The Discrete Wavelet Transform in S

G. P. Nason* and B. W. Silverman†

Journal of Computational and Graphical Statistics (1994)
3 163–191

Abstract

The theory of wavelets has recently undergone a period of rapid development. We introduce a software package called `wavethresh` that works within the statistical language S to perform one- and two-dimensional discrete wavelet transforms. The transforms and their inverses can be computed using any particular wavelet selected from a range of different families of wavelets. Pictures can be drawn of any of the one- or two-dimensional wavelets available in the package. The wavelet coefficients can be presented in a variety of ways to aid in the interpretation of data. The package's wavelet transform "engine" is written in C for speed and the object-orientated functionality of S makes `wavethresh` easy to use. We provide a tutorial introduction to wavelets and the `wavethresh` software. We also discuss how the software may be used to carry out nonlinear regression and image compression. In particular, thresholding of wavelet coefficients is a method for attempting to extract signal from noise and `wavethresh` includes functions to perform thresholding according to methods in the literature.

1 Introduction

This paper provides a tutorial on the use of the discrete wavelet transform for statistical purposes, and a guide to a publicly available package of routines, Nason (1993), for the statistical language S. The package is called `wavethresh` and is available from the StatLib archive. Appendix A gives full instructions on how to obtain the `wavethresh` package.

A gentle introduction to wavelet methods is provided by Strang (1993). For a more detailed discussion the reader is referred, for example, to Daubechies (1992) and Chui (1992). The statistical aspects of the package are mainly due to Donoho and Johnstone (1993). In this paper we concentrate on the discrete

*School of Mathematics, University of Bristol, University Walk, Bristol, BS8 1TW, UK
†School of Mathematics, University of Bristol, University Walk, Bristol, BS8 1TW, UK

wavelet transform. This is based on filtering ideas that have been discussed extensively in the engineering literature. Vaidyanathan (1990) and Vetterli and Herley (1992), provide detailed surveys and numerous references. Some other specific references are mentioned in Section 7 below.

We do not claim that wavelets are useful in all statistical curve and surface estimation problems. Our aim in making this software and tutorial available is to widen interest in, and access to, wavelet methods so that they can be tried and tested in practice, and a mature view thereby obtained of their usefulness and potential.

Standard linear regression techniques formulate a model of the response in terms of some explanatory variables. If a polynomial regression is appropriate then orthogonal polynomials may be useful. Although the resultant variables may be more complicated, the regression coefficients of the polynomial-variables are independent. This independence is desirable, especially when the modelling is of a changing process when modelling terms have to be added or dropped. Alternatively, linear regression may be extended by using nonparametric regression methods such as spline smoothing. Most of the methods mentioned above are linear in the response, but for wavelet regression with thresholding this is not the case. Wavelet regression is potentially highly locally adaptive; it retains the independence of regression coefficients, like orthogonal polynomials; and can, under the right conditions, provide an insightful interpretation of the data.

This paper is set out as follows. A brief discussion of the wavelet transform is given in Section 2. The next three sections give a tutorial guide to the `wavethresh` software. In Section 3 the one-dimensional transform and its inverse are discussed, and an explanation is given of the way in which the wavelet decomposition is held as an object in S. Section 4 covers various thresholding procedures for smoothing or data compression. Wavelet transforms for two-dimensional (image) data are discussed in Section 5 and image compression in Section 6. Section 7 gives precise details on the implementation of the software. The appendix gives details on how to retrieve and install the `wavethresh` software.

2 The Wavelet Transform

We first set the scene by remembering Fourier series. Given a function f, defined on $[-\pi, \pi]$ we can represent f exactly in terms of the Fourier basis $\{\exp(inx)\}_{n=-\infty}^{\infty}$ as follows

$$f(x) = \sum_{-\infty}^{\infty} c_n \exp(inx), \tag{1}$$

where the *Fourier coefficients* are computed by

$$c_m = (2\pi)^{-1} \int_{-\pi}^{\pi} f(x) \exp(-imx)\, dx. \tag{2}$$

Since $\exp(inx) = \cos(nx) + i \sin(nx)$ the Fourier series in (1) can be regarded as an expansion of f in terms of sine and cosine functions. We regard the series expansion as a transform, taking a function f into a set of coefficients c_n.

The wavelet transform is in some, but not all, ways similar to the Fourier transform. Given a function f we wish to expand that function in terms of some orthonormal basis functions ψ_ν. For example, the Fourier series expansion uses the orthonormal system $(2\pi)^{-\frac{1}{2}} \exp(i\nu x)$ (on $[-\pi, \pi]$). Wavelet expansions are orthogonal series expansions where the basis functions are constructed in an intriguing way. We will write our general wavelet basis element as ψ_{jk}, notice that there are *two* subscripts not one. This is because the wavelet basis functions are all dilations and translates of a single function called the *mother wavelet*, ψ. The *wavelet* ψ_{jk} is obtained from the mother wavelet by shrinking by a factor of 2^j and translating by $2^{-j}k$, to obtain

$$\psi_{jk}(x) = 2^{\frac{j}{2}} \psi(2^j x - k). \tag{3}$$

so that the j subscript represents the dilation number and the k subscript will represent the translation number. The scale factor $2^{\frac{j}{2}}$ normalizes ψ_{jk} so that $||\psi_{jk}|| = ||\psi||$.

For certain choices of ψ the set of functions ψ_{jk} form an orthonormal basis for all functions in $L^2(\Re)$, and therefore we shall use the wavelets, $\{\psi_{jk}\}$, to approximate functions.

The continuous wavelet series representation is:

$$f(x) = \sum_{jk} f_{jk} \psi_{jk}(x), \tag{4}$$

where the *wavelet coefficients* are found in the usual way:

$$\begin{aligned} f_{jk} &= \int_{-\infty}^{\infty} f(x) \psi_{jk}(x)\, dx \\ &= < f, \psi_{jk} > \end{aligned} \tag{5}$$

where $< \cdot, \cdot >$ denotes inner product. Clearly, given a function we will wish to compute its wavelet coefficients. The software described later computes wavelet coefficients but uses a discretized version of (5).

2.1 What can wavelets offer?

This question is discussed in detail by Strang (1993) so we will say relatively little here. In the Fourier transform of a function f on $(-\infty, \infty)$:

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

we usually identify t with time and ω with frequency. To obtain information about a particular frequency, we have to integrate over the whole domain of f from $t = -\infty$ to $t = \infty$, even though we might want to know the frequency behaviour of f only over a particular time period. The wavelet transform is one way in which such local frequency information can be obtained. Wavelets provide *time-frequency localisation* in that the coefficient f_{jk} gives information about the function f near time point $2^{-j}k$ and near frequency proportional to 2^j. Daubechies (1992) has more details and an example of this.

A simple example of a wavelet basis is the Haar basis, generated from the mother wavelet

$$\psi(x) = \begin{cases} -1 & 0 \le x \le \frac{1}{2}, \\ 1 & \frac{1}{2} \le x \le 1, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

It is easy to convince oneself that wavelets derived from the Haar mother wavelet form an orthonormal system by observing:

1. wavelets with different translate numbers but on the same scale do not have intersecting supports;

2. wavelets on different scales either have non-intersecting supports or if they do, then one wavelet takes the value $-k$ and then k over a set where the other wavelet is constant (for some k).

3. $< \psi_{jk}, \psi_{jk} >= 1$ for all j, k.

The major impetus to the theory of wavelets was the realization that there are many other families of wavelets, arising from mother functions more regular than the Haar function (6). Some of the useful properties of wavelets are exhibited by the Haar wavelet basis, but in many contexts more regular wavelets are useful.

