for this purpose and results in a method of image compression (storing much less information than would otherwise be necessary). At first though we examine some of the advantages of these transforms.

■ Wavelet Transform Benefits

Let us look at the advantages of using the Haar wavelet transform.

```
data = {74, 58, 26, 42, 66, 66, 58, 34};
discreteWaveletTransform[data]
```

```
{53., -3., 16., 10., 8., -8., 0., 12.}
```

53 is the general average and the coefficients -3, 16, 10, 8, -8, 0, 12 are the details. We have transformed a set of eight numbers into another set of eight numbers and, moreover, this transformation is reversible, as we have seen. In doing this transformation our advantage is that we can "play around" with the detail coefficients. By taking advantage of regions of little variation, we can change the set of coefficient details and use this new set (with the inverse wavelet transform) to approximate the original one. This approximation should be very close to the original.

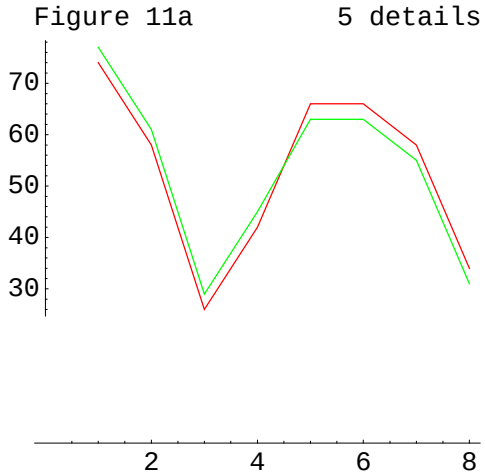
In the above example, one detail coefficient is zero indicating a region of little variation (due to the two adjacent numbers 66). The next smallest detail coefficient is -3. We change -3 to 0 and applying the inverse wavelet transform we obtain:

```
dataApproximated =
 discreteInverseWaveletTransform[{53, 0, 16, 10, 8, -8, 0, 12}]
```

```
{77, 61, 29, 45, 63, 63, 55, 31}
```

The plot of the original (red) against the approximated (green) set of numbers is seen in Figure 11a.

```
plot1 = ListPlot[data, PlotJoined → True, AxesOrigin → {0, 0},
   PlotStyle → RGBColor[1, 0, 0], DisplayFunction :→ Identity];
plot2 = ListPlot[dataApproximated, PlotJoined → True,
   AxesOrigin → {0, 0}, PlotStyle → RGBColor[0, 1, 0],
   DisplayFunction :→ Identity];
Show[plot1, plot2, DisplayFunction :→ $DisplayFunction,
  PlotLabel → "Figure 11a          5 details"];
```

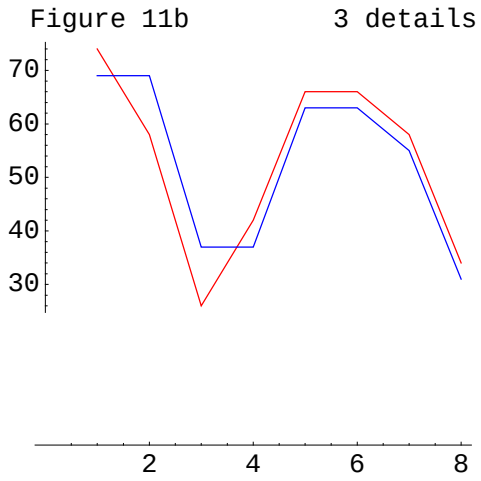


A very good approximation. Changing additionally -8 *and* 8 to zero we obtain Figure 11b in which the approximated set is in blue:

```
dataApproximatedAgain =
 discreteInverseWaveletTransform[{53, 0, 16, 10, 0, 0, 0, 12}]
```

```
{69, 69, 37, 37, 63, 63, 55, 31}
```

```
plot3 = ListPlot[dataApproximatedAgain,
   PlotJoined → True, AxesOrigin → {0, 0},
   PlotStyle → RGBColor[0, 0, 1], DisplayFunction :→ Identity];
Show[plot1, plot3, DisplayFunction :→ $DisplayFunction,
  PlotLabel → "Figure 11b          3 details"];
```



Considering that this is based only on 3 detail coefficients, the approximation is surprisingly good!

Overall, this scheme can be applied to a number string of any length. We can always pad at the end with zeros until the length of the string is a power of two. As we are going to see next, the full potential of this transformation comes when we apply it to strings of big length.

■ Plotting with Wavelet Transforms — Lossless and Lossy Compression

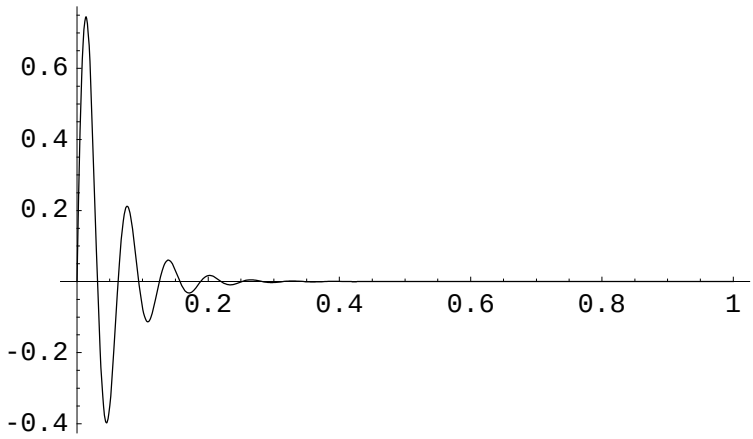
The compression process works as follows: We start with a data string of any length and using the wavelet transform we obtain another string of the same length. We can now use the inverse transform algorithm in two ways: (a) We can use all the detail coefficients to obtain the original list in which case the transform is called *lossless compression.* That is, we lost nothing. (b) We can define a *threshold* value ϵ, replace by zero all the detail coefficients with magnitude $\leq \epsilon$ and, based on the altered version of the transformed string, obtain an approximation of the original string. This is referred to as *lossy compression.* The surprising thing is that we can zero a sizable proportion of the detail coefficients and still get a decent approximation to the original string.

Let us apply the above in the function $e^{-10\,x}\,\text{Sin}[100\,x]$ which we are going to plot with uniform sampling in the interval [0, 1]. We picked this particular function because it has a great region of no variation (low activity) as can be seen from Figure 12.

```
f[x_] := E^-20 x Sin[100 x];
```

```
Plot[f[x], {x, 0, 1}, PlotRange → All, PlotLabel → "Figure 12"];
```

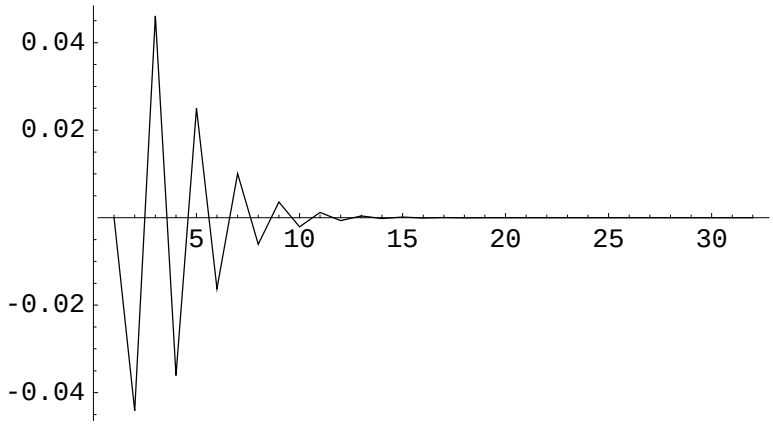
Figure 12



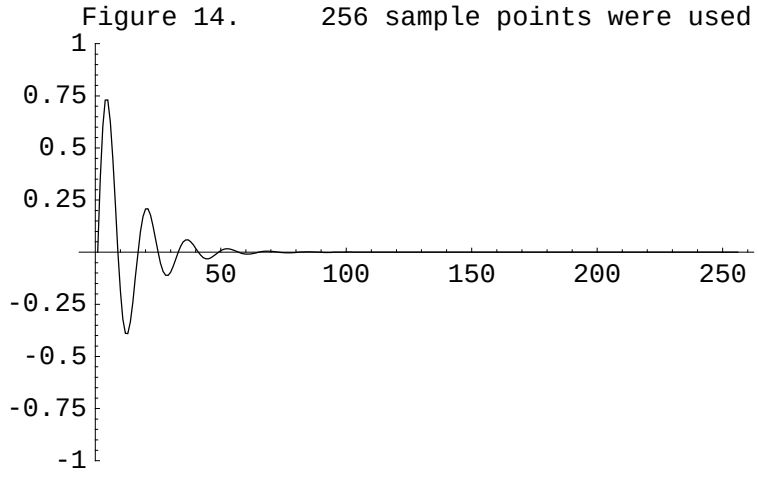
We now plot e^{-10x} Sin[100 x] using 32 and 255 uniformly sampled points in the interval [0, 1].

```
xPoints5 = Table[x, {x, 0, 1, 1 / 31}];
yPoints5 = Map[f, xPoints5];
ListPlot[yPoints5, PlotJoined → True,
  PlotRange → All, AxesOrigin → {0, 0},
  PlotLabel → "Figure 13.  32 sample points were used"];
```

Figure 13. 32 sample points were used



```
xPoints6 = Table[x, {x, 0, 1, 1 / 255}];
yPoints6 = Map[f, xPoints6];
plot15 = ListPlot[yPoints6, PlotJoined → True,
   AxesOrigin → {0, 0}, PlotRange → {-1, 1},
   PlotLabel → "Figure 14.     256 sample points were used"];
```



We see that the approximation with 256 sample points is excellent, whereas the one with 32 sample points is quite poor. In both cases though half of the points plotted were essentially wasted!

