Chapter 26

Fast wavelet transforms

The discrete wavelet transforms are a class of transforms that can be computed in linear time. We treat wavelet transforms whose basis functions have compact support. These can be derived as a generalization of the Haar transform.

26.1 Wavelet filters

We motivate the *wavelet transform* as a generalization of the 'standard' Haar transform given in section 23.1 on page 461. We reformulate the Haar transform as a sequence of filtering steps.

We consider only (moving average) filters F defined by n coefficients ('taps') f_0, f_1, $\ldots$, f_{n-1}. Let A be the length-N sequence a_0, a_1, $\ldots$, a_{N-1}. We define $F_k(A)$ as the weighted sum

$$F_k(A) \quad := \quad \sum_{j=0}^{n-1} f_j \, a_{k+j \bmod N} \tag{26.1-1}$$

That is, $F_k(A)$ is the result of applying the filter F to the n elements a_k, a_{k+1}, a_{k+2}, $\ldots a_{k+n-1}$, possibly wrapping around.

Now assume that N is a power of two. Let H be the low-pass filter defined by $h_0 = h_1 = +1/\sqrt{2}$, and G be the high-pass filter defined by $g_0 = +1/\sqrt{2}$, $g_1 = -1/\sqrt{2}$. A single filtering step of the Haar transform consists of

- Computing the sums $s_0 = H_0(A)$, $s_2 = H_2(A)$, $s_4 = H_4(4)$, $\ldots$, $s_{N-2} = H_{N-2}(A)$
- Computing the differences $d_0 = G_0(A)$, $d_2 = G_2(A)$, $d_4 = G_4(4)$, $\ldots$, $d_{N-2} = G_{N-2}(A)$
- Writing the sums to the left half of A, and the differences to the right half:
  $A = [s_0, s_2, s_4, s_6, \ldots, s_{N-2}, d_0, d_2, d_4, d_6, \ldots, d_{N-2}]$

The Haar transform is obtained by applying the step to the whole sequence, then to its left half, then to its left quarter, $\ldots$, the left four elements, the left two elements. With the Haar transform no wrap-around occurs.

A the analogous filtering step for the wavelet transform is obtained by defining two length-n filters H (low-pass) and G (high-pass) subject to certain conditions. Firstly, we consider only filters with an even number n of coefficients.

Secondly, we define coefficients of G to be the reversed sequence of the coefficients of H with alternating signs:
$g_0 = +h_{n-1}$, $g_1 = -h_{n-2}$, $g_2 = +h_{n-3}$, $g_4 = -h_{n-3}$, $\ldots$, $g_{n-3} = -h_2$, $g_{n-2} = +h_1$, $g_{n-1} = -h_0$.

Thirdly, we require that the resulting transform is orthogonal. Let S be the matrix corresponding to one filtering step, ignoring the order:

$$S\,A \quad = \quad [s_0, d_0, s_2, d_2, s_4, d_4, s_6, d_6, \ldots, s_{N-2}, d_{N-2}] \tag{26.1-2}$$

With length-6 filters and $N = 16$ the matrix S would be

$$S \quad = \quad \begin{bmatrix} h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & h_4 & h_5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & g_4 & g_5 \\ h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 \\ g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 \\ h_2 & h_3 & h_4 & h_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 \\ g_2 & g_3 & g_4 & g_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 \end{bmatrix} \tag{26.1-3a}$$

$$= \quad \begin{bmatrix} +h_0 & +h_1 & +h_2 & +h_3 & +h_4 & +h_5 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\ +h_5 & -h_4 & +h_3 & -h_2 & +h_1 & -h_0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\ 0 & 0 & +h_0 & +h_1 & +h_2 & +h_3 & +h_4 & +h_5 & 0 & 0 & 0 & \ldots & 0 \\ 0 & 0 & +h_5 & -h_4 & +h_3 & -h_2 & +h_1 & -h_0 & 0 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 0 & +h_0 & +h_1 & +h_2 & +h_3 & +h_4 & +h_5 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 0 & +h_5 & -h_4 & +h_3 & -h_2 & +h_1 & -h_0 & 0 & \ldots & 0 \\ & & & \ddots & & & & & & \ddots & & & \end{bmatrix} \tag{26.1-3b}$$

