Giunta, G. and Murli, A. 1987, *ACM Transactions on Mathematical Software*, vol. 13, pp. 97–107. [4]

Lyness, J.N. 1987, in *Numerical Integration*, P. Keast and G. Fairweather, eds. (Dordrecht: Reidel). [5]

Pantis, G. 1975, *Journal of Computational Physics*, vol. 17, pp. 229–233. [6]

Blakemore, M., Evans, G.A., and Hyslop, J. 1976, *Journal of Computational Physics*, vol. 22, pp. 352–376. [7]

Lyness, J.N., and Kaper, T.J. 1987, *SIAM Journal on Scientific and Statistical Computing*, vol. 8, pp. 1005–1011. [8]

Thakkar, A.J., and Smith, V.H. 1975, *Computer Physics Communications*, vol. 10, pp. 73–79. [9]

13.10 Wavelet Transforms

Like the fast Fourier transform (FFT), the discrete wavelet transform (DWT) is a fast, linear operation that operates on a data vector whose length is an integer power of two, transforming it into a numerically different vector of the same length. Also like the FFT, the wavelet transform is invertible and in fact orthogonal — the inverse transform, when viewed as a big matrix, is simply the transpose of the transform. Both FFT and DWT, therefore, can be viewed as a rotation in function space, from the input space (or time) domain, where the basis functions are the unit vectors $\mathbf{e}_i$, or Dirac delta functions in the continuum limit, to a different domain. For the FFT, this new domain has basis functions that are the familiar sines and cosines. In the wavelet domain, the basis functions are somewhat more complicated and have the fanciful names "mother functions" and "wavelets."

Of course there are an infinity of possible bases for function space, almost all of them uninteresting! What makes the wavelet basis interesting is that, *unlike* sines and cosines, individual wavelet functions are quite localized in space; simultaneously, *like* sines and cosines, individual wavelet functions are quite localized in frequency or (more precisely) characteristic scale. As we will see below, the particular kind of dual localization achieved by wavelets renders large classes of functions and operators sparse, or sparse to some high accuracy, when transformed into the wavelet domain. Analogously with the Fourier domain, where a class of computations, like convolutions, become computationally fast, there is a large class of computations — those that can take advantage of sparsity — that become computationally fast in the wavelet domain [1].

Unlike sines and cosines, which define a unique Fourier transform, there is not one single unique set of wavelets; in fact, there are infinitely many possible sets. Roughly, the different sets of wavelets make different trade-offs between how compactly they are localized in space and how smooth they are. (There are further fine distinctions.)

Daubechies Wavelet Filter Coefficients

A particular set of wavelets is specified by a particular set of numbers, called *wavelet filter coefficients*. Here, we will largely restrict ourselves to wavelet filters in a class discovered by Daubechies [2]. This class includes members ranging from highly localized to highly smooth. The simplest (and most localized) member, often called *DAUB4*, has only four coefficients, $c_0, \dots, c_3$. For the moment we specialize to this case for ease of notation.

Consider the following transformation matrix acting on a column vector of data to its right:

$$
\begin{bmatrix}
c_0 & c_1 & c_2 & c_3 & & & & & & & \\
c_3 & -c_2 & c_1 & -c_0 & & & & & & & \\
 & & c_0 & c_1 & c_2 & c_3 & & & & & \\
 & & c_3 & -c_2 & c_1 & -c_0 & & & & & \\
\vdots & \vdots & & & & & \ddots & & & & \\
 & & & & & & & c_0 & c_1 & c_2 & c_3 \\
 & & & & & & & c_3 & -c_2 & c_1 & -c_0 \\
c_2 & c_3 & & & & & & & & c_0 & c_1 \\
c_1 & -c_0 & & & & & & & & c_3 & -c_2
\end{bmatrix}
\tag{13.10.1}
$$

Here blank entries signify zeroes. Note the structure of this matrix. The first row generates one component of the data convolved with the filter coefficients $c_0 \dots, c_3$. Likewise the third, fifth, and other odd rows. If the even rows followed this pattern, offset by one, then the matrix would be a circulant, that is, an ordinary convolution that could be done by FFT methods. (Note how the last two rows wrap around like convolutions with periodic boundary conditions.) Instead of convolving with $c_0, \dots, c_3$, however, the even rows perform a different convolution, with coefficients $c_3, -c_2, c_1, -c_0$. The action of the matrix, overall, is thus to perform two related convolutions, then to decimate each of them by half (throw away half the values), and interleave the remaining halves.

It is useful to think of the filter $c_0, \dots, c_3$ as being a smoothing filter, call it H, something like a moving average of four points. Then, because of the minus signs, the filter $c_3, -c_2, c_1, -c_0$, call it G, is *not* a smoothing filter. (In signal processing contexts, H and G are called *quadrature mirror filters* [3].) In fact, the c's are chosen so as to make G yield, insofar as possible, a *zero* response to a sufficiently smooth data vector. This is done by requiring the sequence $c_3, -c_2, c_1, -c_0$ to have a certain number of vanishing moments. When this is the case for p moments (starting with the zeroth), a set of wavelets is said to satisfy an "approximation condition of order p." This results in the output of H, decimated by half, accurately representing the data's "smooth" information. The output of G, also decimated, is referred to as the data's "detail" information [4].