Consider the string of 256 elements, *yPoints6*, used to plot this function. Their values range from -0.39 to 0.73.

```
{Length[yPoints6], Max[yPoints6], Min[ yPoints6]} // N
```

```
{256., 0.730718, -0.39016}
```

After eight cycles of averaging and differencing we get a transformed string of the same length, *yPoints6WT*, with values ranging from -0.245 to 0.355.

```
yPoints6WT = discreteWaveletTransform[yPoints6];
{Length[yPoints6WT], Max[yPoints6WT], Min[ yPoints6WT]}
```

```
{256, 0.355324, -0.245031}
```

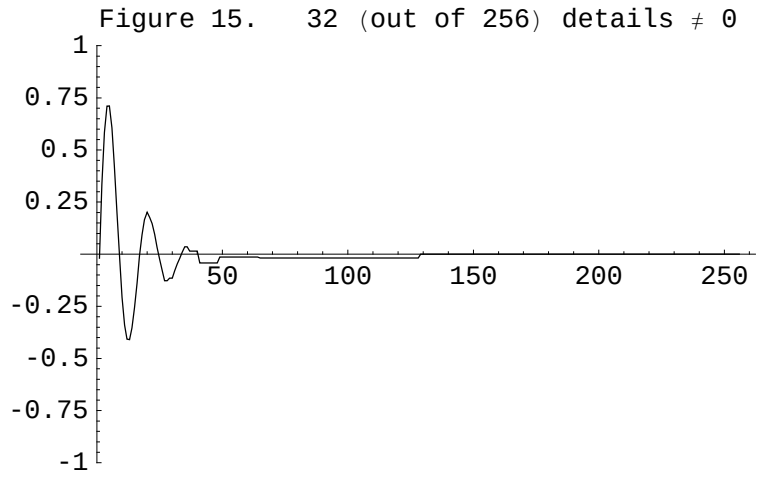
We next pick a threshold value of $\epsilon = 0.014$ and zero all values of *yPoints6WT* (the detail coefficients) $\leq \epsilon$. This way we get an altered transformed list, *yPoints6WTSparse*, with 32 nonzero entries.

```
yPoints6WTSparse =
  Fold[If[Abs[#2] <= 0.014, Append[#1, 0], Append[#1, #2]] &,
   {}, yPoints6WT];
Select[yPoints6WTSparse, # != 0 &] // Length
```

```
32
```

From this sparse list, *yPoints6WTSparse*, we obtain an approximation, *yPoints6WTApprox*, of the original list and we plot it in Figure 15.

```
yPoints6WTApprox =
  discreteInverseWaveletTransform[yPoints6WTSparse];
ListPlot[yPoints6WTApprox, PlotJoined → True,
  AxesOrigin → {0, 0}, PlotRange → {-1, 1},
  PlotLabel → "Figure 15.   32 (out of 256) details ≠ 0"];
```

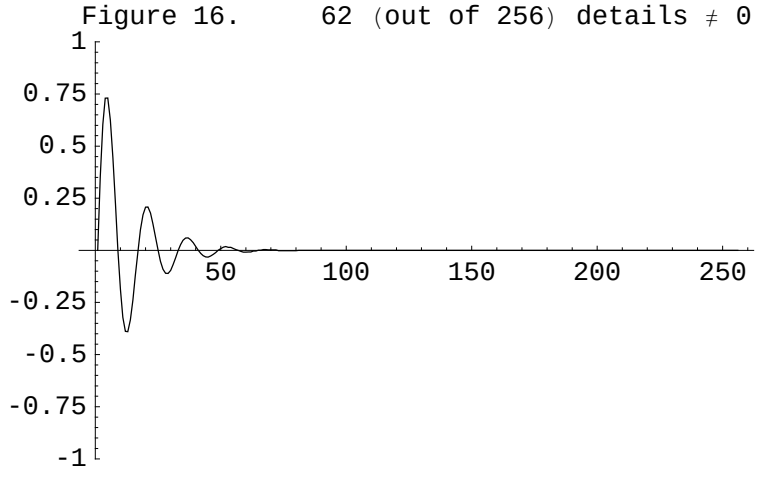


Despite its limitations this does a better job of approximating the actual graph of $e^{-10x}\,\mathrm{Sin}[100\,x]$, than the plot with 32 uniformly sampled points in Figure 13. Finally, Figure 16 shows the even better approximation of $E^{-10x}\,\mathrm{Sin}[100\,x]$ obtained by using the threshold value of $\epsilon = 0.0014$, in which case the altered transformed list, *yPoints6WTSparse*, has 62 nonzero entries.

```
yPoints6WTSparse =
  Fold[If[Abs[#2] <= 0.0014, Append[#1, 0], Append[#1, #2]] &,
   {}, yPoints6WT];
Select[yPoints6WTSparse, # != 0 &] // Length
```

```
62
```

```
yPoints6WTApprox =
  discreteInverseWaveletTransform[yPoints6WTSparse];
ListPlot[yPoints6WTApprox, PlotJoined → True,
  AxesOrigin → {0, 0}, PlotRange → {-1, 1},
  PlotLabel → "Figure 16.    62 (out of 256) details ≠ 0"];
```



It should be noted that the threshold values are very low compared to the range of values. Yet, despite this fact, we obtain good results using few detail coefficients. The reason for this behavior is that there is no variation in more than half of the interval of interest. Using 32 detail coefficients, out of 256, gives a compression ration of 8:1, whereas using 62 gives a compression ratio of 4.1:1.

Also, note that Figures 15 and 16 were generated using 32 and 62 nonzero detail coefficients respectively, in sparse strings of 256 elements; however, the plots themselves used all 256 mostly nonzero numbers obtained from those strings by applying the *inverse* Haar wavelet transform.

Lossy compression comes into play once we realize that it takes less space to store *sparse* strings of length 256 — with 32 or 62 nonzero entries — than arbitrary strings of length 256. Using *Mathematica*'s special function SparseArray[], we see that storing the 256-element list *yPoints6WT* requires about 5.5K bytyes of memory:

```
(*   Memory used by a non-sparse string of length 256   *)

a = MemoryInUse[]; yPoints6WT = discreteWaveletTransform[yPoints6];
b = MemoryInUse[]; b - a
```

```
5472
```

whereas, for a sparse array about half that memory is needed for storing. Note that, to be able to use the SparseArray[] function of *Mathematica*, we first take the list *yPoints6WTSparse*, generated above with 32 or 62 nonzero entries, compute the positions where these nonzero entries are and turn the position lists into rules of the form "position → value". The list of these rules, *positionAndValueRules*, is the input to the function SparseArray[].

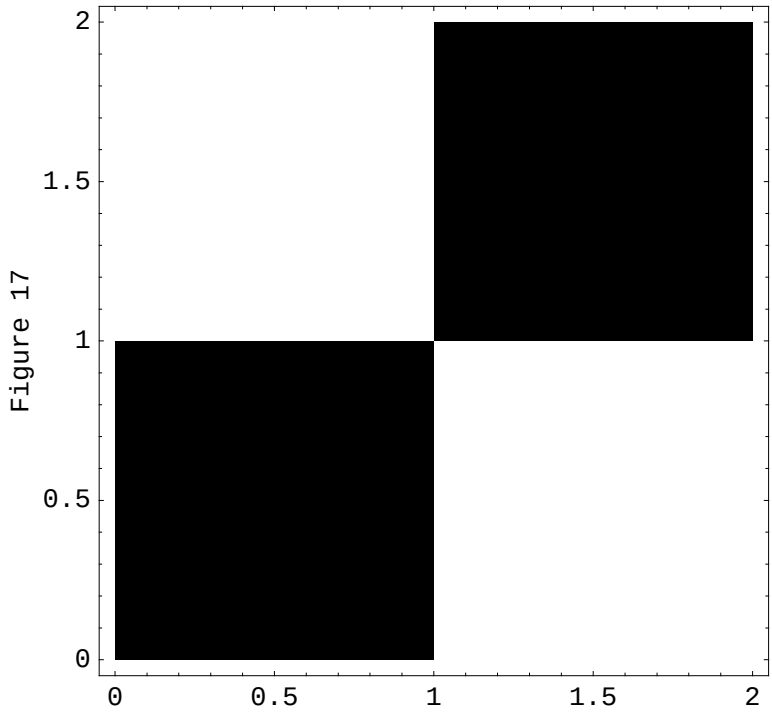
```
yPoints6WTSparse;
res = {};
Map[If[# ≠ 0, AppendTo[res, {Position[yPoints6WTSparse, #], #}]] &,
 yPoints6WTSparse];
positions = Map[{Flatten[First[#]], Last[#]} &, res];
positionAndValueRules = Map[Apply[Rule, #] &, positions];


(*   Memory used by a sparse string of length 256   *)

a = MemoryInUse[];
SparseArray[positionAndValueRules];
b = MemoryInUse[]; b - a
```

```
2384
```

■ Primitive Image Compression

Suppose we have the image shown in Figure 17. Please note that 0 corresponds to black and 1 to white; moreover, the rows in the Raster function are printed in reverse order.