Release 2.2 of the **`wavethresh`** software, the latest release at the time of writing, includes two families of wavelets. Both these families are due to Daubechies (1988) and we refer to them as the "extremal phase" or "least-asymmetric" wavelets. The wavelets in each of the families have compact support. As in Daubechies (1988) we index each mother wavelet in a family by N. For each family the regularity of the mother wavelet (and hence all the derived wavelets) is proportional to N. If you look forward to Figure 4 you will see a picture of Daubechies' "extremal phase" wavelet for $N = 2$. Figure 4 shows an irregular wavelet but as N increases they become smoother. In the **`wavethresh`** software the number N is identified with the **`filter.number`** argument.

Wavelets usually possess another interesting property, that of "vanishing moments". We say a wavelet has L vanishing moments if for each $l = 0, \ldots, L-1$ a wavelet ψ satisfies:

$$\int x^l \psi(x)\, dx = 0.$$

This property has important implications for applications. Daubechies (1992) provides a good example that shows that this property ensures that fine-scale wavelet coefficients will only be large where a function or its derivatives have singularities. This property promotes the use of wavelets for compression techniques because they provide *sparse* representations of functions.

Another often quoted benefit of wavelet methods is that wavelet bases are capable of representing various classes of functions more efficiently than, say, Fourier bases (see Donoho and Johnstone (1993) for example). For example, take functions that are only piecewise continuous, that is they contain some discontinuities. You will need many Fourier basis functions to represent the discontinuities accurately, and the effect of the basis functions will be global. Wavelets will be able to represent the discontinuities more efficiently, and at the same time they will be local, and not affect the representation elsewhere.

The discrete wavelet transform is fast, in principle faster even than the fast Fourier transform. However, both the standard FFT and the discrete wavelet transform only operate on data sets that contain 2^M observations (for some M). There are some methods of overcoming the difficulty, but these are not implemented in the current release.

3 The Discrete Wavelet Transform in S

This section assumes that someone has installed the discrete wavelet transform software. If you want instructions in retrieving, unpacking and setting up the software see Appendix A. The transform software is an implementation of the method described in Mallat (1989b), but uses the compactly supported orthonormal wavelets as described by Daubechies in (1988) and (1992). We discuss our implementation in greater detail in Section 7.

3.1 Accessing the software

We assume that the `wavethresh` software is installed somewhere on your system. In this paper we will invent a directory name so that we can refer to the location of the software. On your system the directory where the software is stored will most likely be different. The directory we will invent is `/stats/WAVELETS/DISTRIB2.2`. Accessing the `wavethresh` software is made possible after the the following function has been executed in S:

```
> attach("/stats/WAVELETS/DISTRIB2.2/.Data")
```

Expert users may wish to use the `search()` or `objects()` functions to verify that the `wavethresh` directory has been attached. Non-expert users can simply use the `wvrelease()` function to verify this:

```
> wvrelease()
```

```
S wavelet software, release 2.2, installed
Copyright Guy Nason 1993
```

If you don't get this message (or something like it) then the wavelet software is not present — see your local guru! If you want to use the software often then you may want to attach the software every time you start up S. This is accomplished by means of the `.First()` function. The following lines form a plausible `.First()` function:

```
> .First <- function(){
 attach("/stats/WAVELETS/DISTRIB2.2/.Data")
 wvrelease()
 }
```

3.2 Getting help on the software

You can get help on any of the wavelet functions or objects by using the S help facility. For example, to obtain help on the `wd` wavelet decomposition function simply type:

```
> help(wd)
```

You may be able to use the S-Plus `help.start()` function that starts an interactive help viewer. See your local documentation to see if this is available. Typing "wavelet" as a help search topic lists all the help available for the wavelet software.

3.3 Preparing an example

We assume that you have started S and attached the `wavethresh` software. You will probably want to try out the wavelet techniques on your own data. However, for the purposes of the tutorial, we will create some simulated data. We create a sampled version of the contrived function

$$y(x) = \begin{cases} 4x^2(3-4x) & \text{for } x \in [0, \frac{1}{2}] \\ \frac{4}{3}x(4x^2-10x+7) - \frac{3}{2} & \text{for } x \in [\frac{1}{2}, \frac{3}{4}] \\ \frac{16}{3}x(x-1)^2 & \text{for } x \in [\frac{3}{4}, 1] \end{cases}$$

and sample it 512 times in the interval $[0, 1]$. Figure 1 shows y plotted against x.

The `wavethresh` software contains a function called `example.1()` that computes x and y automatically for you.

```
> x <- example.1()
> y <- x$y
> x <- x$x
```

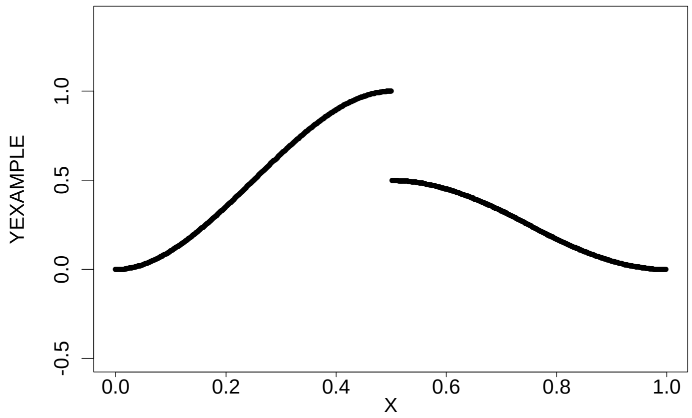

Figure 1: A plot of the function y versus x

This sampled function can be viewed using the S `plot()` function (first start up your favourite graphics device, such as `X11()`, `openlook()` or `motif()`).

We will add some Gaussian white noise to `y` to create the vector `ynoise` by

```
> ynoise <- y + rnorm(512, s=0.15)
> plot(x, ynoise, type="l")
```

This time the plot should look something like Figure 2.

3.4 Applying the discrete wavelet transform

The function to do a discrete wavelet transform is `wd` which stands for "Wavelet Decomposition". This function can take several arguments, a complete explanation is given in the `wd` help page. The first argument, `data`, is the vector to which you wish to apply the transform. The length of this vector, N, must be a power of 2, that is why we specified 512 as the length of y above (let $N = 2^M$, say). The remaining arguments specify the type of wavelet that is used, the regularity or smoothness of the wavelet and the method of handling the transform at the boundaries. For Release 2.2 of `wavethresh` these arguments are:

`data` the vector that you wish to transform;

`filter.number` determines the regularity of the wavelet. The wavelets get smoother as `filter.number` increases (see Daubechies (1992));

`family` determines which family of wavelets is used. At present there are two families. They are both compactly supported orthonormal wavelets and are defined in Daubechies (1992). The family names are `DaubExPhase`

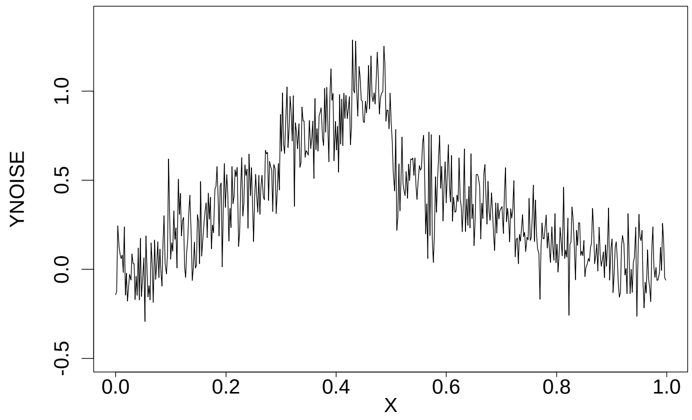

Figure 2: The sampled function with Gaussian white noise

and `DaubLeAsymm`, the second of these families is less asymmetric than the first! Ten members of the `DaubExPhase` family are available (indexed by `filter.number`s between 1 and 10) and seven members of the `DaubLeAsymm` family (indexed by 4 to 10). Note that the first wavelet in the `DaubExPhase` family is the Haar wavelet as in (6);

`bc` the boundary handling conditions. This can be set to `periodic` (the default) or `symmetric`;

`verbose` if this is true then the `wd` functions prints messages as it performs the transform.