For such a characterization to be useful, it must be possible to reconstruct the original data vector of length N from its $N/2$ smooth or s-components and its $N/2$ detail or d-components. That is effected by requiring the matrix (13.10.1) to be orthogonal, so that its inverse is just the transposed matrix

$$
\begin{bmatrix}
c_0 & c_3 & & & \cdots & & & & & c_2 & c_1 \\
c_1 & -c_2 & & & \cdots & & & & & c_3 & -c_0 \\
c_2 & c_1 & c_0 & c_3 & & & & & & & \\
c_3 & -c_0 & c_1 & -c_2 & & & & & & & \\
 & & & & \ddots & & & & & & \\
 & & & & & c_2 & c_1 & c_0 & c_3 & & \\
 & & & & & c_3 & -c_0 & c_1 & -c_2 & & \\
 & & & & & & & c_2 & c_1 & c_0 & c_3 \\
 & & & & & & & c_3 & -c_0 & c_1 & -c_2
\end{bmatrix}
\tag{13.10.2}
$$

One sees immediately that matrix (13.10.2) is inverse to matrix (13.10.1) if and only if these two equations hold,

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1 \\ c_2 c_0 + c_3 c_1 &= 0 \end{aligned} \qquad (13.10.3)$$

If additionally we require the approximation condition of order $p = 2$, then two additional relations are required,

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0 \\ 0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0 \end{aligned} \qquad (13.10.4)$$

Equations (13.10.3) and (13.10.4) are 4 equations for the 4 unknowns $c_0, \ldots, c_3$, first recognized and solved by Daubechies. The unique solution (up to a left-right reversal) is

$$\begin{aligned} c_0 &= (1 + \sqrt{3})/4\sqrt{2} \qquad & c_1 &= (3 + \sqrt{3})/4\sqrt{2} \\ c_2 &= (3 - \sqrt{3})/4\sqrt{2} \qquad & c_3 &= (1 - \sqrt{3})/4\sqrt{2} \end{aligned} \qquad (13.10.5)$$

In fact, DAUB4 is only the most compact of a sequence of wavelet sets: If we had six coefficients instead of four, there would be three orthogonality requirements in equation (13.10.3) (with offsets of zero, two, and four), and we could require the vanishing of $p = 3$ moments in equation (13.10.4). In this case, DAUB6, the solution coefficients can also be expressed in closed form,

$$\begin{aligned} c_0 &= (1 + \sqrt{10} + \sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \qquad & c_1 &= (5 + \sqrt{10} + 3\sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \\ c_2 &= (10 - 2\sqrt{10} + 2\sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \qquad & c_3 &= (10 - 2\sqrt{10} - 2\sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \\ c_4 &= (5 + \sqrt{10} - 3\sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \qquad & c_5 &= (1 + \sqrt{10} - \sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \end{aligned} \qquad (13.10.6)$$

For higher p, up to 10, Daubechies [2] has tabulated the coefficients numerically. The number of coefficients increases by two each time p is increased by one.

Discrete Wavelet Transform

We have not yet defined the discrete wavelet transform (DWT), but we are almost there: The DWT consists of applying a wavelet coefficient matrix like (13.10.1) *hierarchically*, first to the full data vector of length N, then to the "smooth" vector of length $N/2$, then to the "smooth-smooth" vector of length $N/4$, and so on until only a trivial number of "smooth-. . .-smooth" components (usually 2) remain. The procedure is sometimes called a *pyramidal algorithm* [4], for obvious reasons. The output of the DWT consists of these remaining components and all the "detail" components that were accumulated along the way. A diagram should make the procedure clear:

$$
\begin{bmatrix} y_1\\ y_2\\ y_3\\ y_4\\ y_5\\ y_6\\ y_7\\ y_8\\ y_9\\ y_{10}\\ y_{11}\\ y_{12}\\ y_{13}\\ y_{14}\\ y_{15}\\ y_{16} \end{bmatrix}
\xrightarrow{13.10.1}
\begin{bmatrix} s_1\\ d_1\\ s_2\\ d_2\\ s_3\\ d_3\\ s_4\\ d_4\\ s_5\\ d_5\\ s_6\\ d_6\\ s_7\\ d_7\\ s_8\\ d_8 \end{bmatrix}
\xrightarrow{\text{permute}}
\begin{bmatrix} s_1\\ s_2\\ s_3\\ s_4\\ s_5\\ s_6\\ s_7\\ s_8\\ \hline d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{13.10.1}
\begin{bmatrix} S_1\\ D_1\\ S_2\\ D_2\\ S_3\\ D_3\\ S_4\\ D_4\\ \hline d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{\text{permute}}
\begin{bmatrix} S_1\\ S_2\\ S_3\\ S_4\\ \hline D_1\\ D_2\\ D_3\\ D_4\\ \hline d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\xrightarrow{\text{etc.}}
\begin{bmatrix} \mathcal{S}_1\\ \mathcal{S}_2\\ \hline \mathcal{D}_1\\ \mathcal{D}_2\\ \hline D_1\\ D_2\\ D_3\\ D_4\\ \hline d_1\\ d_2\\ d_3\\ d_4\\ d_5\\ d_6\\ d_7\\ d_8 \end{bmatrix}
\tag{13.10.7}
$$

If the length of the data vector were a higher power of two, there would be more stages of applying (13.10.1) (or any other wavelet coefficients) and permuting. The endpoint will always be a vector with two $\mathcal{S}$'s and a hierarchy of $\mathcal{D}$'s, D's, d's, etc. Notice that once d's are generated, they simply propagate through to all subsequent stages.

A value d_i of any level is termed a "wavelet coefficient" of the original data vector; the final values $\mathcal{S}_1$, $\mathcal{S}_2$ should strictly be called "mother-function coefficients," although the term "wavelet coefficients" is often used loosely for both d's and final $\mathcal{S}$'s. Since the full procedure is a composition of orthogonal linear operations, the whole DWT is itself an orthogonal linear operator.

To invert the DWT, one simply reverses the procedure, starting with the smallest level of the hierarchy and working (in equation 13.10.7) from right to left. The inverse matrix (13.10.2) is of course used instead of the matrix (13.10.1).

As already noted, the matrices (13.10.1) and (13.10.2) embody periodic ("wrap-around") boundary conditions on the data vector. One normally accepts this as a minor inconvenience: the last few wavelet coefficients at each level of the hierarchy are affected by data from both ends of the data vector. By circularly shifting the matrix (13.10.1) $N/2$ columns to the left, one can symmetrize the wrap-around; but this does not eliminate it. It is in fact possible to eliminate the wrap-around completely by altering the coefficients in the first and last N rows of (13.10.1), giving an orthogonal matrix that is purely band-diagonal [5]. This variant, beyond our scope here, is useful when, e.g., the data varies by many orders of magnitude from one end of the data vector to the other.