The orthogonality requires that $S\,S^T = \mathrm{id}$, that is (setting $h_j = 0$ for $j < 0$ and $j \geq n$)

$$\sum_j h_j^2 \quad = \quad 1 \tag{26.1-4a}$$

$$\sum_j h_j\, h_{j+2} \quad = \quad 0 \tag{26.1-4b}$$

$$\sum_j h_j\, h_{j+4} \quad = \quad 0 \tag{26.1-4c}$$

In general, the following $n/2$ *wavelet conditions* are obtained:

$$\sum_j h_j^2 \quad = \quad 1 \tag{26.1-5a}$$

$$\sum_j h_j\, h_{j+2i} \quad = \quad 0 \quad \text{where} \quad i = 1, 2, 3, \ldots, n/2 - 1 \tag{26.1-5b}$$

We call a filter H satisfying these conditions a *wavelet filter*.

For the wavelet transform with $n = 2$ filter taps there is only condition, $h_0^2 + h_1^2 = 1$, leading to the parametric solution $h_0 = \sin(\phi)$, $h_1 = \cos(\phi)$. Setting $\phi = \pi/4$ one obtains $h_0 = h_1 = 1/\sqrt{2}$, corresponding to the Haar transform.

26.2 Implementation

A container class for wavelet filters is [FXT: `class wavelet_filter` in wavelet/waveletfilter.h]:

```
class wavelet_filter
{
public:
    double *h_;  // low-pass filter
    double *g_;  // high-pass filter
    ulong n_;    // number of taps

    void ctor_core()
    {
        h_ = new double[n_];
        g_ = new double[n_];
    }

    wavelet_filter(const double *w, ulong n=0)
    {
        if ( 0!=n )  n_ = n;
        else  // zero terminated array w[]
        {
            n_ = 0;
            while ( w[n_]!=0 )  ++n_;
        }

        ctor_core();

        for (ulong i=0, j=n_-1;  i<n_;  ++i, --j)
        {
            h_[i] = w[i];

            if ( !(i&1) )  g_[j] = -h_[i];  // even indices
            else           g_[j] = +h_[i];  // odd indices
        }
    }
  [--snip--]
```

The wavelet conditions can be checked via

```
    bool check(double eps=1e-6) const
    {
        if ( fabs(norm_sqr(0)-1.0) > eps )  return false;

        for (ulong i=1;  i<n_/2;  ++i)
            if ( fabs(norm_sqr(i)) > eps )  return false;

        return true;
    }
```

where `norm_sqr()` computes the sums in the relations 26.1-5a and 26.1-5b:

```
    static double norm_sqr(const double *h, ulong n, ulong s=0)
    {
        s *= 2;  // Note!
        if ( s>=n )  return 0.0;

        double v = 0;
        for (ulong k=0,j=s;  j<n;  ++k,++j)  v += (h[k]*h[j]);
        return  v;
    }

    double norm_sqr(ulong s=0)  const  { return norm_sqr(h_, n_, s); }
```

A wavelet step can be implemented as [FXT: wavelet/wavelet.cc]:

```
void
wavelet_step(double *f, ulong n, const wavelet_filter &wf, double *t)
{
    const ulong nh = (n>>1);
    const ulong m = n-1;  // mask to compute modulo n (n is a power of two)
    for (ulong i=0,j=0; i<n; i+=2,++j) // i \in [0,2,4,..,n-2];  j \in [0,1,2,..,n/2-1]
    {
        double s = 0.0,  d = 0.0;
        for (ulong k=0; k<wf.n_; ++k)
        {
            ulong w = (i+k) & m;
            s += (wf.h_[k] * f[w]);
            d += (wf.g_[k] * f[w]);
```

```
        }
        t[j] = s;
        t[nh+j] = d;
    }
    copy(t, f, n);  // f[] := t[]
}
```