We will apply the discrete wavelet transform (DWT) to the `ynoise` data by

```
> ywd <- wd(ynoise)
```

The `wd()` function applies `filter.number=2` for the `DaubExPhase` family by default.

3.5 Wavelet decomposition objects

The S object `ywd` is an example of a `wd.object` and it has class `wd`. The `wd` class objects are lists, and you can find out what they contain by using the `names()` function or by looking at the `wd.object` help page.

There are various methods for this class of object. Probably the simplest is the `summary()` method. For example,

```
> summary(ywd)
```

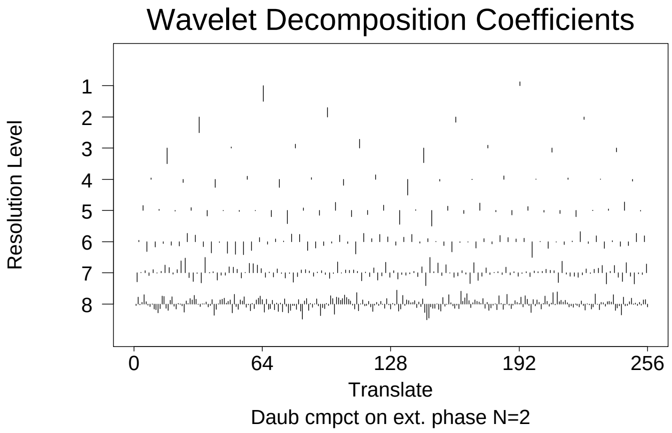

Figure 3: Wavelet coefficients for the `ynoise` data

```
Levels:  9
Length of original:  512
Filter was:  Daub cmpct on ext. phase N=2
Boundary handling:  periodic
```

tells us that the length of the original data was 512 samples, the transform was performed using the $N = 2$ Daubechies compactly supported wavelet from the `DaubExPhase` family, the decomposition consists of 9 levels and the boundary handling was periodic. The number of levels refers to the number of levels of smoothed data and wavelet coefficients (and therefore $2^{\texttt{levels}} =$ length of data).

Another method is `plot()`, which plots the wavelet coefficients in the manner of Donoho and Johnstone (1993). This method is applied by

```
> plot(ywd)
```

and results in the plot shown in Figure 3. Roughly speaking the horizontal axis of Figure 3 corresponds to spatial position along the x axis of Figure 1. It is possible to change the x-axis of Figure 3 so it depicts the actual x values rather than the translates. The vertical axis of Figure 3 represents a frequency-like quantity, the coefficients at the bottom are "high-frequency" information and the "frequency" decreases as you move up the axis. So the coefficients at the left-hand side of Figure 3 correspond to the left-hand side of the function in Figure 2 and the right-hand side in both correspond. The reason the x-axis in Figure 3 is labelled 0 to 256 is that the number of wavelet coefficients at the highest resolution level (at the bottom of Figure 3) is exactly half the number of original data points and the number of coefficients decreases by half at each level. At the highest level, the coefficient with translation number k is plotted at position $k + \frac{1}{2}$.

3.6 Exact reconstruction

From `ywd` we can exactly reconstruct the original sampled function. We will do this and compare it to the original. The function to do the inverse discrete wavelet transform (IDWT) is `wr()` which stands for "wavelet reconstruction". So, reconstructing

```
> ywr <- wr(ywd)
```

Note that it is not necessary to specify the type of wavelet because `wr()` works it out. The `wr()` function can return a `wd` class object, but by default it produces a vector — this vector is the reconstructed function at the highest possible resolution level. You can plot the reconstructed values by

```
> plot(x, ywr, type="l")
```

and you should get exactly the same plot as in Figure 2. To check that the reconstruction is exactly the same, up to numerical error, we can subtract the original from the reconstruction and look at the error:

```
> max(abs(ywr - ynoise))
[1] 1.109357e-11
```

and as you can see the error is small.

3.7 What do the Daubechies wavelets look like?

The orthonormal basis functions involved with the Fourier transform are familiar since they are just sine and cosine functions. For the Daubechies wavelets things are not so simple and there is no known closed form formula for the wavelets. The wavelets are the result of a process that says that we want functions that are orthonormal, have good time-frequency localisation properties and relate to each other by dilation and translation. The nearest one can get to a closed-form expression is an infinite-product expression for the Fourier transform of the wavelets. All this is more fully explained in Daubechies (1992).

In any event we do not really need to have a closed form formula since we have an algorithm that will perform the inverse discrete wavelet transform. To get a picture of a wavelet, all we need to do to arrange for a wavelet series to have only one nonzero wavelet coefficient, and all the others set to zero. Then we apply the inverse wavelet transform to this series and plot the reconstruction. The `wavethresh` software has this built in to the `draw` function. If we take our `ywd` `wd.object` above and issue the command

```
> draw(ywd, enhance=F)
```

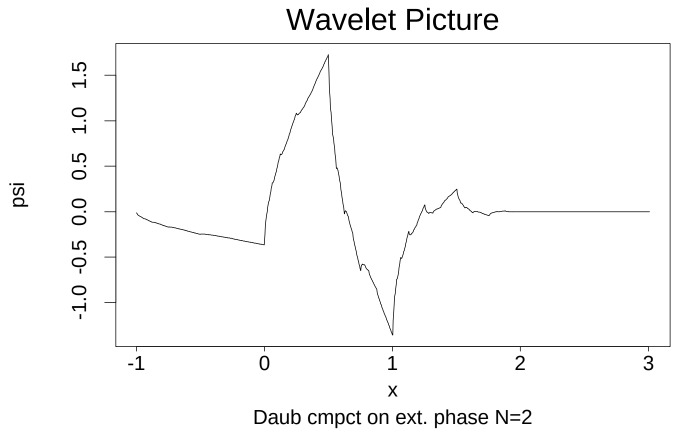

Figure 4: Daubechies' "extremal phase" wavelet $N = 2$

then you should see a picture of Daubechies' $N = 2$ "extremal phase" wavelet for `filter.select=2` as in Figure 4. The Daubechies' wavelets in `wavethresh` are all compactly supported. The support is related to the `filter.number` or N in Daubechies' notation. The support for the "extremal phase" wavelets is $[N-1, N]$. So the support of the wavelet in Figure 4 is $[-1, 2]$ but you will notice that the size of the wavelet is not large over the whole of that interval. For example, the significant parts of the Daubechies' $N = 2$ wavelet are contained within the interval $[0, 1.2]$ which we would term the *effective* support of the wavelet. For the smoother wavelets the effective support is much less than the actual support. The `enhance` option to `draw()` draws the wavelet on the effective support. In the `wavethresh` package we define the effective support in the following way. Suppose $\psi(x)$ is the wavelet under consideration. Define z_0 to be the maximum absolute value of $\psi(x)$ multiplied by the `draw()` argument `efactor`. Then define the set

$$A_{\texttt{efactor}} = \{x : |\psi(x)| > z_0\}.$$

We define the effective support of ψ to be the smallest interval (or square region for two-dimensions) containing $A_{\texttt{efactor}}$. The argument `efactor` is under user control but the default of 0.05 seems to work well.

For an example, try the following commands that show the difference between enhanced and non-enhanced pictures:

```
> draw.default(filter.number=10, enhance=F)
> draw.default(filter.number=10)
```

You may also have guessed that the `draw` function is generic and methods exist for one- and two-dimensional wavelets. We will look at some two-dimensional pictures of the wavelet later.