Here is a routine, `wt1`, that performs the pyramidal algorithm (or its inverse if `isign` is negative) on some data vector `a(1:n)`. Successive applications of the wavelet filter, and accompanying permutations, are done by an assumed routine `wtstep`, which must be provided. (We give examples of several different `wtstep` routines just below.)

```
      SUBROUTINE wt1(a,n,isign,wtstep)
      INTEGER isign,n
      REAL a(n)
      EXTERNAL wtstep
C     USES wtstep
```

One-dimensional discrete wavelet transform. This routine implements the pyramid algorithm, replacing `a(1:n)` by its wavelet transform (for `isign=1`), or performing the inverse operation (for `isign=-1`). Note that `n` MUST be an integer power of 2. The subroutine

```
    wtstep, whose actual name must be supplied in calling this routine, is the underlying
    wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.
      INTEGER nn
      if (n.lt.4) return
      if (isign.ge.0) then                 Wavelet transform.
          nn=n                             Start at largest hierarchy,
1         if (nn.ge.4) then
              call wtstep(a,nn,isign)      and work towards smallest.
              nn=nn/2
          goto 1
          endif
      else                                 Inverse wavelet transform.
          nn=4                             Start at smallest hierarchy,
2         if (nn.le.n) then
              call wtstep(a,nn,isign)
              nn=nn*2                      and work towards largest.
          goto 2
          endif
      endif
      return
      END
```

Here, as a specific instance of wtstep, is a routine for the DAUB4 wavelets:

```
      SUBROUTINE daub4(a,n,isign)
      INTEGER n,isign,NMAX           NMAX is the maximum allowed value of n.
      REAL a(n),C3,C2,C1,C0
      PARAMETER (C0=0.4829629131445341,C1=0.8365163037378079,
     *     C2=0.2241438680420134,C3=-0.1294095225512604,NMAX=1024)
    Applies the Daubechies 4-coefficient wavelet filter to data vector a(1:n) (for isign=1) or
    applies its transpose (for isign=-1). Used hierarchically by routines wt1 and wtn.
      REAL wksp(NMAX)
      INTEGER nh,nh1,i,j
      if(n.lt.4)return
      if(n.gt.NMAX) pause 'wksp too small in daub4'
      nh=n/2
      nh1=nh+1
      if (isign.ge.0) then                           Apply filter.
          i=1
          do 11 j=1,n-3,2
              wksp(i)=C0*a(j)+C1*a(j+1)+C2*a(j+2)+C3*a(j+3)
              wksp(i+nh)=C3*a(j)-C2*a(j+1)+C1*a(j+2)-C0*a(j+3)
              i=i+1
          enddo 11
          wksp(i)=C0*a(n-1)+C1*a(n)+C2*a(1)+C3*a(2)
          wksp(i+nh)=C3*a(n-1)-C2*a(n)+C1*a(1)-C0*a(2)
      else                                           Apply transpose filter.
          wksp(1)=C2*a(nh)+C1*a(n)+C0*a(1)+C3*a(nh1)
          wksp(2)=C3*a(nh)-C0*a(n)+C1*a(1)-C2*a(nh1)
          j=3
          do 12 i=1,nh-1
              wksp(j)=C2*a(i)+C1*a(i+nh)+C0*a(i+1)+C3*a(i+nh1)
              wksp(j+1)=C3*a(i)-C0*a(i+nh)+C1*a(i+1)-C2*a(i+nh1)
              j=j+2
          enddo 12
      endif
      do 13 i=1,n
          a(i)=wksp(i)
      enddo 13
      return
      END
```

For larger sets of wavelet coefficients, the wrap-around of the last rows or columns is a programming inconvenience. An efficient implementation would handle the wrap-arounds as special cases, outside of the main loop. Here, we will content ourselves with a more general scheme involving some extra arithmetic at run time. The following routine sets up any particular wavelet coefficients whose values you happen to know.

```
      SUBROUTINE pwtset(n)
      INTEGER n,NCMAX,ncof,ioff,joff
      PARAMETER (NCMAX=50)          Maximum number of wavelet coefficients passed to pwt.
      REAL cc(NCMAX),cr(NCMAX)
      COMMON /pwtcom/ cc,cr,ncof,ioff,joff
         Initializing routine for pwt, here implementing the Daubechies wavelet filters with 4, 12,
         and 20 coefficients, as selected by the input value n. Further wavelet filters can be included
         in the obvious manner. This routine must be called (once) before the first use of pwt. (For
         the case n=4, the specific routine daub4 is considerably faster than pwt.)
      INTEGER k
      REAL sig,c4(4),c12(12),c20(20)
      SAVE c4,c12,c20,/pwtcom/
      DATA c4/0.4829629131445341, 0.8365163037378079,
     *     0.2241438680420134,-0.1294095225512604/
      DATA c12 /.111540743350, .494623890398, .751133908021,
     *      .315250351709,-.226264693965,-.129766867567,
     *      .097501605587, .027522865530,-.031582039318,
     *      .000553842201, .004777257511,-.001077301085/
      DATA c20 /.026670057901, .188176800078, .527201188932,
     *      .688459039454, .281172343661,-.249846424327,
     *     -.195946274377, .127369340336, .093057364604,
     *     -.071394147166,-.029457536822, .033212674059,
     *      .003606553567,-.010733175483, .001395351747,
     *      .001992405295,-.000685856695,-.000116466855,
     *      .000093588670,-.000013264203 /
      ncof=n
      sig=-1.
      do 11 k=1,n
          if(n.eq.4)then
              cc(k)=c4(k)
          else if(n.eq.12)then
              cc(k)=c12(k)
          else if(n.eq.20)then
              cc(k)=c20(k)
          else
              pause 'unimplemented value n in pwtset'
          endif
          cr(ncof+1-k)=sig*cc(k)
          sig=-sig
      enddo 11
      ioff=-n/2                 These values center the "support" of the wavelets at each level.
      joff=-n/2                    Alternatively, the "peaks" of the wavelets can be approxi-
      return                       mately centered by the choices ioff=-2 and joff=-n+2.
      END
```