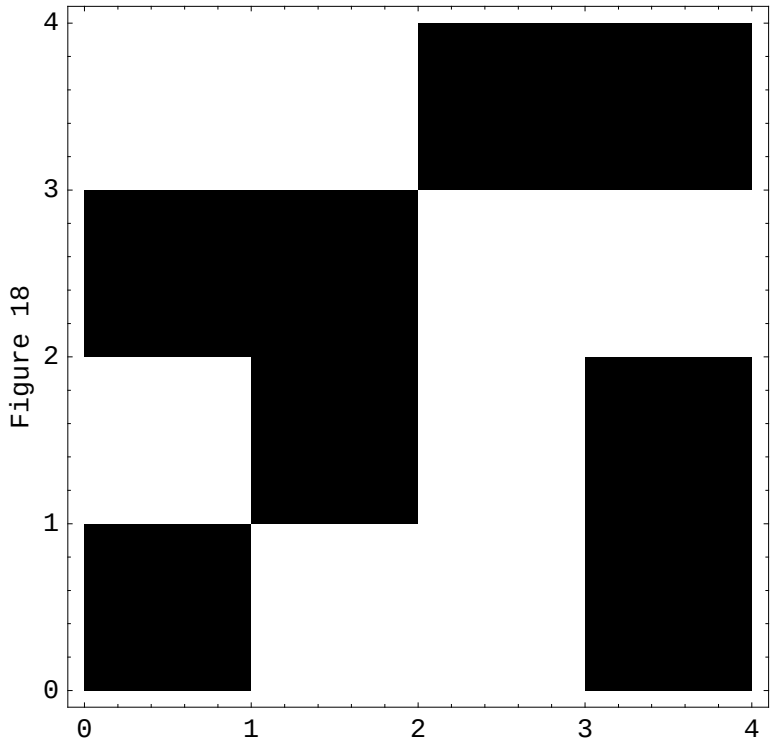
```
Show[Graphics[Raster[{{0, 1}, {1, 0}}],
   Frame → True, FrameLabel → "Figure 17", AspectRatio → 1]];
```



The above image can be considered as a $2^i \times 2^i$ pixel image which has 4 pixels if $i = 1$. What information is needed to store it? If we assume that we have black unless specified otherwise, then we need only say where there is white. In the case $i = 1$, we obviously need 2 pieces of information, namely that pixels {1, 1} and {2, 2} are white. We will show the quite interesting fact that, for *any* value of $i > 1$ we need *only* 2 pieces of information.

Next consider the more complex image shown in Figure 18.

```
Show[
  Graphics[Raster[{{0, 1, 1, 0}, {1, 0, 1, 0}, {0, 0, 1, 1}, {1, 1, 0, 0}}],
   Frame → True, FrameLabel → "Figure 18", AspectRatio → 1]];
```



Again, we can treat it as $4^i \times 4^i$ pixel image which has 16 pixels if $i = 1$. Arguing as above, in the case $i = 1$ we need 8 pieces of information. However, we can do better if we use the fact that several of the white blocks are adjacent to each other. We will show that for any value of i (i.e. no matter what the resolution) we only need 5 pieces of information.

We assume that both images of Figures 17 and 18 have resolution 256×256; that is, each image is represented by a 256×256 matrix with entries 0 or 1. Using the technique of lossy compression we will show that to store the first image we need 2 pieces of information whereas for the second we need 5. In each case we apply the Haar transform on each row separately, and then do the same on each column of the resulting matrix. The final result is a new 256×256 matrix with an overall average at the top left hand corner and a lot of detail coefficients. Regions of little variation in the original image manifest themselves as numerous small or zero elements in the transformed matrix.

We start with the image of Figure 17. Each one of the first 128 rows is made up of 128 ones (1→white) followed by 128 zeros (0→black).

```
matrixRowF = Join[Table[1, {i, 1, 128}], Table[0, {i, 1, 128}]];
```

When we apply the Haar wavelet transform to each one of these 128 rows the result is a row with all, but the first two elements, zero.

```
discreteWaveletTransform[matrixRowF];
```

Hence after the transformation the *first* 128 rows of the matrix are identical; in each row, the first two elements are 0.5, 0.5 and the rest 254 are 0.

Likewise, after the transformation, the *last* 128 rows of the matrix are identical; in each row, the first two elements are 0.5, -0.5 and the rest 254 are 0.

```
matrixRowL = Join[Table[0, {i, 1, 128}], Table[1, {i, 1, 128}]];
discreteWaveletTransform[matrixRowL];
```

At this point the columns of the 256×256 matrix are all zero except for the first two. The first column has 256 identical entries of 0.5 whereas the second column consists of 128 rows of 0.5 followed by 128 rows of -0.5. Transforming the first column we obtain another one with all entries zero, except for the *first* entry which is 0.5; likewise, from the second column we obtain another one with all entries zero, except for the *second* entry which is 0.5.

```
matrixColumn1 = Table[0.5, {i, 1, 256}];
discreteWaveletTransform[matrixColumn1];

matrixColumn2 =
  Join[Table[0.5, {i, 1, 128}], Table[-0.5, {i, 1, 128}]];
discreteWaveletTransform[matrixColumn2];
```

Since our wavelet transform is invertible the image of Figure 17 can be stored in a 2×2 matrix form with only two nonzero pieces of information

$$\begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}.$$

The image compression process can be summarized as follows: (a) form the original matrix, (b) transform all the rows to obtain the transformed matrix, (c) transform the columns of the transformed matrix and (d) print the first few rows and columns. (To do (c) in *Mathematica* we have to transpose the transformed matrix so that the columns become rows, apply the wavelet transform on the rows and transpose it again to obtain the final matrix.)

```
originalMatrix = Join[
   Table[matrixRowF, {i, 1, 128}], Table[matrixRowL, {i, 1, 128}]];

transformedMatrix = Map[discreteWaveletTransform, originalMatrix];

transformedTransposedMatrix = Transpose[transformedMatrix];

finalMatrix = Map[discreteWaveletTransform,
    transformedTransposedMatrix] // Transpose;

Table[finalMatrix[[i, j]], {i, 1, 7}, {j, 1, 7}] // MatrixForm
```

$$\begin{pmatrix} 0.5 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0.5 & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0 & 0. & 0. & 0 \\ 0. & 0. & 0 & 0. & 0 & 0 & 0. \\ 0. & 0. & 0. & 0 & 0. & 0. & 0 \\ 0. & 0. & 0. & 0 & 0. & 0. & 0 \\ 0. & 0. & 0 & 0. & 0 & 0 & 0. \end{pmatrix}$$

Applying the same process to the image of Figure 18 we see that it can be stored in a 4×4 matrix using 5 nonzero pieces of information

$$\begin{pmatrix} 0.5 & 0 & 0 & 0.25 \\ 0 & 0 & 0 & -0.25 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \end{pmatrix}$$

- **Compression of the Image of Figure 18**

■ Compression of arbitrary images (Thanks to my brother M. G. Akritas for suggesting this section)

To process an arbitrary digital black-and-white image we have first to get a digital version of it. For this purpose, we first read off as a string the contents of the file containing the encapsulated postscript form of a picture (in our case “pedro.eps”)

```
string1 = ReadList[
   ToFileName[{"AddOns", "Applications"}, "pedro.eps"], String];
```

and get rid off of the header and trailer information of the file. Notice the length of string2 below, the string with only image information, whicn in our case is 314! This tells us the number of rows of the picture.

```
(*  For some versions of Mma we have to use *)

(*   string2=Drop[Drop[string1,17],-2];   *)

(*  string2=Drop[Drop[string1,18],-3];  *)

string2 = Drop[Drop[string1, 18], -3];

Length[string2]

314
```

Convert the string into a character list, and then partition it in pairs

```
chars = Flatten[Characters[ string2]];

Length[chars]

102992


chars2 = Partition[chars, 2];
```

Turn each character pair, say {7, A}, into the string form "16^^7A" and that into an expression; this gives us list1, a list of numbers in the range [0, 255]—since 16^^FF (the maximum) is 255. The length of the partitioned list is half the length of the original character list.

```
string3 = Map[StringJoin[Join[{"16^^"}, #]] &, chars2];

list1 = Map[ToExpression, string3];

Length[list1]

51496
```

We next need to partition this list of numbers (list1) in such a way so that we obtain a matrix with 314 rows (the length of string2 above). Half the length of the original character list should be greater than 65536 (= 256×256) if we want to have a 256×256 matrix. In our case $\frac{\text{Length[chars]}}{2} = \frac{102992}{2} = 51496$ is slightly less than 65536. (This could be rectified by increasing the resolution of the picture, but we will do something else below.)