4 Wavelet shrinkage—smoothing

We now move on to the statistical techniques of wavelet regression and smoothing. Following Donoho and Johnstone (1993) we have implemented a thresholding function. The thresholding function is generic and called `threshold()`. The idea behind thresholding is the removal of small wavelet coefficients, considered to be noise. This leaves the large coefficients in the `wd` object that can then be used to estimate the signal after reconstruction.

There are many ways to threshold. To threshold using our software you have to choose a thresholding "policy". The policies specify how the threshold is computed as listed below. Once the threshold has been computed it is applied to the coefficients either as a hard or soft threshold, as specified by the argument `type`. Finally, each of the policies can be applied separately to each level or a policy can be applied to a group of levels simultaneously; this is controlled by the `by.level` and `levels` arguments.

The policies are:

`universal:` the threshold is computed as

$$\lambda = s\sqrt{2\log M}$$

where M is the number of data points (equivalently the number of wavelet coefficients) and s is an estimate of the variation of the coefficients (on the standard deviation scale). The `dev` argument allows the user to replace the default measure of variation (`var`) by their own choice, for example mean absolute deviation (`mad`). This type of thresholding was proposed by Donoho and Johnstone (1993).

`manual:` the threshold is supplied by the user.

`probability:` the user supplies a probability `value` p. The threshold is then the pth quantile of the coefficients.

The types are:

`hard:` the coefficients are compared to the threshold(s). If a coefficient is smaller in absolute magnitude than the threshold it is removed, otherwise it is left alone ("keep" or "kill").

`soft:` the coefficients are modified by the formula:

$$d_{jk}^{\text{new}} = \text{sgn}(d_{jk})(|d_{jk}| - \lambda)_+$$

where λ is the threshold.

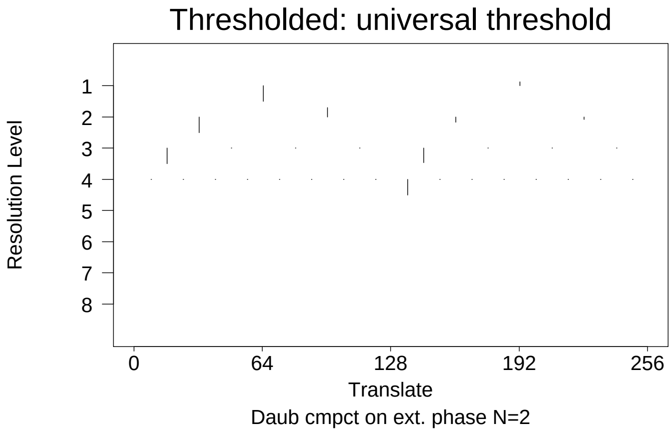

Figure 5: Wavelet coefficients after universal thresholding

4.1 Example of thresholding

We apply thresholding to the coefficients illustrated in Figure 3. We use the `universal` policy:

```
> threshy <- threshold(ywd)
> plot(threshy)
```

The `threshy` object is of class `wd`, and so `plot` uses the `plot.wd` function to plot the wavelet coefficients as in Figure 5. If you compare Figure 5 to Figure 3 you will see that many of the smaller coefficients have disappeared. We can now reconstruct the function corresponding to the thresholded coefficients by:

```
> yrecon <- wr(threshy)
> plot(x, yrecon, type="l", xlab = "x", ylab =
                "Reconstructed function")
```

The plot should look like Figure 6 which is a slightly "smoother" plot than Figure 2 and should look a bit more like Figure 1. However, the plot in Figure 6 still looks jagged. Figure 7 shows what happens if a smoother wavelet is used. We have repeated the decomposition, shrinkage and reconstruction using the Daubechies wavelet selected with `filter.select`=4. Here is the command we issued to S:

```
> plot(x, wr(threshold(wd(ynoise, filter.number=4))),
xlab="x", ylab="Reconstructed function", type="l")
```

As you can see the reconstruction looks better. You may like to try the procedure again, but for different wavelets and see what you think! The default

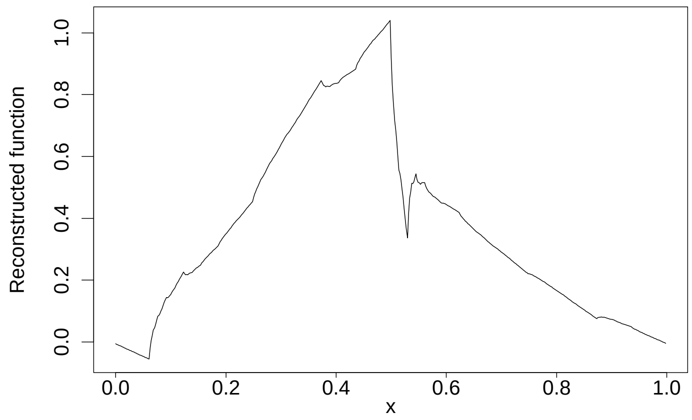

Figure 6: Reconstructed function after universal thresholding

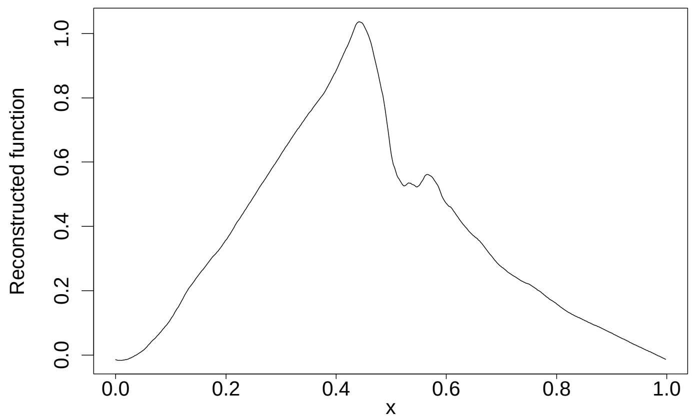

Figure 7: Reconstruction with a smoother wavelet and universal thresholding

action of `threshold()` is to select the universal threshold of Donoho and Johnstone (1993), but you can select your own threshold value. To do this you need to tell the `threshold()` function that *you* want to set the threshold — do this by setting `policy="manual"` and supply the value of the threshold by `value=`t, where t is your threshold. It is useful to know what threshold has been used and setting the `verbose=T` argument causes the thresholding information to be printed.

5 The two-dimensional discrete wavelet transform

The theory for the two-dimensional transform is a straightforward extension of that for the one-dimensional transform. More details can be found in Section 7.5 and in Mallat (1989b). Raw S does not appear to have an `image()` function, although S-Plus does. If you only have raw S you can still do all the 2D wavelet transforms but you will have to invent your own method of presenting the answers. The only implementation difference is to the function `plot.imwd`. This has a `package` argument that causes the function not to use the `image()` function when raw S is being used.

Images in S are stored as matrices, with the (i, j)th element of a matrix containing an image intensity value. The S-Plus function `image` takes an image matrix and displays it on the current graphics device. So that our PostScript prints are displayed the same as our images we usually start up our `X11()` graphics device using

```
> X11(bwimage.colors)
```

where `bwimage.colors` sets up the `X11()` device to display colors as shades of grey. When using this setup the intensity values represent shades of grey with 1 representing white and 249 representing black, and values between are uniformly increasing shades of grey.