Once pwtset has been called, the following routine can be used as a specific instance of wtstep.

```
      SUBROUTINE pwt(a,n,isign)
      INTEGER isign,n,NMAX,NCMAX,ncof,ioff,joff
      PARAMETER (NMAX=2048,NCMAX=50)
      REAL a(n),wksp(NMAX),cc(NCMAX),cr(NCMAX)
      COMMON /pwtcom/ cc,cr,ncof,ioff,joff
```

```
    Partial wavelet transform: applies an arbitrary wavelet filter to data vector a(1:n) (for
    isign=1) or applies its transpose (for isign=-1). Used hierarchically by routines wt1
    and wtn. The actual filter is determined by a preceding (and required) call to pwtset,
    which initializes the common block pwtcom.
INTEGER i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod
REAL ai,ai1
if (n.lt.4) return
nmod=ncof*n                      A positive constant equal to zero mod n.
n1=n-1                           Mask of all bits, since n a power of 2.
nh=n/2
do 11 j=1,n
    wksp(j)=0.
enddo 11
if (isign.ge.0) then             Apply filter.
    ii=1
    do 13 i=1,n,2
        ni=i+nmod+ioff           Pointer to be incremented and wrapped-around.
        nj=i+nmod+joff
        do 12 k=1,ncof
            jf=iand(n1,ni+k)     We use bitwise and to wrap-around the pointers.
            jr=iand(n1,nj+k)
            wksp(ii)=wksp(ii)+cc(k)*a(jf+1)
            wksp(ii+nh)=wksp(ii+nh)+cr(k)*a(jr+1)
        enddo 12
        ii=ii+1
    enddo 13
else                             Apply transpose filter.
    ii=1
    do 15 i=1,n,2
        ai=a(ii)
        ai1=a(ii+nh)
        ni=i+nmod+ioff           See comments above.
        nj=i+nmod+joff
        do 14 k=1,ncof
            jf=iand(n1,ni+k)+1
            jr=iand(n1,nj+k)+1
            wksp(jf)=wksp(jf)+cc(k)*ai
            wksp(jr)=wksp(jr)+cr(k)*ai1
        enddo 14
        ii=ii+1
    enddo 15
endif
do 16 j=1,n                      Copy the results back from workspace.
    a(j)=wksp(j)
enddo 16
return
END
```

What Do Wavelets Look Like?

We are now in a position actually to see some wavelets. To do so, we simply run unit vectors through any of the above discrete wavelet transforms, with `isign` negative so that the inverse transform is performed. Figure 13.10.1 shows the DAUB4 wavelet that is the inverse DWT of a unit vector in the 5th component of a vector of length 1024, and also the DAUB20 wavelet that is the inverse of the 22nd component. (One needs to go to a later hierarchical level for DAUB20, to avoid a wavelet with a wrapped-around tail.) Other unit vectors would give wavelets with the same shapes, but different positions and scales.

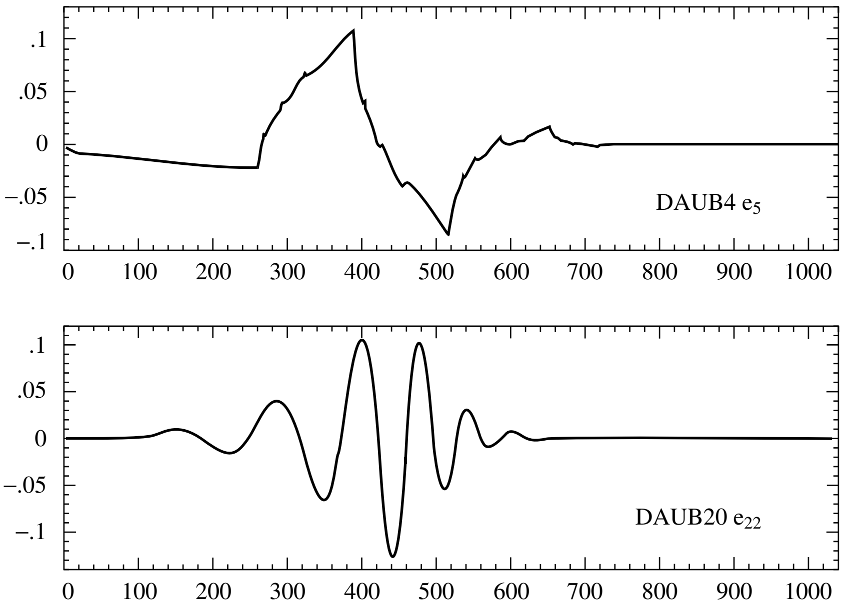

Figure 13.10.1. Wavelet functions, that is, single basis functions from the wavelet families DAUB4 and DAUB20. A complete, orthonormal wavelet basis consists of scalings and translations of either one of these functions. DAUB4 has an infinite number of cusps; DAUB20 would show similar behavior in a higher derivative.

One sees that both DAUB4 and DAUB20 have wavelets that are continuous. DAUB20 wavelets also have higher continuous derivatives. DAUB4 has the peculiar property that its derivative exists only *almost* everywhere. Examples of where it fails to exist are the points $p/2^n$, where p and n are integers; at such points, DAUB4 is left differentiable, but not right differentiable! This kind of discontinuity — at least in some derivative — is a necessary feature of wavelets with compact support, like the Daubechies series. For every increase in the number of wavelet coefficients by two, the Daubechies wavelets gain about *half* a derivative of continuity. (But not exactly half; the actual orders of regularity are irrational numbers!)