As seen in the expression list2 = Partition[list1, 164] below, the partition number in our case is 164. The partition number is the number of columns of the matrix and *must* be a divisor of the length of the character list. We repeat: the partition number is chosen so that the resulting number of rows is *equal* to the length of string2, the string with only image information (314 in our example)! This is achieved by trial and error:

```
FactorInteger[51496]
```

```
{{2, 3}, {41, 1}, {157, 1}}
```

```
list2 = Partition[list1, 164];
```

```
Dimensions[list2]
```

```
{314, 164}
```

In this case we need to "massage" the resulting 314×164 matrix into a square one to be wavelet transformed. Hence, we remove the excess 58 (= 314 - 256) rows and add 92 (= 256 - 164) columns. The rows can be removed from the top or from the bottom — chosen here — or as the picture dictates; since each row has only 164 “columns” we pad them with “white columns” accordingly — 46 on the left and 46 on the right.

```
whiteColumn = Table[255, {256}];
```

```
(data1 = Transpose[
   Join[Table[whiteColumn, {46}], #, Table[whiteColumn, {46}]] &[
    Transpose[Drop[list2, -58]]]];
 Dimensions[data1])
```

```
{256, 256}
```

We finally have the picture of *Pedro*!

```
ListDensityPlot[data1, Mesh → False];
```

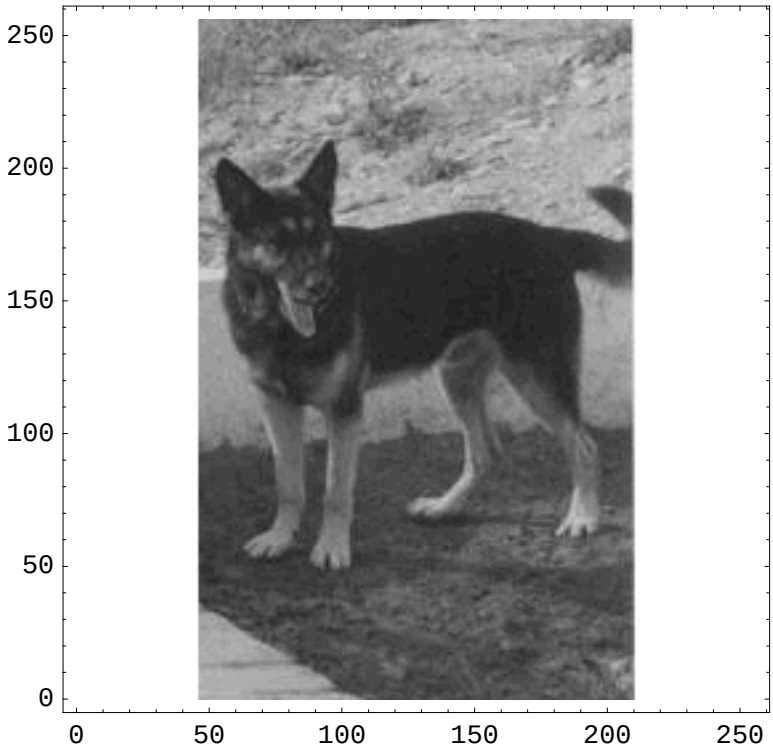


Figure *Pedro-1*: The original picture of Pedro

The "padded" columns on the left and right of Pedro can be hardly missed!

We add parenthetically that the digital information about Pedro — that is, the matrix data1 — can be stored in the seperate file pedro.data using the function Write

```
Write[ToFileName[{"AddOns", "Applications"}, "pedro.data"], data1];
```

from which it can be read into the variable pedroDigitalData with the function ReadList

```
pedroDigitalData =
  ReadList[ToFileName[{"AddOns", "Applications"}, "pedro.data"],
    Expression] // Last;
```

Once we have the picture in digital form, all the fun begins. We run it through the wavelet transform to compress it; rows are transformed first and then columns as was done for the case of the primitive images.

```
originalMatrix = data1;
transformedMatrix =
  Map[discreteWaveletTransform, originalMatrix];
transformedTransposedMatrix = Transpose[transformedMatrix];
finalMatrix = Map[discreteWaveletTransform,
    transformedTransposedMatrix] // Transpose;
```

Compression can be achieved by zeroing (for example) the detail coefficients which are smaller than 3 in absolute value. This way we obtain a matrix with only 6629 nonzero entries (about 10% of the entries)!

```
a = MemoryInUse[];
```

```
finalMatrixSparse = Map[
   Fold[If[Abs[#2] ≤ 3.0, Append[#1, 0], Append[#1, #2]] &, {}, #] &,
   finalMatrix];
Select[Flatten[finalMatrixSparse], # != 0 &] // Length
```

```
6629
```

```
b = MemoryInUse[]; b - a
```

```
275408
```

Stored in the usual way — with the zero entries included — it takes about 270K bytes of memory. However, *Mathematica* 5 allows us to store this matrix as a sparse array, and doing so it takes about 80K bytes of memory. The improvement in memory storage is quite impressive.

```
sparseFinalMatrixSparse = SparseArray[finalMatrixSparse]
```

```
SparseArray[<6629>, {256, 256}]
```

```
c = MemoryInUse[]; c - b
```

```
83664
```

Compression achieved!

We then reverse the process and obtain an *approximation* to the original matrix! Notice that the sparse array has to be converted to a "normal" matrix first.

```
backUpOneStep = Map[discreteInverseWaveletTransform,
   Transpose[sparseFinalMatrixSparse // Normal]];
backUpOneStepTransposed = Transpose[backUpOneStep];
approximatedMatrix =
  Map[discreteInverseWaveletTransform, backUpOneStepTransposed];
```

Note that the approximated matrix has 65536 (= 256×256) nonzero entries, just like the original matrix.

```
Select[Flatten[approximatedMatrix], # != 0 &] // Length
```

```
65536
```

When we plot the approximated matrix, and consider the fact that we used only 10% of the entries in the original matrix, the result is not at all bad!

```
ListDensityPlot[approximatedMatrix, Mesh → False];
```

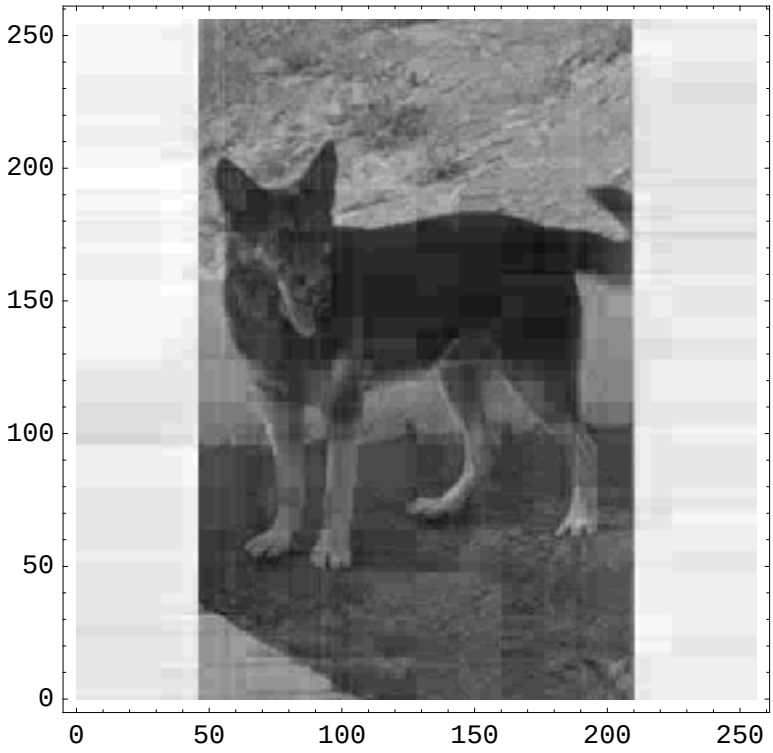


Figure *Pedro-2*: Picture of Pedro obtained using only 10% of the details.

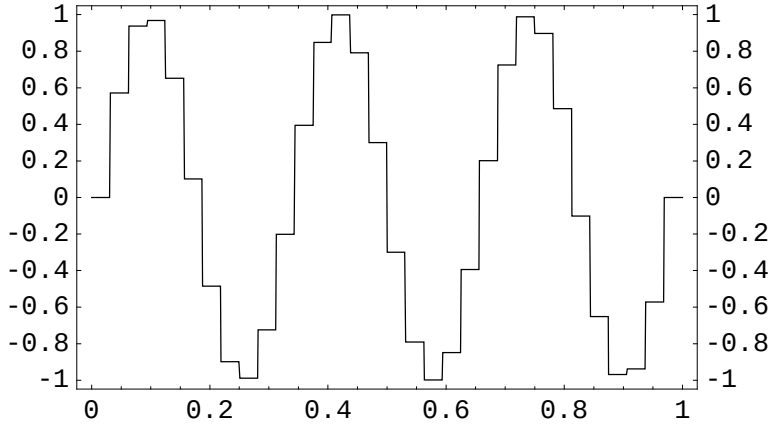
Wavelets and Multiresolution Analysis

In the spirit of Shannon (at least two samples per cycle) we are going over wavelets a second time and introduce the general concepts and notations used in their study.

■ Scaling Functions

We start by associating a list of k elements with a step function on [0, 1]; this step function could change at $k-1$ equally spaced points and uses the list elements as its step heights. As an example consider Figure 7 (shown below again) where the step function plotted is associated with the list of the 32 $\sin(6\pi x)$-values obtained by uniformly subdividing the interval [0, 1].