Using this setup, the original image that we work on is displayed in Figure 8 and is stored in a matrix called `lennon`. The function to do the DWT on images is `imwd`:

```
> lwd <- imwd(lennon, filter=8)
```

This creates an object of class `imwd` and the five methods written for this class are `summary`, `plot`, `threshold`, `draw` and `compress`. The `imwd` uses `filter.number=2` by default, but you can alter this as the above example shows. The results of `summary` applied to `lwd` are

```
> summary(lwd)
Levels:  8
Original image was 256 x 256  pixels.
Filter was:  Daubechies compact orthonormal wavelet N=8
```

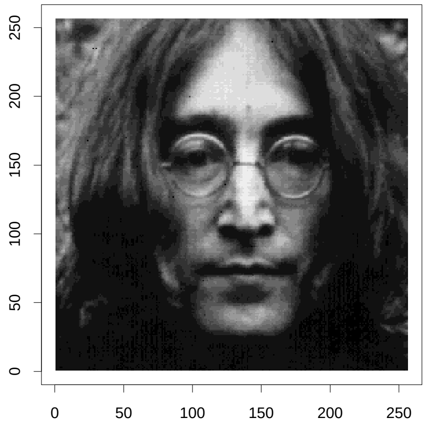

Figure 8: Original image

5.1 Viewing coefficients

This time coefficients are viewed as an image, rather than a plot. The `plot` function has a method `plot.imwd` written to "plot" the coefficients. The command

```
> plot(lwd)
```

produces a plot like Figure 9. There are other ways of viewing the coefficients, for example the `plot.type="rows"` puts the level/orientation sub-images into tabular form. The coefficient's image is precisely the same size as the original and is arranged into L-shaped blocks (after Mallat (1989b)) consisting of 3 sub-blocks. Figure 10 illustrates how the coefficients are arranged. The "S" in the bottom left hand corner refers to the number resulting multiply smoothed data, and is like an "average" intensity for the whole image. Note that each sub-image retains some features of the original picture.

As in the one-dimensional case it is possible to obtain a picture of the wavelets that you are using. Figure 11 shows a picture of Daubechies' least asymmetric $N = 10$ two-dimensional wavelet. This picture was produced using the command:

```
draw.default(filter = 10, family = "DaubLeAsymm",
    resolution = 256, dim= 2)
```

but you can use `draw()` directly on an `imwd` object to obtain a picture of the wavelets that were associated with your wavelet decomposition.

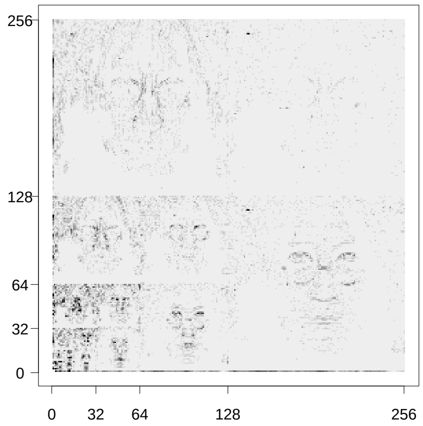

Figure 9: Wavelet coefficients of original image

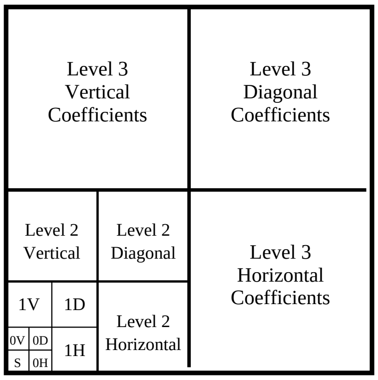

Figure 10: Layout of wavelet coefficients for image `plot` method (after Mallat (1989b))

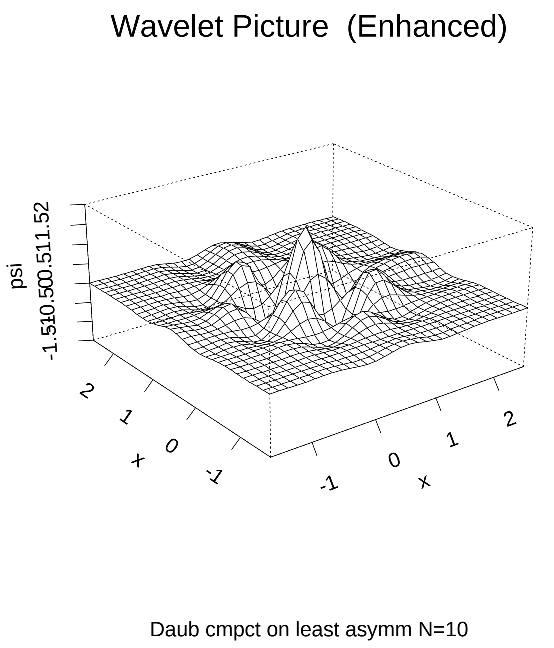

Figure 11: Daubechies' least asymmetric $N = 10$ two-dimensional wavelet

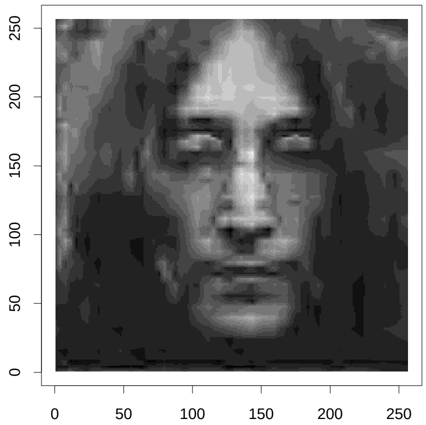

Figure 12: Reconstructed universally-thresholded Lennon image

5.2 Wavelet shrinkage and compression for images

Wavelet shrinkage is achieved by the `threshold()` function as before. However, for images the `threshold()` function returns an object of type `imwdc`, this is a compressed 2-dimensional wavelet decomposition object. The process of thresholding reduces many wavelet coefficients to zero, and these zeroes are not stored in an `imwdc` object. Section 6 gives more details on the compression of images with wavelets.

The reconstruction function is `imwr` and it returns the highest resolution reconstructed function as a matrix. Therefore, the matrix returned by `imwr` can be directly displayed using `image`. Figure 12 shows the reconstructed, universally-thresholded version of the example image. This was produced with the command:

```
image(imwr(threshold(lwd)))
```

6 Image Compression

In this section we describe how the `wavethresh` package can compress images. DeVore *et al.* (1992) provide a fascinating insight into methods of compression using wavelets. As we mentioned before thresholding typically sets many of the wavelet coefficients to zero. If you type

```
> lwdt <- threshold(lwd)
```

and then examine the class of `lwdt` with

```
> class(lwdt)
[1] "imwdc"
```

and notice that the `lwdt` is an object of class `imwdc`, this is a compressed image wavelet decomposition object. It is possible to uncompress this type of object into an ordinary image wavelet decomposition object by typing

```
> lwdt.big <- uncompress(lwdt)
```

To see how much space we save by compression we can use the `object.size()` function as follows:

```
> object.size(lwdt)
[1] 11971
> object.size(lwdt.big)
[1] 526991
```

and, as you can see, the compressed object is about 44 times as small as the uncompressed object (in terms of actual file size required to store the objects, the compression ratio is nearer 60). It is important to realize that

- this compression ratio is quoted only for this image using universal thresholding;
- the actual reconstruction from the thresholded wavelet decomposition is not convincing, and the compression is excessively lossy. Better accuracy would be achieved by specifying a lower threshold value, leading to a lower compression ratio, by using the `manual` policy option in `threshold()`.

You may want to delete the `lwdt.big` object, as it is large.

6.1 How does `wavethresh` compress objects?

Each `imwd` object is a list. The components of the list contain not only the wavelet coefficients but also other useful information. The wavelet coefficients are stored as vectors, one for each combination of level and either horizontal, vertical or diagonal detail. So the process of compression for `imwd` objects relies on the ability to compress ordinary vectors.