Note that the fact that wavelets are not smooth does not prevent their having exact representations for some smooth functions, as demanded by their approximation order p. The continuity of a wavelet is not the same as the continuity of functions that a set of wavelets can represent. For example, DAUB4 can represent (piecewise) linear functions of arbitrary slope: in the correct linear combinations, the cusps all cancel out. Every increase of two in the number of coefficients allows one higher order of polynomial to be exactly represented.

Figure 13.10.2 shows the result of performing the inverse DWT on the input vector $\mathbf{e}_{10} + \mathbf{e}_{58}$, again for the two different particular wavelets. Since 10 lies early in the hierarchical range of $9 - 16$, that wavelet lies on the left side of the picture. Since 58 lies in a later (smaller-scale) hierarchy, it is a narrower wavelet; in the range of 33–64 it is towards the end, so it lies on the right side of the picture. Note that smaller-scale wavelets are taller, so as to have the same squared integral.

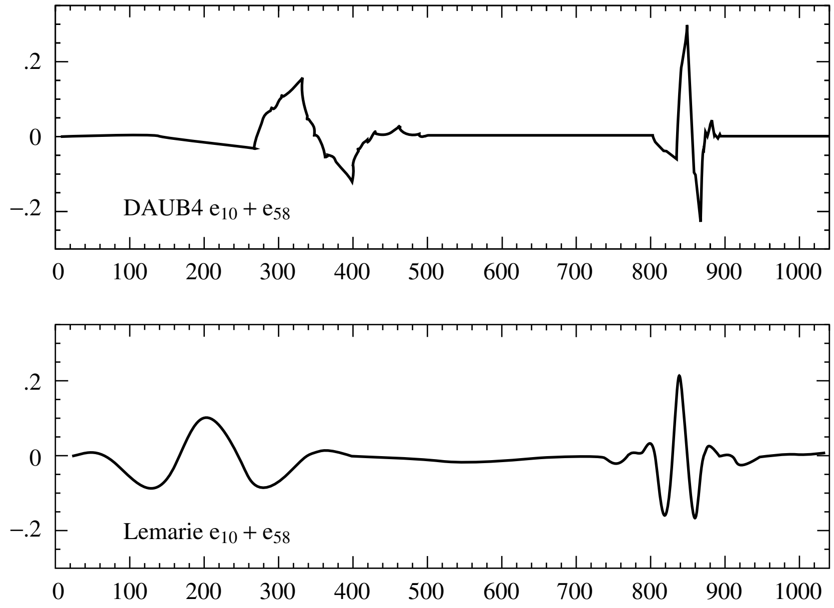

Figure 13.10.2. More wavelets, here generated from the sum of two unit vectors, $\mathbf{e}_{10} + \mathbf{e}_{58}$, which are in different hierarchical levels of scale, and also at different spatial positions. DAUB4 wavelets (a) are defined by a filter in coordinate space (equation 13.10.5), while Lemarie wavelets (b) are defined by a filter most easily written in Fourier space (equation 13.10.14).

Wavelet Filters in the Fourier Domain

The Fourier transform of a set of filter coefficients c_j is given by

$$H(\omega) = \sum_j c_j e^{ij\omega} \tag{13.10.8}$$

Here H is a function periodic in 2π, and it has the same meaning as before: It is the wavelet filter, now written in the Fourier domain. A very useful fact is that the orthogonality conditions for the c's (e.g., equation 13.10.3 above) collapse to two simple relations in the Fourier domain,

$$\frac{1}{2}|H(0)|^2 = 1 \tag{13.10.9}$$

and

$$\frac{1}{2}\left[|H(\omega)|^2 + |H(\omega+\pi)|^2\right] = 1 \tag{13.10.10}$$

Likewise the approximation condition of order p (e.g., equation 13.10.4 above) has a simple formulation, requiring that $H(\omega)$ have a pth order zero at $\omega = \pi$, or (equivalently)

$$H^{(m)}(\pi) = 0 \qquad m = 0, 1, \ldots, p-1 \tag{13.10.11}$$

It is thus relatively straightforward to invent wavelet sets in the Fourier domain. You simply invent a function $H(\omega)$ satisfying equations (13.10.9)–(13.10.11). To find the actual c_j's applicable to a data (or s-component) vector of length N, and with periodic wrap-around as in matrices (13.10.1) and (13.10.2), you invert equation (13.10.8) by the discrete Fourier transform

$$c_j = \frac{1}{N} \sum_{k=0}^{N-1} H(\frac{2\pi k}{N}) e^{-2\pi i j k/N} \tag{13.10.12}$$

The quadrature mirror filter G (reversed c_j's with alternating signs), incidentally, has the Fourier representation

$$G(\omega) = e^{-i\omega} H^*(\omega + \pi) \tag{13.10.13}$$

where asterisk denotes complex conjugation.

In general the above procedure will *not* produce wavelet filters with compact support. In other words, all N of the c_j's, $j = 0, \ldots, N-1$ will in general be nonzero (though they may be rapidly decreasing in magnitude). The Daubechies wavelets, or other wavelets with compact support, are specially chosen so that $H(\omega)$ is a trigonometric polynomial with only a small number of Fourier components, guaranteeing that there will be only a small number of nonzero c_j's.

On the other hand, there is sometimes no particular reason to demand compact support. Giving it up in fact allows the ready construction of relatively smoother wavelets (higher values of p). Even without compact support, the convolutions implicit in the matrix (13.10.1) can be done efficiently by FFT methods.

Lemarie's wavelet (see [4]) has $p = 4$, does not have compact support, and is defined by the choice of $H(\omega)$,

$$H(\omega) = \left[2(1-u)^4 \frac{315 - 420u + 126u^2 - 4u^3}{315 - 420v + 126v^2 - 4v^3}\right]^{1/2} \tag{13.10.14}$$

where

$$u \equiv \sin^2 \frac{\omega}{2} \qquad v \equiv \sin^2 \omega \tag{13.10.15}$$

It is beyond our scope to explain where equation (13.10.14) comes from. An informal description is that the quadrature mirror filter $G(\omega)$ deriving from equation (13.10.14) has the property that it gives identically zero when applied to any function whose odd-numbered samples are equal to the cubic spline interpolation of its even-numbered samples. Since this class of functions includes many very smooth members, it follows that $H(\omega)$ does a good job of truly selecting a function's smooth information content. Sample Lemarie wavelets are shown in Figure 13.10.2.