The wavelet transform itself is

```
void
wavelet(double *f, ulong ldn, const wavelet_filter &wf, ulong minm/*=2*/)
{
    ulong n = (1UL<<ldn);
    ALLOCA(double, t, n);
    for (ulong m=n; m>=minm; m>>=1)  wavelet_step(f, m, wf, t);
}
```

The step for the inverse transform is [FXT: wavelet/invwavelet.cc]:

```
void
inverse_wavelet_step(double *f, ulong n, const wavelet_filter &wf, double *t)
{
    const ulong nh = (n>>1);
    const ulong m = n-1;  // mask to compute modulo n (n is a power of two)
    null(t, n);  // t[] := [0,0,...,0]
    for (ulong i=0, j=0;  i<n;  i+=2, ++j)
    {
        const double x = f[j],  y = f[nh+j];
        for (ulong k=0;  k<wf.n_;  ++k)
        {
            ulong w = (i+k) & m;
            t[w] += (wf.h_[k] * x);
            t[w] += (wf.g_[k] * y);
        }
    }
    copy(t, f, n);  // f[] := t[]
}
```

The inverse transform itself now is

```
void
inverse_wavelet(double *f, ulong ldn, const wavelet_filter &wf, ulong minm/*=2*/)
{
    ulong n = (1UL<<ldn);
    ALLOCA(double, t, n);
    for (ulong m=minm; m<=n; m<<=1)  inverse_wavelet_step(f, m, wf, t);
}
```

A readable source about wavelets is [248].

26.3 Moment conditions

As the wavelet conditions do not uniquely define the wavelet filters on can impose additional properties for the filters used. We require that, for an $2n$-tap wavelet filter, the first $n/2$ moments vanish:

$$\sum_j (-1)^j \, h_j \quad = \quad 0 \tag{26.3-1a}$$

$$\sum_j (-j)^k \, h_j \quad = \quad 0 \quad \text{where} \quad k = 1, 2, 3, \ldots, n/2 - 1 \tag{26.3-1b}$$

One motivation for these *moment conditions* is that for reasonably smooth signals (for which a polynomial approximation is good) the transform coefficients from the high-pass filter (the d_k) will be close to zero. With compression schemes that simply discard transform coefficients with small values this is a desirable property.

The class [FXT: `class wavelet_filter` in wavelet/waveletfilter.h] has a method to compute the moments of the filter:

```
static double moment(const double *h, ulong n, ulong x=0)
{
    if ( 0==x )
    {
        double v = 0.0;
        for (ulong k=0; k<n; k+=2)  v += h[k];
        for (ulong k=1; k<n; k+=2)  v -= h[k];
        return v;
    }

    double dk;

    double ve = 0;
    dk = 2.0;
    for (ulong k=2;  k<n;  k+=2, dk+=2.0)  ve += (pow(dk,x) * h[k]);

    double vo = 0;
    dk = 1.0;
    for (ulong k=1;  k<n;  k+=2, dk+=2.0)  vo += (pow(dk,x) * h[k]);
    return  ve - vo;
}

double moment(ulong x=0)  const  { return moment(h_, n_, x); }
```

Filter coefficients that satisfy the moment conditions are given in [FXT: wavelet/daubechies.cc]:

```
extern const double Daub1[] = {
+7.071067811865475244008443621048e-01,
+7.071067811865475244008443621048e-01 };

extern const double Daub2[] = {
+4.829629131445341433748715998644e-01,
+8.365163037378079055752937809168e-01,
+2.241438680420133810259727622404e-01,
-1.294095225512603811744494188120e-01 };

extern const double Daub3[] = {
+3.326705529500826159985115891390e-01,
+8.068915093110925764944936040887e-01,
+4.598775021184915700951519421476e-01,
-1.350110200102545886963899066993e-01,
-8.544127388202666169281916918177e-02,
+3.522629188570953660274066471551e-02 };

extern const double Daub4[] = {
+2.303778133088965008632911830440e-01,
+7.148465705529156470899219552739e-01,
+6.308807679298589078817163383006e-01,
-2.798376941685985421141374718007e-02,
-1.870348117190930840795706727890e-01,
+3.084138183556076362721936253495e-02,
+3.288301166688519973540751354924e-02,
-1.059740178506903210488320852402e-02 };

  [--snip--]
extern const double Daub38[] = {...}
```