Figure 7



Step function version of the function Sin[6 πx] in the interval [0, 1]; 32 sample points were used

In turn, these step functions can be thought of as linear combinations of dyadically dilated and translated unit step functions on [0, 1). To explain the last statement we define the *Haar scaling function*:

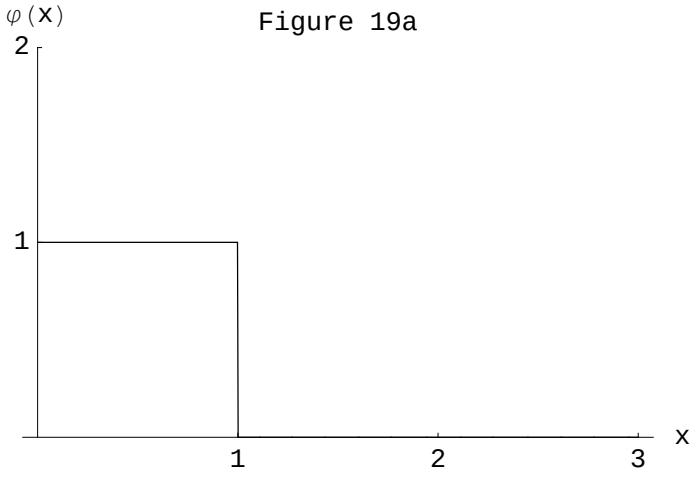
$$\varphi(x) = \begin{cases} 1 & \text{for } 0 \le x < 1 \\ 0 & \text{otherwise} \end{cases}$$

```
Clear[x];
φ[x_] := Which[0 ≤ x < 1, 1, True, 0];
```

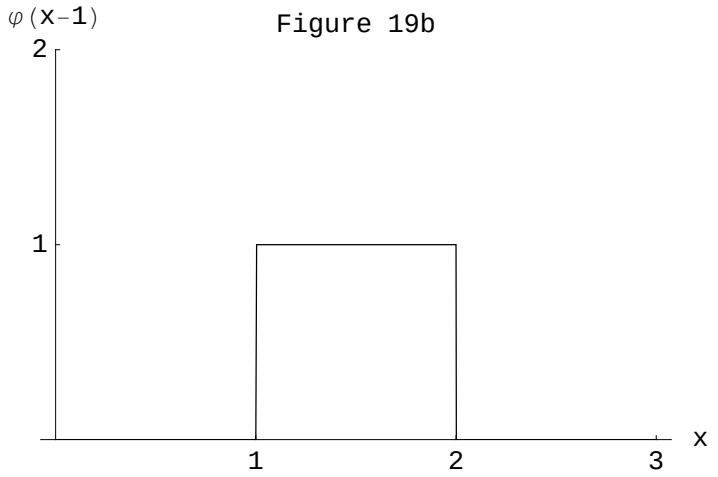
and look at two of its plots for better understanding.

```
Plot[φ[x], {x, 0, 3}, PlotRange → {0, 2}, Ticks → {{1, 2, 3}, {1, 2}},
  PlotLabel → "Figure 19a", AxesLabel → {"x", "φ(x)"}];
```

Figure 19a

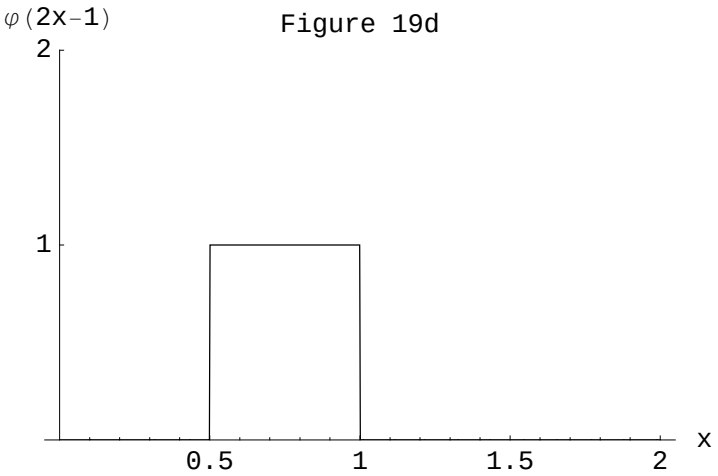


```
Plot[φ[x - 1], {x, 0, 3}, PlotRange → {0, 2}, Ticks → {{1, 2, 3}, {1, 2}},
  PlotLabel → "Figure 19b", AxesLabel → {"x", "φ(x-1)"}];
```



φ satisfies a scaling equation of the form $\varphi(x) = \sum_{i=0}^{\infty} c_i \varphi(2x - i)$, where in our case the only nonzero coefficients are $c_0 = c_1 = 1$; that is, we have $\varphi(x) = \varphi(2x) + \varphi(2x - 1)$. Note that the equations $h(i) = \frac{c_i}{2}$, $i = 0, 1$ define the filter coefficients.

```
Plot[φ[2 x - 1], {x, 0, 2},
  PlotRange → {0, 2}, Ticks → {Automatic, {1, 2}},
  PlotLabel → "Figure 19d", AxesLabel → {"x", "φ(2x-1)"}];
```



We now define, for each nonnegative integer i, the *approximating* vector space $\mathcal{V}^i$ of piecewise constant functions on [0, 1), with possibble breaks at $\frac{1}{2^i}$, $\frac{2}{2^i}$, ..., $\frac{2^i-1}{2^i}$. Then the 2^i dyadically dilated and translated scaling functions $\varphi_k^i(x) = \varphi(2^i x - k)$, $0 \le k \le 2^i$-1, form a basis for $\mathcal{V}^i$. We thus have the (theoretically infinite) chain of ascending vector spaces $\mathcal{V}^0 \subset \mathcal{V}^1 \subset \mathcal{V}^2 \subset \mathcal{V}^3 \subset \ldots$; however, since we are dealing with sampled signals, the resolution is finite and hence this chain stops at some finite value i.

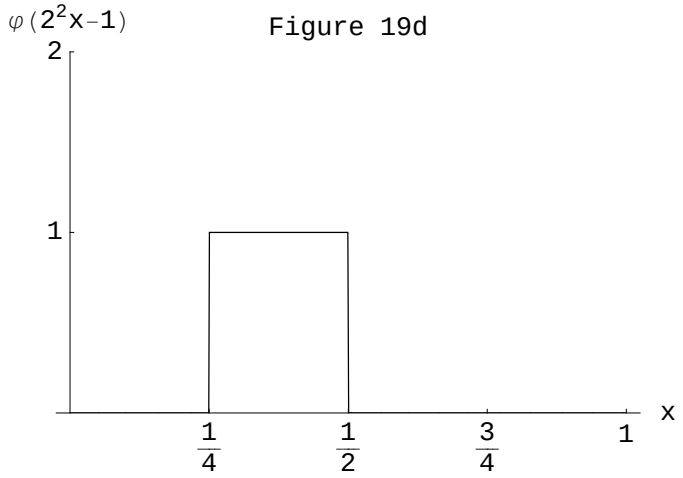
```
φ[x_, i_, k_] := φ[2^i x - k];
```

Taking for example $i = 2$, we have the set of 4 functions $\varphi(2^2 x - k)$, $k = 0, 1, 2, 3$.

Note that the function $\varphi(2^2 x)$ is 1 on the interval $\left[0, \frac{1}{4}\right)$,

the function $\varphi(2^2 x - 1)$ is 1 on the interval $\left[\frac{1}{4}, \frac{2}{4}\right)$ etc.

```
Plot[φ[2^2 x - 1], {x, 0, 1}, PlotRange → {0, 2},
  Ticks → {{0, 1 / 4, 2 / 4, 3 / 4, 1}, {1, 2}},
  PlotLabel → "Figure 19d", AxesLabel → {"x", "φ(2^2x-1)"}];
```



These four functions form a basis for the *approximating* vector space $\mathcal{V}^2$ of piecewise constant functions on [0, 1), with possibble breaks at 1/4, 2/4, 3/4. Taking a random list of four numbers $\{e_0, e_1, e_2, e_3\}$ we can think of them as the element

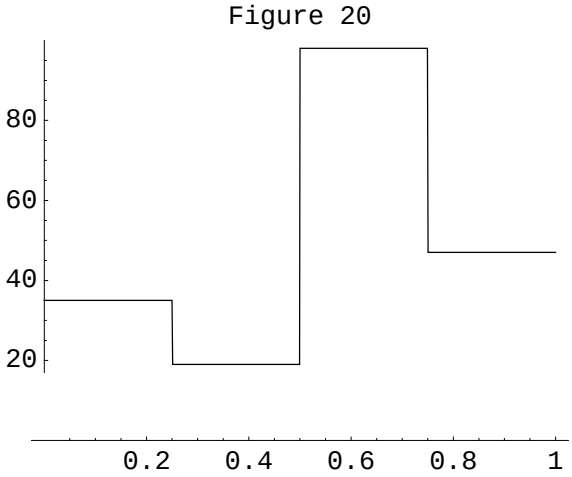
$$e_0\,\varphi[2^3 x] + e_1\,\varphi[2^3 x - 1] + e_2\,\varphi[2^3 x - 2] + e_3\,\varphi[2^3 x - 3]$$

of the space $\mathcal{V}^2$. Given a specific list of length 4, its plot is shown in Figure 20:

```
data = Table[Random[Integer, {10, 99}], {4}]
```

```
{35, 19, 98, 47}
```

```
yStep = Table[φ[x, 2, k], {k, 0, 3}];
Plot[data.yStep, {x, 0, 1},
  PlotLabel → "Figure 20", AxesOrigin → {0, 0}];
```



In other words, we have a step function representation of our data list. Similarly, any list of length 4 can be associated with an element of the approximating space $\mathcal{V}^2$.