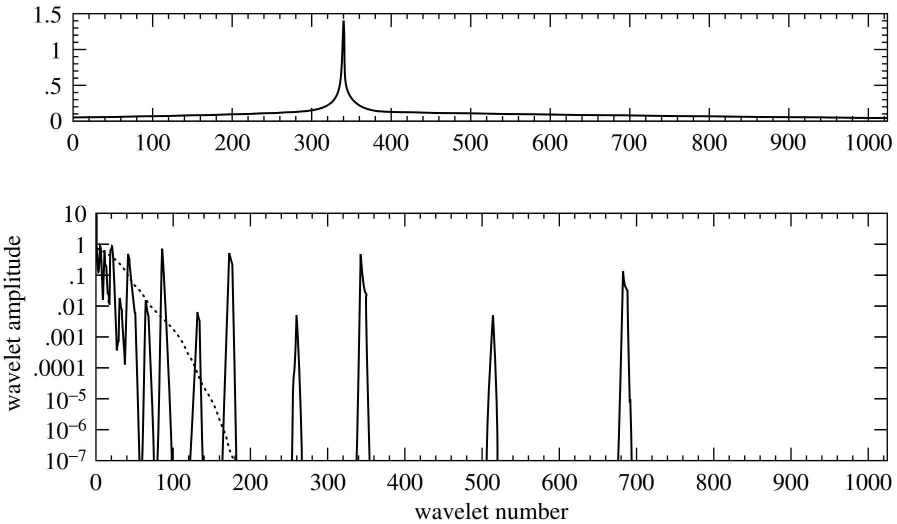

Figure 13.10.3. (a) Arbitrary test function, with cusp, sampled on a vector of length 1024. (b) Absolute value of the 1024 wavelet coefficients produced by the discrete wavelet transform of (a). Note log scale. The dotted curve plots the same amplitudes when sorted by decreasing size. One sees that only 130 out of 1024 coefficients are larger than 10^{-4} (or larger than about 10^{-5} times the largest coefficient, whose value is ~ 10).

Truncated Wavelet Approximations

Most of the usefulness of wavelets rests on the fact that wavelet transforms can usefully be severely truncated, that is, turned into sparse expansions. The case of Fourier transforms is different: FFTs are ordinarily used without truncation, to compute fast convolutions, for example. This works because the convolution operator is particularly simple in the Fourier basis. There are not, however, any standard mathematical operations that are especially simple in the wavelet basis.

To see how truncation works, consider the simple example shown in Figure 13.10.3. The upper panel shows an arbitrarily chosen test function, smooth except for a square-root cusp, sampled onto a vector of length 2^{10}. The bottom panel (solid curve) shows, on a log scale, the absolute value of the vector's components after it has been run through the DAUB4 discrete wavelet transform. One notes, from right to left, the different levels of hierarchy, 513–1024, 257–512, 129–256, etc. Within each level, the wavelet coefficients are non-negligible only very near the location of the cusp, or very near the left and right boundaries of the hierarchical range (edge effects).

The dotted curve in the lower panel of Figure 13.10.3 plots the same amplitudes as the solid curve, but sorted into decreasing order of size. One can read off, for example, that the 130th largest wavelet coefficient has an amplitude less than 10^{-5} of the largest coefficient, whose magnitude is ~ 10 (power or square integral ratio less than 10^{-10}). Thus, the example function can be represented quite accurately by only 130, rather than 1024, coefficients — the remaining ones being set to zero. Note that this kind of truncation makes the vector sparse, but not shorter than 1024. It is very important that vectors in wavelet space be truncated according to the *amplitude* of the components, not their position in the vector. Keeping the first 256 components of the vector (all levels of the hierarchy except the last two) would give an extremely poor, and jagged, approximation to the function. When you compress a function with wavelets, you have to record both the values *and the positions* of the nonzero coefficients.

Generally, compact (and therefore unsmooth) wavelets are better for lower accuracy approximation and for functions with discontinuities (like edges), while smooth (and therefore noncompact) wavelets are better for achieving high numerical accuracy. This makes compact wavelets a good choice for image compression, for example, while it makes smooth wavelets best for fast solution of integral equations.

Wavelet Transform in Multidimensions

A wavelet transform of a d-dimensional array is most easily obtained by transforming the array sequentially on its first index (for all values of its other indices), then on its second, and so on. Each transformation corresponds to multiplication by an orthogonal matrix. By matrix associativity, the result is independent of the order in which the indices were transformed. The situation is exactly like that for multidimensional FFTs. A routine for effecting the multidimensional DWT can thus be modeled on a multidimensional FFT routine like `fourn`:

```
      SUBROUTINE wtn(a,nn,ndim,isign,wtstep)
      INTEGER isign,ndim,nn(ndim),NMAX
      REAL a(*)
      EXTERNAL wtstep
      PARAMETER (NMAX=1024)
C     USES wtstep
        Replaces a by its ndim-dimensional discrete wavelet transform, if isign is input as 1. nn
        is an integer array of length ndim, containing the lengths of each dimension (number of real
        values), which MUST all be powers of 2. a is a real array of length equal to the product
        of these lengths, in which the data are stored as in a multidimensional real FORTRAN array.
        If isign is input as −1, a is replaced by its inverse wavelet transform. The subroutine
        wtstep, whose actual name must be supplied in calling this routine, is the underlying
        wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.
      INTEGER i1,i2,i3,idim,k,n,nnew,nprev,nt,ntot
      REAL wksp(NMAX)
      ntot=1
      do 11 idim=1,ndim
        ntot=ntot*nn(idim)
      enddo 11
      nprev=1
      do 16 idim=1,ndim                          Main loop over the dimensions.
        n=nn(idim)
        nnew=n*nprev
        if (n.gt.4) then
          do 15 i2=0,ntot-1,nnew
            do 14 i1=1,nprev
              i3=i1+i2
              do 12 k=1,n                        Copy the relevant row or column or etc. into
                wksp(k)=a(i3)                      workspace.
                i3=i3+nprev
              enddo 12
              if (isign.ge.0) then               Do one-dimensional wavelet transform.
                nt=n
1               if (nt.ge.4) then
                call wtstep(wksp,nt,isign)
                nt=nt/2
                goto 1
                endif
```

```
                else                            Or inverse transform.
                    nt=4
2                   if (nt.le.n) then
                    call wtstep(wksp,nt,isign)
                    nt=nt*2
                    goto 2
                    endif
                endif
                i3=i1+i2
                do 13 k=1,n                     Copy back from workspace.
                    a(i3)=wksp(k)
                    i3=i3+nprev
                enddo 13
            enddo 14
        enddo 15
    endif
    nprev=nnew
enddo 16
return
END
```