Recall that we are only interested in compressing vectors that contain many zeroes. So compression of a vector v works as follows:

1. let n be the length of v;
2. let r be the number of zeroes within v;

   if $n < 2r$ then return v uncompressed;

otherwise extract the position and value of the non-zero elements of v;

3. return the position and value;

For example, suppose we wish to compress the vector `v=c(0,0,99)`, this is achieved with the command:

```
> compress(c(0,0,99))
$position:
[1] 3

$values:
[1] 99

$original.length:
[1] 3

attr(, "class"):
[1] "compressed"
```

The only non-zero element in the vector was at position 3, and this is indeed the value of the `$position` component. A vector is easily uncompressed in S by using the position/value information. If you are knowledgeable about S you will realize that, for this example, the `compressed` object actually takes up a bit more space than the original vector! This is because the `compressed` object stores a small amount of extra information such as the original length of the uncompressed vector, and the class of the object. However, the size of these components remains constant whatever the size of the uncompressed vector. It is certainly true that we could invent a better compression scheme, we could possibly encode the `$values` and `$position` numbers using some advanced compression, but we feel that the gains would be minimal over what is already achieved by discarding the zeroes after thresholding. The compression scheme we adopt has two main advantages:

- it is simple, both in design and implementation;
- the operations are trivial to vectorize in S, and are therefore fast.

The same is also true of the uncompression scheme. However, image quality is another matter that we do not address here (however both DeVore *et al.* (1992) and, to a certain extent, Donoho and Johnstone (1993) do).

7 Implementation of the Wavelet Transform

This section is a reference for the `wavethresh` implementation of the discrete wavelet transform (DWT) and the inverse discrete wavelet transform (IDWT) software. The algorithm follows exactly that described by Mallat (1989b). and this report should be read with it. The DWT algorithm as described by Mallat (1989b) is a special case of a two channel subband coder using the conjugate quadrature filters of Smith and Barnwell (1986). Vaidyanathan (1990) provides a comprehensive survey and comparison of many filtering methods including subband coders; other significant contributions include Vetterli (1984), Mintzer (1982; 1985) on filter design and Smith and Eddins (1990) on subband coding for images

To proceed with this report we assume that we have a scaling function $\phi(x)$ and from this we can obtain a "mother" wavelet $\psi(x)$ and from this obtain a family of wavelets

$$\psi_{jk}(x) = 2^{\frac{j}{2}}\psi(2^j x - k)$$

such that $\{\psi_{jk}\}_{(j,k)\in\mathbf{Z}^2}$ forms an orthonormal basis for $L^2(\Re)$, the vector space of measurable, square-integrable one-dimensional functions. The rest of this report is concerned with expansions of functions in $L^2(\Re)$ with respect to wavelet bases. In what follows the coefficients of such expansions will be written as the vector d^j_k with j representing the scaling and k the translation.

7.1 Computing the DWT

As in Mallat (1989b) we begin with a set of $N = 2^M$ data

$$c_0^M, \ldots c_{N-1}^M. \tag{7}$$

In all that follows, in contrast to the double subscript notation c_{jk} used previously, the superscript denotes the resolution level, and the subscript represents the coefficient within that level. The reason for promoting the j to a superscript is that the k subscript may get a little more complicated! The algorithm consists of $M = \log_2 N$ stages, and the superscript M signifies that this is the original data at the highest resolution level. At each stage we produce a sequence of smoothed c and a sequence of detail d described by the formulae below.

We do not consider ways of extending data sets of other sizes to length 2^M, but natural possible approaches are periodic extension and symmetric reflection (both considered in detail by Smith and Eddins (1990)), as well as zero-padding, boundary value replication, and anti-symmetric reflection. Note that the choice of boundary conditions for the DWT itself is a separate matter, discussed below and in Section 3.4.

At each stage, new c and d are produced from the previously smoothed data c by using finite impulse response filters h and g as in equations (9) and (16) below. The filters h and g are intimately related by the relation

$$g(n) = (-1)^n h(1-n), \tag{8}$$

and they are known as quadrature mirror filters (see Vaidyanathan (1990) for further information on filter terminology and design). It is because of this relation that `filter.select()` need only know one of the pair (h in our case) for a particular wavelet family. The filter h is a smoothing filter, and the filter g is a highpass filter. If periodic boundary conditions are used for the DWT the lengths of the new c and d are exactly half the length of the previous c, so the total length of the c and d at each stage remains the same throughout the algorithm. If symmetric boundary reflection is used in the DWT, then some extra numbers are required at either end of the c and d filters. This explains why the lengths of cs and ds obtained by `accessC()` and `accessD()` functions are not necessarily exact powers of two when the `boundary=T` option is supplied. An increase in the filter length produces a corresponding increase in the number of these "extras".

In order to allow different boundary methods to be used, we define the sequences $f[j]$ and $l[j]$ to represent the first and last indices for a particular sequence at a particular level. So, for example, if we say a sequence s at level 3 has $f[3] = -4$ and $l[3] = 3$, then the sequence elements are:

$$s^3_{-4}, s^3_{-3}, \ldots, s^3_2, s^3_3$$

For the original data in (7) we would have $f[M] = 0$ and $l[M] = N - 1$. With periodic boundary handling it is simple to predict the values of $f[j], l[j]$ for $j = 0, \ldots, M - 1$, since the sequence of c and d halve exactly at each step. However, for symmetric end-reflection boundary handling we will need to develop some inequalities below that specify exactly where the start and finish for each sequence are. Sometimes we will subscript the f, l with C or D to make it clear which sequence we are referring to. For both periodic and symmetric boundary handling the values for f_C, l_C, f_D, l_C are usually precomputed by the `first.last()` function.

Level j to level $j - 1$

We now describe the decomposition of a sequence at level j into a sequence $j-1$. A decomposition means extracting a "smoother" signal at a lower resolution *and* extracting the signal detail at that resolution. The first decomposition step starts at level M and produces level $M - 1$.

At level j we have data

$$c^j_{f[j]}, \ldots, c^j_{l[j]},$$

with $f[j] \leq 0 \leq l[j]$. The smoothing operation to the next level, $j-1$ is achieved with the filter

$$h(n), \quad 0 \leq n \leq N_h - 1,$$

that has N_h nonzero coefficients. The smoothing formula is a convolution followed by dyadic decimation as in Mallat (1989b):

$$c^{j-1}_k = \sum_{n \in \mathbf{Z}} h(n - 2k) c^j_n \tag{9}$$

$$
\begin{aligned}
&= \sum_{m\in\mathbf{Z}} h(m)c^j_{m+2k} \\
&= \sum_{m=0}^{m=N_h-1} h(m)c^j_{m+2k}. \qquad (10)
\end{aligned}
$$

To identify which k we can compute, and to obtain exact reconstruction the summation indices must satisfy the following inequalities (for symmetric end-reflection):

$$0 \le m \le N_h - 1, \qquad (11)$$

$$f[j] \le m + 2k \le l[j]. \qquad (12)$$

From (12) we can write

$$\tfrac{1}{2}(f[j] - m) \le k \le \tfrac{1}{2}(l[j] - m). \qquad (13)$$

Using (11) and (13) we can obtain the range of the next level coefficient indices:

$$\tfrac{1}{2}(f[j] - N_h + 1) \le k \le \tfrac{1}{2}l[j].$$

Thus, the next level coefficient indices are

$$f[j-1] = \lceil \tfrac{1}{2}(f[j] - N_h + 1) \rceil, \qquad (14)$$

and

$$l[j-1] = \lfloor \tfrac{1}{2}l[j] \rfloor, \qquad (15)$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x, and $\lfloor x \rfloor$ is the largest integer less than or equal to x. Formulae (14) and (15) are built into the `first.last()` function.