We can explain the Haar wavelet transform (lowpass/highpass filters) in terms of this version of data lists, but we need additional (vector) approximating spaces. As above, the two functions defined by $\varphi_k^1(x) = \varphi(2^1 x - k)$, $k = 0, 1$ form a basis of the approximating space $\mathcal{V}^1$ of the piecewise constant functions on [0, 1) with possible break at the point 1/2. Finally, the function $\varphi_0^0(x) = \varphi(x)$ itself is a basis of the approximating space $\mathcal{V}^0$ of the constant functions on [0, 1). We have $\mathcal{V}^0 \subset \mathcal{V}^1 \subset \mathcal{V}^2$

If we think of the various averages — that we computed in the Haar wavelet transform — as lower-resolution versions of the original data list, we can associate them with elements of these new vector spaces. The higher the dimension of the vector space $\mathcal{V}^i$ the higher the resolution. Let us come back to the example we did earlier, where now we print the intermediate results:

```
discreteWaveletTransform[{7, 5, 6, 2}, 1]
```

```
{7., 5., 6., 2.}

{6., 4., 1., 2.}

{5., 1., 1., 2.}
```

The first output line (the original data list) is associated with the element $7\varphi(2^2 x - 0) + 5\varphi(2^2 x - 1) + 6\varphi(2^2 x - 2) + 2\varphi(2^2 x - 3)$ of the approximating space $\mathcal{V}^2$ and has the highest resolution. The average values 6 and 4 of the second output line are associated with the element $6\varphi(2^1 x - 0) + 4\varphi(2^1 x - 1)$ of the approximating space $\mathcal{V}^1$ (the resolution is cut in half) and the overall average value 5 is associated with the element $5\varphi(2^0 x)$ of the approximating space $\mathcal{V}^0$ (the resolution is cut *yet again* in half). ("Going down"

the approximating spaces the resolution goes down as well. This is the dyadical dilation we mentioned above.)

- **Wavelets — at last!**

It only remains to associate the detail coefficients with "something". This is where wavelets come in as the *details* or *wavelet* space!

For any nonnegative integer i, consider the inner product

$$\langle \mathrm{f}\,,\, \mathrm{g} \rangle = \int_0^1 f[t]\, g[t]\, dt$$

defined in the vector space $\mathcal{V}^i$. Two functions are orthogonal if and only if their product on [0, 1] encloses equal areas above and below the horizontal axis. In $\mathcal{V}^{i+1}$ we define the wavelet space $\mathcal{W}^i$ as the orthogonal complement of the approximating space $\mathcal{V}^i$. Hence, we have the following (orthogonal) direct sum decomposition:

$$\begin{aligned} \mathcal{V}^{i+1} &= \mathcal{V}^i \oplus \mathcal{W}^i = \mathcal{V}^{i-1} \oplus \mathcal{W}^{i-1} \oplus \mathcal{W}^i \\ &\vdots \\ &= \mathcal{V}^0 \oplus \mathcal{W}^0 \oplus \mathcal{W}^1 \oplus \ldots \oplus \mathcal{W}^i . \end{aligned}$$

Note that at every resolution (approximating space) there exist details corresponding to *all* the previous resolutions.

Every wavelet space $\mathcal{W}^i$ has a basis $\chi_k^i(x) = \chi(2^i x - k)$, $k = 0, 1, \ldots, 2^i - 1$, which is used to associate the details, obtained during the Haar transform, with elements of the wavelet space; the details are called *wavelet coefficients*.

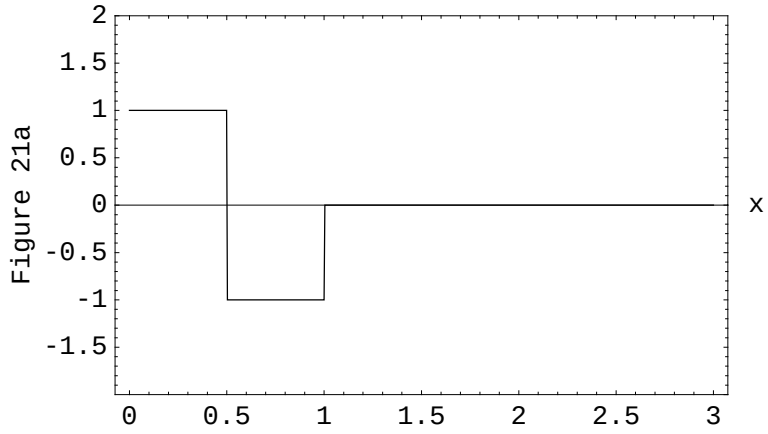
Equivalent to the Haar scaling function is the *mother Haar wavelet*, defined by:

$$\chi(x) = \begin{cases} 1 & \text{for } 0 \le x < \frac{1}{2} \\ -1 & \text{for } \frac{1}{2} \le x < 1 \\ 0 & \text{otherwise} \end{cases}$$

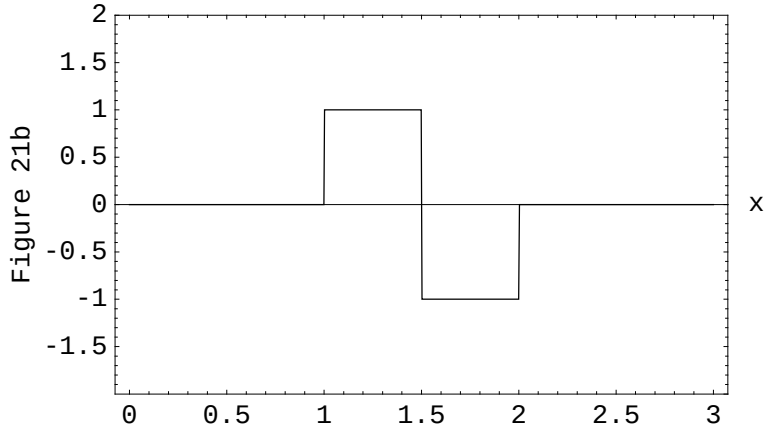
```
Clear[x];
χ[x_] := Which[0 ≤ x < 1 / 2, 1, 1 / 2 ≤ x < 1, -1, True, 0];
```

along with two plots for better understanding.

```
Plot[χ[x], {x, 0, 3}, PlotRange → {-2, 2}, Ticks → {{0, 1, 2, 3}, {1, 2}},
  Frame → True, FrameLabel → "Figure 21a", AxesLabel → {"x", "χ(x)"}];
```




```
Plot[χ[x - 1], {x, 0, 3}, PlotRange → {-2, 2},
  Ticks → {{0, 1, 2, 3}, {1, 2}}, Frame → True,
  FrameLabel → "Figure 21b", AxesLabel → {"x", "χ(x)"}];
```



The function $\chi(x)$ is orthogonal to $\varphi(x)$ and forms a basis for $\mathcal{W}^0$. Likewise, the functions

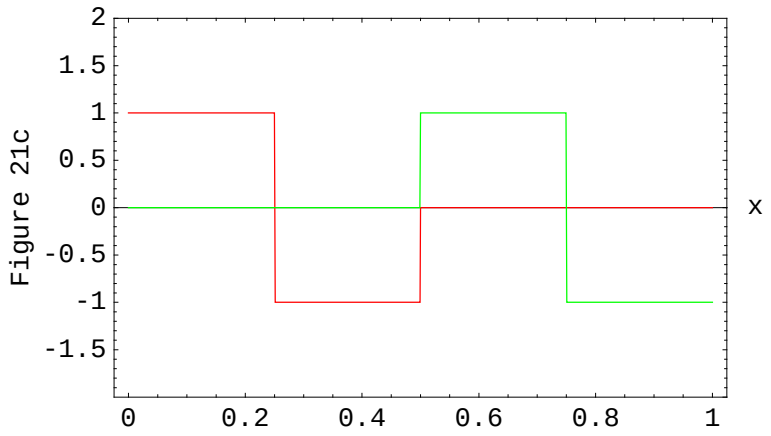
$$\chi_k^i(x) = \chi(2^i x - k) \text{ , } k = 0, 1, ..., 2^i - 1$$

are visibly orthogonal to each other and to the corresponding functions $\varphi_k^i(x)$; hence, they form a basis for $\mathcal{W}^i$.

```
χ[x_, i_, k_] := χ[2^i x - k];
```

Moreover, for $i = 1$ we have the 2 functions $\chi_k^1(x) = \chi(2^1 x - k)$, $k = 0, 1$ which are visibly orthogonal to each other and to the corresponding functions $\varphi_k^1(x)$ which form a basis of the resolution space $\mathcal{V}^1$. Hence they form a basis of $\mathcal{W}^1$.

```
Plot[{χ[2 x - 0], χ[2 x - 1]}, {x, 0, 1},
  PlotRange → {-2, 2}, Ticks → {{0, 1 / 4, 2 / 4, 3 / 4, 1}, {1, 2}},
  PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 1, 0]}, Frame → True,
  FrameLabel → "Figure 21c", AxesLabel → {"x", "χ(2x)"}];
```



Taking another look at our example

```
discreteWaveletTransform[{7, 5, 6, 2}, 1]
```

```
{7., 5., 6., 2.}
```

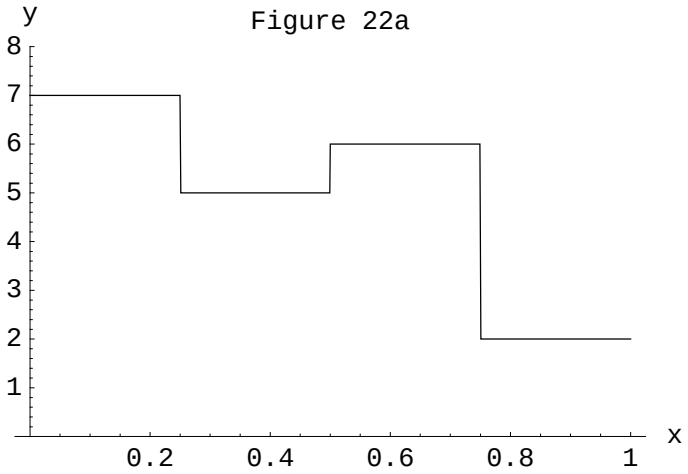
```
{6., 4., 1., 2.}
```

```
{5., 1., 1., 2.}
```