Here, as before, wtstep is an individual wavelet step, either daub4 or pwt.

Compression of Images

An immediate application of the multidimensional transform wtn is to image compression. The overall procedure is to take the wavelet transform of a digitized image, and then to "allocate bits" among the wavelet coefficients in some highly nonuniform, optimized, manner. In general, large wavelet coefficients get quantized accurately, while small coefficients are quantized coarsely with only a bit or two — or else are truncated completely. If the resulting quantization levels are still statistically nonuniform, they may then be further compressed by a technique like Huffman coding (§20.4).

While a more detailed description of the "back end" of this process, namely the quantization and coding of the image, is beyond our scope, it is quite straightforward to demonstrate the "front-end" wavelet encoding with a simple truncation: We keep (with full accuracy) all wavelet coefficients larger than some threshold, and we delete (set to zero) all smaller wavelet coefficients. We can then adjust the threshold to vary the fraction of preserved coefficients.

Figure 13.10.4 shows a sequence of images that differ in the number of wavelet coefficients that have been kept. The original picture (a), which is an official IEEE test image, has 256 by 256 pixels with an 8-bit grayscale. The two reproductions following are reconstructed with 23% (b) and 5.5% (c) of the 65536 wavelet coefficients. The latter image illustrates the kind of compromises made by the truncated wavelet representation. High-contrast edges (the model's right cheek and hair highlights, e.g.) are maintained at a relatively high resolution, while low-contrast areas (the model's left eye and cheek, e.g.) are washed out into what amounts to large constant pixels. Figure 13.10.4 (d) is the result of performing the identical procedure with Fourier, instead of wavelet, transforms: The figure is reconstructed from the 5.5% of 65536 real Fourier components having the largest magnitudes. One sees that, since sines and cosines are nonlocal, the resolution is uniformly poor across the picture; also, the deletion of any components produces a mottled "ringing" everywhere. (Practical Fourier image compression schemes therefore break up an

Figure 13.10.4. (a) IEEE test image, 256×256 pixels with 8-bit grayscale. (b) The image is transformed into the wavelet basis; 77% of its wavelet components are set to zero (those of smallest magnitude); it is then reconstructed from the remaining 23%. (c) Same as (b), but 94.5% of the wavelet components are deleted. (d) Same as (c), but the Fourier transform is used instead of the wavelet transform. Wavelet coefficients are better than the Fourier coefficients at preserving relevant details.

image into small blocks of pixels, 16×16, say, and do rather elaborate smoothing across block boundaries when the image is reconstructed.)

Fast Solution of Linear Systems

One of the most interesting, and promising, wavelet applications is linear algebra. The basic idea [1] is to think of an integral operator (that is, a large matrix) as a digital image. Suppose that the operator compresses well under a two-dimensional wavelet transform, i.e., that a large fraction of its wavelet coefficients are so small as to be negligible. Then any linear system involving the operator becomes a sparse system in the wavelet basis. In other words, to solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{13.10.16}$$

we first wavelet-transform the operator **A** and the right-hand side **b** by

$$\widetilde{\mathbf{A}} \equiv \mathbf{W} \cdot \mathbf{A} \cdot \mathbf{W}^T, \qquad \widetilde{\mathbf{b}} \equiv \mathbf{W} \cdot \mathbf{b} \tag{13.10.17}$$

where **W** represents the one-dimensional wavelet transform, then solve

$$\widetilde{\mathbf{A}} \cdot \widetilde{\mathbf{x}} = \widetilde{\mathbf{b}} \tag{13.10.18}$$

and finally transform to the answer by the inverse wavelet transform

$$\mathbf{x} = \mathbf{W}^T \cdot \widetilde{\mathbf{x}} \tag{13.10.19}$$

(Note that the routine `wtn` does the complete transformation of **A** into $\widetilde{\mathbf{A}}$.)

A typical integral operator that compresses well into wavelets has arbitrary (or even nearly singular) elements near to its main diagonal, but becomes smooth away from the diagonal. An example might be

$$A_{ij} = \begin{cases} -1 & \text{if } i = j \\ |i - j|^{-1/2} & \text{otherwise} \end{cases} \tag{13.10.20}$$

Figure 13.10.5 shows a graphical representation of the wavelet transform of this matrix, where i and j range over $1 \dots 256$, using the DAUB12 wavelets. Elements larger in magnitude than 10^{-3} times the maximum element are shown as black pixels, while elements between 10^{-3} and 10^{-6} are shown in gray. White pixels are $< 10^{-6}$. The indices i and j each number from the lower left.

In the figure, one sees the hierarchical decomposition into power-of-two sized blocks. At the edges or corners of the various blocks, one sees edge effects caused by the wrap-around wavelet boundary conditions. Apart from edge effects, within each block, the nonnegligible elements are concentrated along the block diagonals. This is a statement that, for this type of linear operator, a wavelet is coupled mainly to near neighbors in its own hierarchy (square blocks along the main diagonal) and near neighbors in other hierarchies (rectangular blocks off the diagonal).