The detail, d, are computed as

$$d^{j-1}_k = \sum_{n\in\mathbf{Z}} g(n-2k)c^j_n. \qquad (16)$$

The condition (8) above follows from Daubechies (1992) (formulae 5.1.34). Hence combining formulae (16) and (8) we have

$$
\begin{aligned}
d^{j-1}_k &= \sum_{n\in\mathbf{Z}} (-1)^n h(2k+1-n)c^j_n \\
&= \sum_{m=0}^{m=N_h-1} (-1)^{1-m} h(m)c^j_{2k+1-m} \qquad (17)
\end{aligned}
$$

The inequalities are similar. From (17) and (11) we have

$$f_D[j-1] = \lceil \tfrac{1}{2}(f_C[j] - 1) \rceil \qquad (18)$$

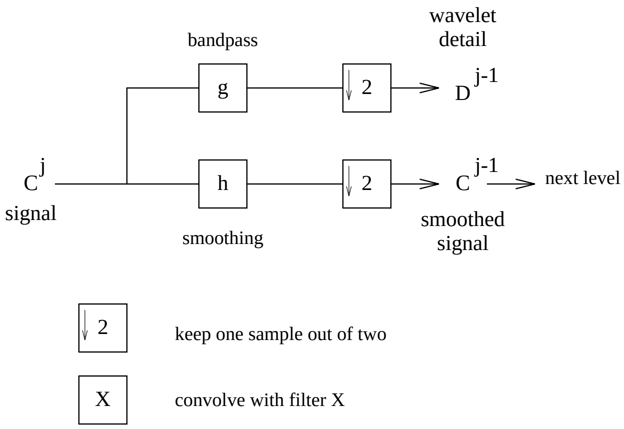

Figure 13: Decomposition of discrete signal c^j into smoothed signal c^{j-1} and wavelet coefficients d^{j-1} by use of conjugate filters h and g (after Mallat (1989b)).

and

$$l_D[j-1] = \lfloor \tfrac{1}{2}(l_C[j] + N_h - 2) \rfloor, \tag{19}$$

where f_C, l_C are the first/last indices for the data c at level j (equations (14) and (15), and f_D and l_D are the corresponding indices for the detail d and are also built into `first.last`. Formulae (10) and (17) are the actual formulae built into the `wd` function. The decomposition process is illustrated in Figure 13.

7.2 The result of the DWT

At the end of the algorithm we obtain

$$DWT\{c^M\} = \{c_0^0, d_0^0, d_0^1, d_1^1, \ldots, d_0^{M-1}, \ldots, d_{\frac{N}{2}-1}^{M-1}\} \tag{20}$$

Most of the coefficients are detail, d, although there is c_0^0 which was the last smoothed data to be produced and is a weighted total of all the data (application of many smoothing filters). If the decomposition was produced using the Haar basis then this c_0^0 is exactly the sample mean multiplied by the square root of the number of original data points. The collection of coefficients in (20), plus *all* the levels of smoothed data constitute the *wavelet decomposition object*. The transform that we actually compute is really closer to the expansion:

$$f(x) = \sum_k c_{0k}\phi_{0k}(x) + \sum_{j\geq 0}\sum_k d_{jk}\psi_{jk}(x), \tag{21}$$

rather than the expansion in (4). The functions ϕ_{0k} are simply the integer translates of the scaling function ϕ mentioned in Section 7. The set $\{\phi_{0k}, \psi_{jk}\}_{(j,k)\in\mathbf{Z}^2}$

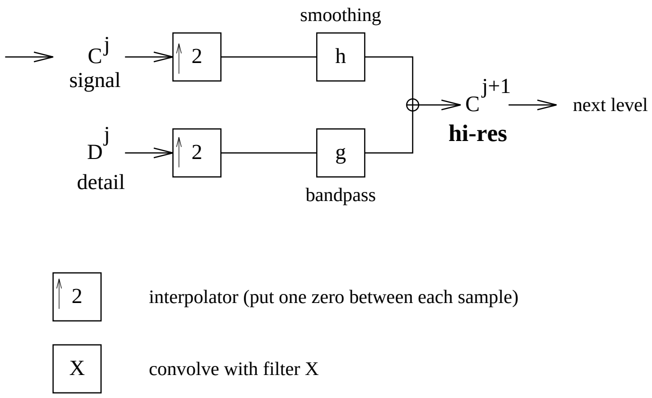

Figure 14: Reconstruction step. Producing higher resolution signal c^{j+1} from smoothed signal c^j and d^j from filters h and g(after Mallat (1989b)).

also forms a basis for $L^2(\Re)$ for appropriate choice of ϕ and ψ.

7.3 Computing the IDWT

To reconstruct we begin with the lowest resolution coefficients and work up towards the full resolution (or whatever level we desire). So, given the wavelet decomposition in (20) we would begin the reconstruction using c_0^0 and d_0^0 to produce c_0^1 and c_1^1. We would then use these c and the d at the same resolution level to produce the c at the next resolution level. The general idea of a step in this reconstruction is shown in Figure 14.

7.4 Level j to level $j+1$

The formula for obtaining the c from the previous level is

$$c_n^{j+1} = \sum_{k\in\mathbf{Z}} h(n-2k)c_k^j + \sum_{k\in\mathbf{Z}} g(n-2k)d_k^j,$$

where h and g are as before. Since there are only finitely many h we can place limits on the indices of summation, furthermore we may rewrite g in terms of h because of (8) and obtain

$$c_n^{j+1} = \sum_{2k\geq n+1-N_h}^{2k\leq n} h(n-2k)c_k^j + \sum_{2k\geq n-1}^{2k\leq N_h+n-2} h(1+2k-n)d_k^j.$$

We know which n of the c to compute since these were worked out in the decomposition part.

7.5 The DWT of images

Mallat (1989b) also gives details of how to compute the DWT and its inverse on an image. We give the algorithm details in Figure 15 for the 2-dimensional decomposition and Figure 16 for the reconstruction. The transform described in Mallat (1989b) is based on a separable multiresolution approximation (see Mallat (1989a) or Daubechies (1992)) where the two-dimensional scale function written as

$$\Phi(x, y) = \phi(x)\phi(y)$$

where $\phi(x)$ is a one-dimensional scale function from a univariate multiresolution approximation. This time there are three "mother" wavelets, not one:

$$\begin{array}{ll} \Psi^1(x, y) = \phi(x)\psi(y), & \Psi^2(x, y) = \psi(x)\phi(y) \\ \Psi^3(x, y) = \psi(x)\psi(y) & \end{array}$$

instead of just one for the univariate case. If we begin with a n^2 image at level j then we obtain four sets of information. First, we obtain a $\left(\frac{n}{2}\right)^2$ image representing the smoothed data at level $j-1$; the other three sets are wavelet coefficients corresponding to the $\left\{\Psi^1, \Psi^2, \Psi^3\right\}$ wavelet functions, and there will be $\left(\frac{n}{2}\right)^2$ coefficients in these "images" as well. Due to the separable nature of the decomposition each of the detail in the wavelet images will be oriented horizontally, vertically and diagonally. This can also be seen from the diagram depicting the decomposition of an image in Figure 15.

8 Acknowledgements

We would like to thank Iain Johnstone for assistance and encouragement with the software and this paper. We would also like to thank Howard Grubb, Emma McCoy, Peter Clifford and Martin Maechler for suggesting improvements and making contributions to the `wavethresh` package. We also thank the referees for some very constructive and helpful comments.

This work was performed with the support of a grant under the Complex Stochastic Systems Initiative of the UK Science and Engineering Research Council. Some of the work was performed whilst both of us were at the University of Bath.