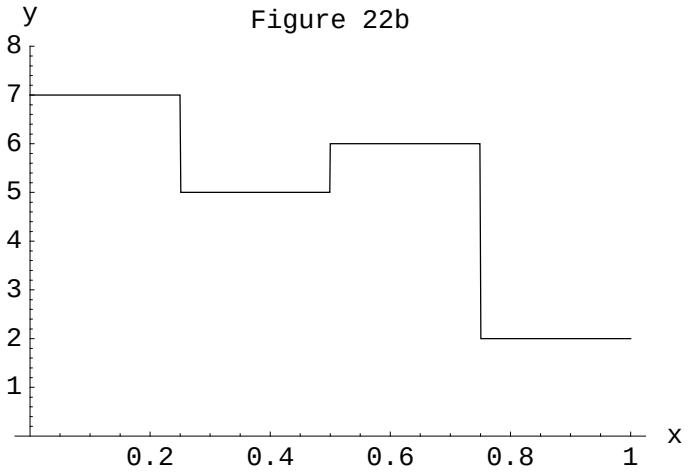
we see that each output line is associated with the corresponding line of the following relations:

$$
\begin{aligned}
&7\varphi(2^2\,x-0)+5\varphi(2^2\,x-1)+6\varphi(2^2\,x-2)+2\varphi(2^2\,x-3)\\
&=6\varphi[2^1\,x-0]+4\varphi(2^1\,x-1)+1\chi(2^1\,x-0)+2\chi(2^1\,x-1)\\
&=5\varphi(2^0\,x)+1\chi(2^0\,x)+1\chi(2^1\,x-0)+2\chi(2^1\,x-1).
\end{aligned}
$$

```
Plot[7 φ[2^2 x - 0] + 5 φ[2^2 x - 1] + 6 φ[2^2 x - 2] + 2 φ[2^2 x - 3] ,
 {x, 0, 1}, PlotRange → {0, 8},
 PlotLabel → "Figure 22a", AxesLabel → {"x", "y"}];
```




```
Plot[5 φ[2^0 x] + 1 χ[2^0 x] + 1 χ[2^1 x - 0] + 2 χ[2^1 x - 1],
 {x, 0, 1}, PlotRange → {0, 8},
 PlotLabel → "Figure 22b", AxesLabel → {"x", "y"}];
```



The images that we compressed in the previous section had a resolution of 256×256. Thus, working with lists of length 256 was equivalent to working in the larger space $\mathcal{V}^8$ and using the identity

$$\mathcal{V}^0 \oplus \mathcal{W}^0 \oplus \mathcal{W}^1 \oplus \ldots \oplus \mathcal{W}^7$$

one overall average and 255 details.

There are two-dimensional analogs of those ideas providing a theoretical framework for the digital image representation and compression ideas discussed in the last section. Details can be found elsewhere.

■ **Wavelet Transform Benefits — Revisited**

As we have seen in an example in the previous section, we set the smallest detail coefficients equal to zero and achived a pretty good approximation of the original list (of eight elements). This amounts to setting some of the wavelet coefficients to zero. Let us look again at the same list *data2*:

```
data2 = {74, 58, 26, 42, 66, 66, 58, 34};
discreteWaveletTransform[data2]
```

```
{53., -3., 16., 10., 8., -8., 0., 12.}
```

Setting the wavelet coefficient -3 to 0 we have the approximated list *data2Approximated*.

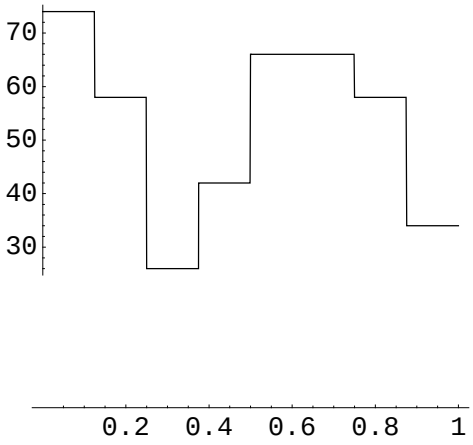
```
data2Approximated =
 discreteInverseWaveletTransform[{53, 0, 16, 10, 8, -8, 0, 12}]
```

```
{77, 61, 29, 45, 63, 63, 55, 31}
```

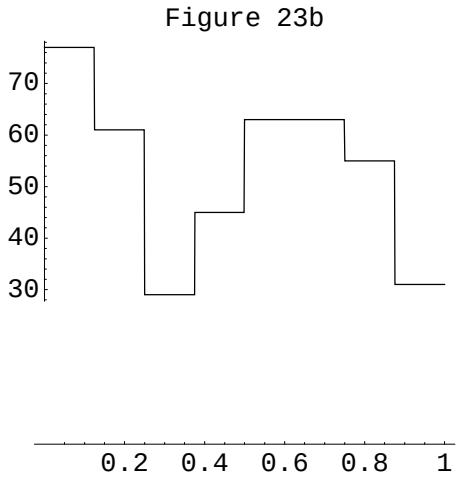
Using step functions — this time — we plot these two lists in Figures 23a and 23b and ask the reader to spot the difference!

```
Clear[x];
yStep2 = Table[φ[x, 3, k], {k, 0, 7}];
Plot[data2.yStep2, {x, 0, 1},
  PlotLabel → "Figure 23a", AxesOrigin → {0, 0}];
```

Figure 23a



```
Plot[data2Approximated.yStep2, {x, 0, 1},
  PlotLabel → "Figure 23b", AxesOrigin → {0, 0}];
```



Filters and Filter Banks with Matrices

In this section we give a matrix formulation of the Haar wavelet transform. Consider the, by now, well known example with the list {7, 5, 6, 2}. We apply the Haar transform and below we see again the intermediate and final results :

```
row1 = {7, 5, 6, 2};
discreteWaveletTransform[row1, 1]
```

```
{7., 5., 6., 2.}

{6., 4., 1., 2.}

{5., 1., 1., 2.}
```

■ Matrix formulation of the Haar transform

The same results can be obtained using the following matrices A_1, A_2 (filter banks):

$$\mathbf{A1} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{-1}{2} \end{pmatrix}; \quad \mathbf{A2} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{-1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

Matrix A_1 is associated with the first pair of lowpass/highpass filter. The intermediate result, row2 = {6, 4, 1, 2}, is obtained from the product *A1.data1*:

```
row2 = A1.row1
```

```
{6, 4, 1, 2}
```

Matrix A_2 is associated with the second pair of lowpass/highpass filter; only the first two elements of *row2*, i.e. {6, 4}, are affected by the filter bank. The result, *row3* = {5, 1, 1, 2}, is obtained from the product *A2.row2*:

```
row3 = A2.row2
```

```
{5, 1, 1, 2}
```

Therefore, in this case the discrete wavelet transform matrix is the matrix $A_2.A_1$

```
( discreteWaveletTransformMatrix = A2.A1) // MatrixForm
```

$$\begin{pmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} \end{pmatrix}$$

```
discreteWaveletTransformMatrix.row1
```

```
{5, 1, 1, 2}
```

Likewise, the discrete inverse wavelet transform matrix in this case is the matrix Inverse[A_1].Inverse[A_2].

```
(discreteInverseWaveletTransformMatrix =
   Inverse[discreteWaveletTransformMatrix]) // MatrixForm
```

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 0 \\ 1 & -1 & 0 & 1 \\ 1 & -1 & 0 & -1 \end{pmatrix}$$

```
discreteInverseWaveletTransformMatrix.row3
```

```
{7, 5, 6, 2}
```

```
Inverse[A1].Inverse[A2].row3
```

```
{7, 5, 6, 2}
```

It is fairly obvious how to construct the $2^i \times 2^i$ matrices A_1, A_2, ..., A_i needed to work with lists of length 2^i.

Observe now that the rows of the discrete wavelet transform matrix are orthogonal, but *not* orthonormal:

```
Table[discreteWaveletTransformMatrix[[i]].
  discreteWaveletTransformMatrix[[j]], {i, 1, 4}, {j, 1, 4}]
```

$$\left\{\left\{\frac{1}{4}, 0, 0, 0\right\}, \left\{0, \frac{1}{4}, 0, 0\right\}, \left\{0, 0, \frac{1}{2}, 0\right\}, \left\{0, 0, 0, \frac{1}{2}\right\}\right\}$$

We know from Linear Algebra that if the rows of a matrix are orthonormal, then its inverse is the same as its transpose. Consequently, the inverse in these cases can be very easily computed; in our example above, the inverse discrete wavelet transform matrix would be the matrix Transpose[A_1].Transpose[A_2] — instead of the matrix Inverse[A_1].Inverse[A_2].