The number of nonnegligible elements in a matrix like that in Figure 13.10.5 scales only as N, the linear size of the matrix; as a rough rule of thumb it is about $10N \log_{10}(1/\epsilon)$, where ϵ is the truncation level, e.g., 10^{-6}. For a 2000 by 2000 matrix, then, the matrix is sparse by a factor on the order of 30.

Various numerical schemes can be used to solve sparse linear systems of this "hierarchically band diagonal" form. Beylkin, Coifman, and Rokhlin [1] make the interesting observations that (1) the product of two such matrices is itself hierarchically band diagonal (truncating, of course, newly generated elements that are smaller than the predetermined threshold ϵ); and moreover that (2) the product can be formed in order N operations.

Fast matrix multiplication makes it possible to find the matrix inverse by Schultz's (or Hotelling's) method, see §2.5.

Figure 13.10.5. Wavelet transform of a 256×256 matrix, represented graphically. The original matrix has a discontinuous cusp along its diagonal, decaying smoothly away on both sides of the diagonal. In wavelet basis, the matrix becomes sparse: Components larger than 10^{-3} are shown as black, components larger than 10^{-6} as gray, and smaller-magnitude components are white. The matrix indices i and j number from the lower left.

Other schemes are also possible for fast solution of hierarchically band diagonal forms. For example, one can use the conjugate gradient method, implemented in §2.7 as `linbcg`.

CITED REFERENCES AND FURTHER READING:

Daubechies, I. 1992, *Wavelets* (Philadelphia: S.I.A.M.).

Strang, G. 1989, *SIAM Review*, vol. 31, pp. 614–627.

Beylkin, G., Coifman, R., and Rokhlin, V. 1991, *Communications on Pure and Applied Mathematics*, vol. 44, pp. 141–183. [1]

Daubechies, I. 1988, *Communications on Pure and Applied Mathematics*, vol. 41, pp. 909–996. [2]

Vaidyanathan, P.P. 1990, *Proceedings of the IEEE*, vol. 78, pp. 56–93. [3]

Mallat, S.G. 1989, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693. [4]

Freedman, M.H., and Press, W.H. 1992, preprint. [5]

13.11 Numerical Use of the Sampling Theorem

In §6.10 we implemented an approximating formula for Dawson's integral due to Rybicki. Now that we have become Fourier sophisticates, we can learn that the formula derives from *numerical* application of the sampling theorem (§12.1), normally considered to be a purely analytic tool. Our discussion is identical to Rybicki [1].

For present purposes, the sampling theorem is most conveniently stated as follows: Consider an arbitrary function $g(t)$ and the grid of sampling points $t_n = \alpha + nh$, where n ranges over the integers and α is a constant that allows an arbitrary shift of the sampling grid. We then write

$$g(t) = \sum_{n=-\infty}^{\infty} g(t_n) \operatorname{sinc} \frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.1}$$

where $\operatorname{sinc} x \equiv \sin x/x$. The summation over the sampling points is called the *sampling representation* of $g(t)$, and $e(t)$ is its error term. The sampling theorem asserts that the sampling representation is exact, that is, $e(t) \equiv 0$, if the Fourier transform of $g(t)$,

$$G(\omega) = \int_{-\infty}^{\infty} g(t) e^{i\omega t}\, dt \tag{13.11.2}$$

vanishes identically for $|\omega| \geq \pi/h$.

When can sampling representations be used to advantage for the approximate numerical computation of functions? In order that the error term be small, the Fourier transform $G(\omega)$ must be sufficiently small for $|\omega| \geq \pi/h$. On the other hand, in order for the summation in (13.11.1) to be approximated by a reasonably small number of terms, the function $g(t)$ itself should be very small outside of a fairly limited range of values of t. Thus we are led to two conditions to be satisfied in order that (13.11.1) be useful numerically: Both the function $g(t)$ and its Fourier transform $G(\omega)$ must rapidly approach zero for large values of their respective arguments.

Unfortunately, these two conditions are mutually antagonistic — the Uncertainty Principle in quantum mechanics. There exist strict limits on how rapidly the simultaneous approach to zero can be in both arguments. According to a theorem of Hardy [2], if $g(t) = O(e^{-t^2})$ as $|t| \to \infty$ and $G(\omega) = O(e^{-\omega^2/4})$ as $|\omega| \to \infty$, then $g(t) \equiv Ce^{-t^2}$, where C is a constant. This can be interpreted as saying that of all functions the Gaussian is the most rapidly decaying in both t and ω, and in this sense is the "best" function to be expressed numerically as a sampling representation.

Let us then write for the Gaussian $g(t) = e^{-t^2}$,

$$e^{-t^2} = \sum_{n=-\infty}^{\infty} e^{-t_n^2} \operatorname{sinc} \frac{\pi}{h}(t - t_n) + e(t) \tag{13.11.3}$$

The error $e(t)$ depends on the parameters h and α as well as on t, but it is sufficient for the present purposes to state the bound,

$$|e(t)| < e^{-(\pi/2h)^2} \tag{13.11.4}$$

which can be understood simply as the order of magnitude of the Fourier transform of the Gaussian at the point where it "spills over" into the region $|\omega| > \pi/h$.

When the summation in (13.11.3) is approximated by one with finite limits, say from $N_0 - N$ to $N_0 + N$, where N_0 is the integer nearest to $-\alpha/h$, there is a further truncation error. However, if N is chosen so that $N > \pi/(2h^2)$, the truncation error in the summation is less than the bound given by (13.11.4), and, since this bound is an overestimate, we shall continue to use it for (13.11.3) as well. The truncated summation gives a remarkably accurate representation for the Gaussian even for moderate values of N. For example, $|e(t)| < 5 \times 10^{-5}$ for $h = 1/2$ and $N = 7$; $|e(t)| < 2 \times 10^{-10}$ for $h = 1/3$ and $N = 15$; and $|e(t)| < 7 \times 10^{-18}$ for $h = 1/4$ and $N = 25$.