The names reflect the number $n/2$ of vanishing moments. Reversing or negating the sequence of filter coefficients leads to trivial variants that also satisfies the moment conditions.

For the filters of length $n \geq 6$ there are solutions that are essentially different. For $n = 6$ there is one complex solution besides `Daub3[]`:

```
-0.09556007476957763 + 0.0508627772544*I
+0.08121662052705924 + 0.1525883317632*I
+0.72145023542906591 + 0.1017255545088*I
+0.72145023542906591 - 0.1017255545088*I
+0.08121662052705924 - 0.1525883317632*I
-0.09556007476957763 - 0.0508627772544*I
```

For $n = 8$ there is, besides `Daub4[]`, an additional real solution (left), and a complex one (right):

```
-0.07576571478950221             +0.02152475910155493 + 0.0184283603930*I
-0.02963552764600249             -0.06571356411493559 + 0.0176790547520*I
+0.49761866763277498             -0.19397617446078878 - 0.1319957453155*I
+0.80373875180513208             +0.24627664139071534 - 0.2801719341011*I
+0.29785779560530605             +0.85723045931761476 - 0.0921418019654*I
-0.09921954357663353             +0.59199318785735184 + 0.2064584925288*I
-0.01260396726203130             +0.02232773722816661 + 0.2057091868878*I
+0.03222310060405146             -0.06544948394658407 + 0.0560343868202*I
```

The numbers of solutions grows exponentially with n. The filters given in [FXT: wavelet/daubechies.cc] are the filters for the so-called *Daubechies wavelets* (some closed form expressions for the filter coefficients are given in [82]).

Filter coefficients that satisfy the wavelet and the moment conditions can be found by a Newton iteration for zeros of the function $F : \mathbb{R}^n \to \mathbb{R}^n$, $F(\vec{h}) := \vec{w}$ where $w_i = F_i(\vec{h}) = F_i(h_0, h_1, \ldots, h_5)$. For example, with $n = 6$, the F_i are defined by

```
F[1]:  h0^2 + h1^2 + h2^2 + h3^2 + h4^2 + h5^2 - 1
F[2]:  h2*h0 + h3*h1 + h4*h2 + h5*h3
F[3]:  h4*h0 + h5*h1
F[4]:  -h0 + h1 + -h2 + h3 + -h4 + h5
F[5]:  h1 + -2*h2 + 3*h3 + -4*h4 + 5*h5
F[6]:  h1 + -4*h2 + 9*h3 + -16*h4 + 25*h5
```

The derivative is given by the *Jacobi matrix* J. It has the components $J_{r,c} := \frac{dF_r}{dh_c}$. Its rows are

```
J[1]= [2*h0, 2*h1, 2*h2, 2*h3, 2*h4, 2*h5]
J[2]= [h2, h3, h0 + h4, h1 + h5, h2, h3]
J[3]= [h4, h5, 0, 0, h0, h1]
J[4]= [-1, 1, -1, 1, -1, 1]
J[5]= [0, 1, -2, 3, -4, 5]
J[6]= [0, 1, -4, 9, -16, 25]
```

Now iterate (the equivalent to Newton's iteration, $x_{k+1} := x_k - f(x_k)/f'(x_k)$)

$$\vec{h}_{k+1} \quad := \quad \vec{h}_k - J^{-1}(\vec{h}_k)\, F(\vec{h}_k) \qquad (26.3\text{-}2)$$

The computations have to be carried out with a rather great precision to avoid catastrophic loss of precision.