A How to obtain and install the wavelet software

A.1 Obtaining the software

The `wavethresh` package is available, along with other statistical software, from the StatLib archive. StatLib is a statistical software archive where the software

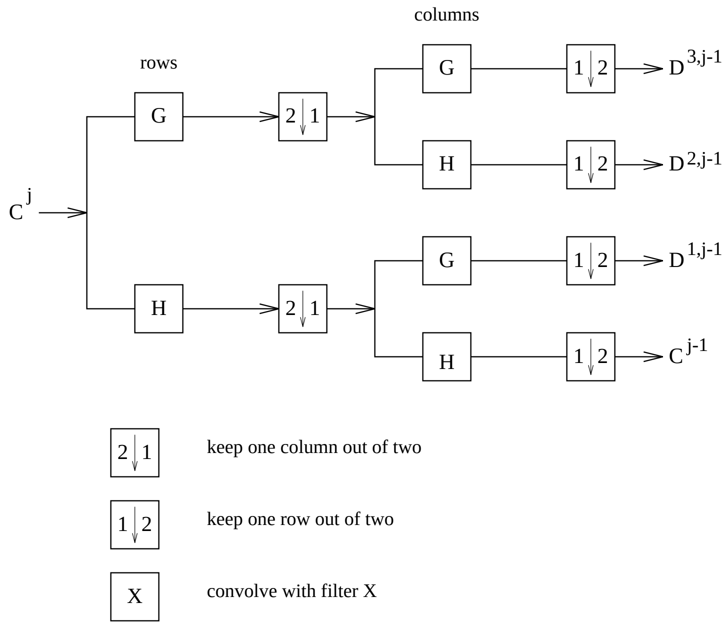

Figure 15: Decomposition of image. The algorithm is based on one-dimensional convolutions along rows and columns of the image (after Mallat (1989b)).

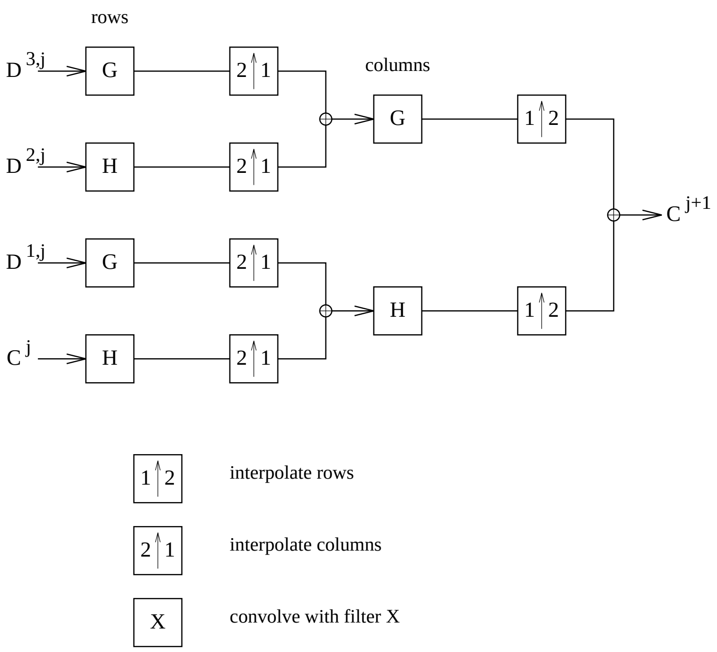

Figure 16: Reconstruction of image (after Mallat (1989b)).

is freely obtainable via email and anonymous FTP.

Using anonymous FTP

To obtain the `wavethresh` software via anonymous FTP type

```
% ftp lib.stat.cmu.edu
```

Then respond to the login id with `statlib` and use your full email address as the password. Then type

```
ftp> cd S
```

to enter the S software directory and then type

```
ftp> get wavethresh
```

and you should then see messages informing you that the software is being retrieved. The StatLib archive is reproduced at other locations, for example UK users will find a StatLib *mirror* at `hensa.unix.ac.uk`. It probably makes sense to try to use an archive that is closer to you.

Using electronic mail

StatLib's electronic mail address is `statlib@lib.stat.cmu.edu`. You can obtain StatLib's index by sending the message `send index`. The `wavethresh` software can be obtained by sending the message `send wavethresh from S`.

A.2 Unpacking the software

The software is distributed as a shar archive file that we will call `wavethresh`. You can unpack the software anywhere you like, but it is a good idea to create a particular subdirectory for it. A good name for a subdirectory is `WAVELET`, so `cd` to somewhere and type:

```
% mkdir WAVELET
```

Then copy the distribution file (`wavethresh`) to this directory and then `cd` to it. To unpack the distribution type:

```
%  sh wavethresh
```

You can now delete the distribution file `wavethresh` if you like. The archive is unpacked into a subdirectory named `DISTRIB2.2`. If you `cd` into this directory and take a look at the files there you should see the following files (although your `ls` function may list them differently).

```
total 88
   2 Copyright                         4 StoIRS.c
   7 ImageDecomposeStep.c              2 conbar.c
   2 Makefile                          4 convolve.c
   2 README                            3 wavedecomp.c
   3 Sconvolve.c                       1 wavelet.h
  53 Source-0.S                        3 waverecons.c
   2 StoIDS.c
```

Most of these files are the C code to perform the transforms, and will have to be compiled. The `Source-0.S` file contains all the S functions in ASCII format. In addition there is a subdirectory `.Data` which `ls -a` will reveal.

Setting up

Before you can use any of the software you must perform some initialisation tasks. The `Makefile` is set up for using SPlus; you may wish to edit and change this to S. You might also need to modify other aspects of the Makefile, but it should be correct for most UNIX systems. Assuming you are still in the `DISTRIB2.2` directory make the software by typing:

```
% make all
```

This sets up the S functions and constructs the C object code. Type:

```
% make cleanobj
```

if you want to clean up.

If you use raw S then `dyn.load` may not work for you, in which case you need to replace the call to `dyn.load` in the `maybe.load` function by `dyn.load2`. At present, the Makefile contains the relevant modifications for a DEC machine running Ultrix V4.2A, although this may give you a guide for running the software on other machines. We are not expert in dynamic loading for all machines so *please* consult local gurus first.

References

Chui, C.K. (1992). *An Introduction to Wavelets*. London: Academic Press.

Daubechies, I. (1988). Orthonormal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, **41**(7), 909–996.

Daubechies, I. (1992). *Ten lectures on Wavelets*. Society for Industrial and Applied Mathematics.

DeVore, R. A., Jawerth, B., & Lucier, B. J. (1992). Image compression through wavelet transform coding. *IEEE Transactions on Information Theory*, **38**(2), 719–746.

Donoho, D.L., & Johnstone, I.M. (1993). Ideal spatial adaptation by wavelet shrinkage. *Biometrika (publication subject to revision).*

Mallat, S. G. (1989a). Multiresolution approximations and wavelet orthonormal bases of $L^2(R)$. *Transactions of the American Mathematical Society*, **315**(1), 69–87.

Mallat, S. G. (1989b). A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **11**(7), 674–693.

Mintzer, F. (1982). On half-band, third-band and Nth band FIR filters and their design. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **30**, 734–738.

Mintzer, F. (1985). Filters for distortion-free two-band multirate filter banks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **33**, 626–630.

Nason, G P. (1993). *The* `wavethresh` *package; wavelet transform and thresholding software for S.* Available from the StatLib archive.

Smith, M.J.T., & Barnwell, T.P. (1986). Exact reconstruction techniques for tree-structured subband coders. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **34**, 434–441.

Smith, M.J.T., & Eddins, S.L. (1990). Analysis/synthesis techniques for subband image coding. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **38**, 1446–1456.

Strang, G. (1993). Wavelet transforms versus Fourier transforms. *Bulletin (New Series) of the American Mathematical Society*, **28**(2), 288–305.

Vaidyanathan, P. P. (1990). Multirate digital filters, filter banks, polyphase networks, and applications: a tutorial. *Proceedings of the IEEE*, **78**(1), 56–93.

Vetterli, M. (1984). Multi-dimensional sub-band coding: some theory and algorithms. *Signal Processing*, **6**, 97–112.

Vetterli, M., & Herley, C. (1992). Wavelets and filter banks: theory and design. *IEEE Transactions on Signal Processing*, **40**, 2207–2232.