Hence, to speed up computations, we need to normalize the rows of matrices A_1 and A_2. This is achieved by dividing *each* row of A_1 and the *first two* rows of A_2 by $\frac{1}{\sqrt{2}}$, their measure. We thus obtain the orthonormal matrices $A_1\,N$ and $A_2\,N$:

```
A1N = {{1/Sqrt[2], 1/Sqrt[2], 0, 0}, {0, 0, 1/Sqrt[2], 1/Sqrt[2]},
   {1/Sqrt[2], -1/Sqrt[2], 0, 0}, {0, 0, 1/Sqrt[2], -1/Sqrt[2]}};
A2N = {{1/Sqrt[2], 1/Sqrt[2], 0, 0}, {1/Sqrt[2], -1/Sqrt[2], 0, 0},
   {0, 0, 1, 0}, {0, 0, 0, 1}};
```

This way the *normalized* discrete wavelet transform matrix and its inverse are computed as follows:

```
( discreteWaveletTransformMatrixN = A2N.A1N ) // MatrixForm
```

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

```
(discreteInverseWaveletTransformMatrixN =
   Transpose[discreteWaveletTransformMatrixN]) // MatrixForm
```

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{2} & -\frac{1}{2} & 0 & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

and both are orthonormal as can be easily verified:

```
Table[discreteWaveletTransformMatrixN[[i]].
  discreteWaveletTransformMatrixN[[j]], {i, 1, 4}, {j, 1, 4}]
```

```
{{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}
```

The results of the transform are somewhat different now. To wit, we have:

```
discreteWaveletTransformMatrixN.row1
```

```
{10, 2, Sqrt[2], 2 Sqrt[2]}
```

```
discreteInverseWaveletTransformMatrixN.%
```

```
{7, 5, 6, 2}
```

This process is called the *normalization process* and is equivalent to working with the *normalized* Haar scaling and wavelet functions; that is, the functions $\varphi_k^i(x) = \sqrt{2^i}\,\varphi(2^i x - k)$, and $\chi_k^i(x) = \sqrt{2^i}\,\chi(2^i x - k)$, $k = 0, 1, ..., 2^i - 1$, are now used as bases of the spaces $\mathcal{V}^i$ and $\mathcal{W}^i$ respectively. In other words, the average value is computed by $\frac{(a+b)}{\sqrt{2}}$ whereas the difference is computed by $\frac{(a-b)}{\sqrt{2}}$. Strange as it might seem, the normalization process results in compressed images that are more pleasing to the eye as we can see below.

■ *Normalized* Haar wavelet transform algorithms.

■ Compression of arbitrary images — revisited

Here we work again on the Pedro picture using 10% and 3% of the detail coefficients. We see that in the first case the result is a picture almost identical to the original picture

```
originalMatrix = data1;
transformedMatrix =
  Map[discreteWaveletTransformNormalized, originalMatrix];
transformedTransposedMatrix = Transpose[transformedMatrix];
finalMatrix = Map[discreteWaveletTransformNormalized,
    transformedTransposedMatrix] // Transpose;
```

```
a = MemoryInUse[];
```

```
finalMatrixSparseNormalized = Map[
   Fold[If[Abs[#2] ≤ 15.0, Append[#1, 0], Append[#1, #2]] &, {}, #] &,
   finalMatrix];
Select[Flatten[finalMatrixSparseNormalized], # != 0 &] // Length
```

```
6826
```

```
b = MemoryInUse[]; b - a
```

```
276024
```

```
sparseFinalMatrixSparseNormalized =
 SparseArray[finalMatrixSparseNormalized]
```

```
SparseArray[<6826>, {256, 256}]
```

```
c = MemoryInUse[]; c - b
```

```
86080
```

```
backUpOneStep = Map[discreteInverseWaveletTransformNormalized,
   Transpose[sparseFinalMatrixSparseNormalized // Normal]];
backUpOneStepTransposed = Transpose[backUpOneStep];
approximatedMatrixNormalized =
  Map[discreteInverseWaveletTransformNormalized,
   backUpOneStepTransposed];
```

```
Select[Flatten[approximatedMatrixNormalized], # != 0 &] // Length
```

```
65536
```

```
ListDensityPlot[approximatedMatrixNormalized, Mesh → False];
```

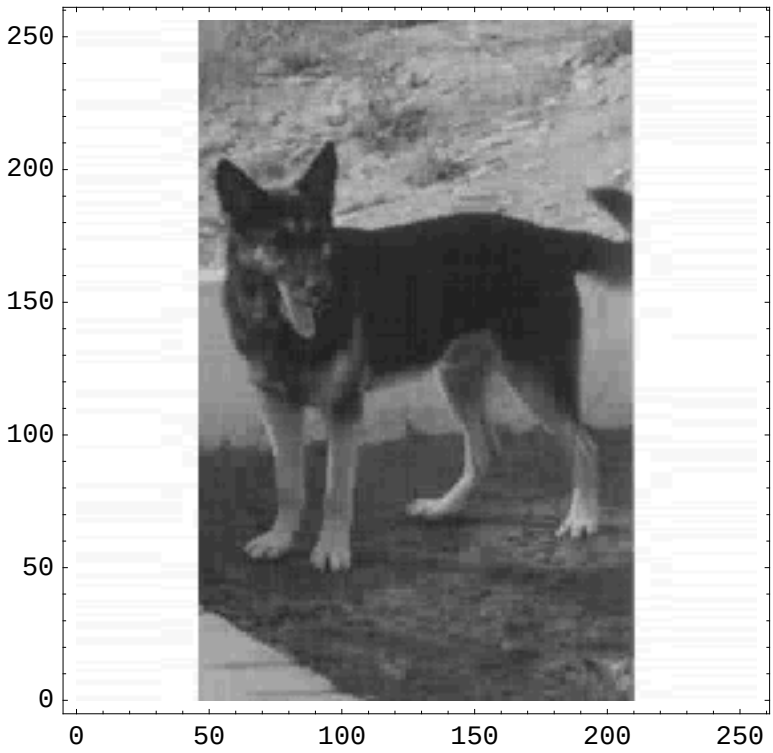


Figure *Pedro-3*: Picture of Pedro obtained using 10% of the details in the normalized wavelet transform; almost identical to *Pedro-1* (the original).

Let us see how far we can compress the picture now. Below we use 2216 nonzero elements and the sparse array can be stored in only 30K bytes of memory!

```
a = MemoryInUse[];
```

```
finalMatrixSparseNormalized = Map[
   Fold[If[Abs[#2] ≤ 35.0, Append[#1, 0], Append[#1, #2]] &, {}, #] &,
   finalMatrix];
Select[Flatten[finalMatrixSparseNormalized], # != 0 &] // Length
```

```
2216
```

```
b = MemoryInUse[]; b - a
```

```
276056
```

```
sparseFinalMatrixSparseNormalized =
 SparseArray[finalMatrixSparseNormalized]
```

```
SparseArray[<2216>, {256, 256}]
```

```
c = MemoryInUse[]; c - b
```

```
30592
```

```
backUpOneStep = Map[discreteInverseWaveletTransformNormalized,
   Transpose[sparseFinalMatrixSparseNormalized // Normal]];
backUpOneStepTransposed = Transpose[backUpOneStep];
approximatedMatrixNormalized =
  Map[discreteInverseWaveletTransformNormalized,
   backUpOneStepTransposed];
```

```
Select[Flatten[approximatedMatrixNormalized], # != 0 &] // Length
```

```
65536
```

```
ListDensityPlot[approximatedMatrixNormalized, Mesh → False];
```

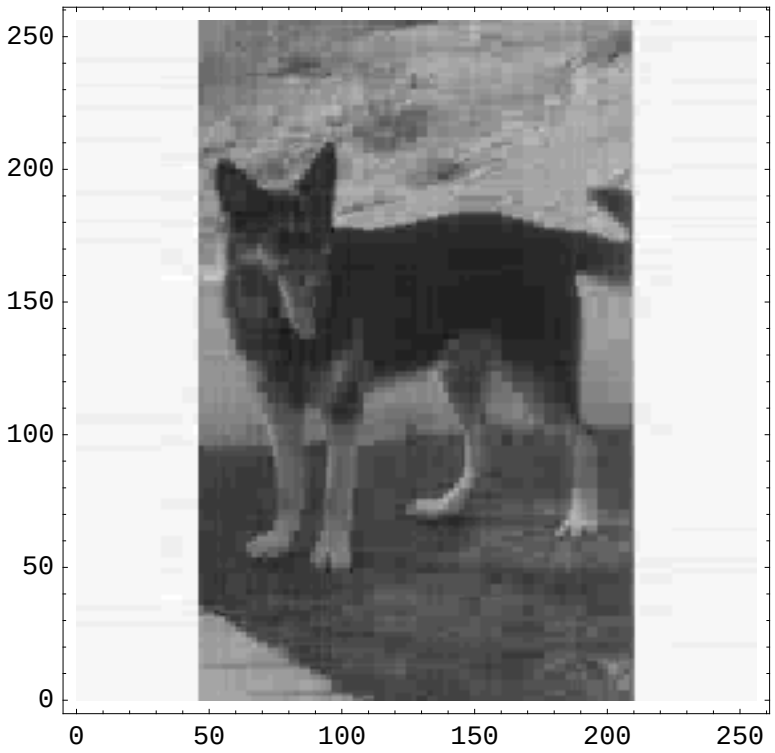


Figure *Pedro-4*: Picture of Pedro obtained using about 3% of the details in the normalized wavelet transform; almost identical to *Pedro-2* (obtained using 10% of the details in the *non*-normalized wavelet transform).

The advantages of normalization are obvious!

References

C. Mulcahy: Plotting and Scheming with Wavelets, *Mathematics Magazine* **69,** pp 323–343, 1996.

G. Nakos and D. Joyner: Linear Algebra with Applications. Brooks/Cole Publishing Company. New York, 1998.

G. Strang: Signal Processing for Everyone. November 1, 2000. Report downloaded from the internet.