

REAL-TIME WAVELET DENOISING

DSP HARDWARE LAB
FINAL PROJECT



NASH BORGES

FOURIER & WAVELET TRANSFORMS

- The **Fourier Transform** decomposes a signal into sinusoids of infinite length to perform frequency analysis. In order to perform temporal analysis, the signal must be “windowed.”
- Short-Time Fourier Transform:

$$(T^{win} f)(\omega, t) = \int f(s)g(s-t)e^{-i\omega s} ds$$

- $g(\omega, t)$ are translated in time.
- The **Wavelet Transform** decomposes a signal into translated and dilated “daughter” wavelets derived from a “mother” wavelet.
- Wavelet Transform:

$$(T^{wav} f)(a, b) = |a|^{-1/2} \int f(t)\psi\left(\frac{t-b}{a}\right) dt$$

- $\psi(a, b)$ are translated in time AND dilated in frequency.

WAVELET BENEFITS

- Provide efficient localization in both time and frequency (or scale).
- Multi-resolution decomposition separates a signal in a way that is often superior to other methods for analysis and processing.
- Near optimal for a wide class of signals for compression, **de-noising**, and detection.
- Adjustable and adaptable: wavelets can be tailored to a specific class of signals or to a specific purpose.

WAVELET TYPES

Crude Wavelets

- Wavelets which lack many interesting properties are called “crude”.
- **Morlet** and **Mexican hat** wavelets
 - Explicit wavelet function
 - Do not have compact support, vanishing moments, and are not orthogonal
 - Filters cannot be constructed
 - Forward CWT is useful for mathematical demonstrations since the wavelet function exists as a formula.



WAVELET TYPES

Orthogonal Wavelets

Haar



Haar wavelet – The only orthogonal wavelet for which symmetry and exact reconstruction are possible with FIR filters. The oldest and simplest wavelet with only 2 filter taps, any coefficient modification results in strong aliasing.

Daubechies wavelets – Good for images containing sharp transitions and high contrast, but very asymmetric yielding nonlinear phase, which is bad for audio.

Coiflets - Similar properties to Daubechies but phi also has maximal number of vanishing moments.

Symmlets - Similar properties to Daubechies but optimized for symmetry (almost linear phase).

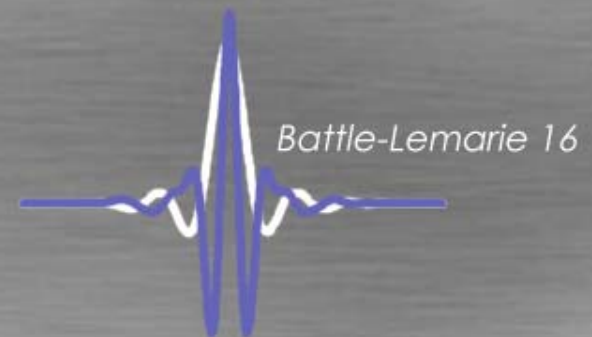
Meyer wavelets - Infinitely differentiable (good for audio resynthesis) and symmetric (linear phase). The wavelet is not compactly supported, but a finite filter can be approximated.



WAVELET TYPES

Biorthogonal Wavelets

- Both wavelet and scaling functions are symmetric yielding linear phase.
- High degree of regularity (good for resynthesis) and frequency band separation.
- Wavelet and scaling functions are different for analysis and resynthesis.
- **Battle and Lemarié** introduced biorthogonal wavelets based on polynomial splines.
 - For splines of degree m , the wavelet function has $m+1$ vanishing moments and is $m-1$ times continuously differentiable (quite smooth).
 - Not compactly supported and must be truncated for finite implementation. However, the wavelet function has exponential decay so this does not introduce much error.
 - Maximum regularity with symmetry and minimum support.



FROM WAVELETS TO FILTER BANKS

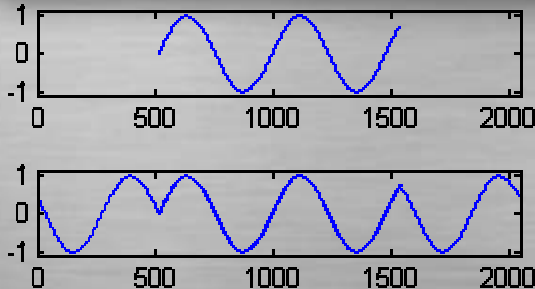
- Wavelet functions are not normally used when calculating the DWT.
- Wavelets can be derived from corresponding low pass and high pass filters of a perfect reconstruction filter bank.
- The scaling function can be calculated from the low pass filter:

$$\phi(t) = 2 \sum_{k=0}^N h_0(k) \phi(2t - k)$$

- The wavelet function can be calculated from the high pass filter:

$$\psi(t) = 2 \sum_{k=0}^N h_1(k) \phi(2t - k)$$

MALLAT'S ALGORITHM

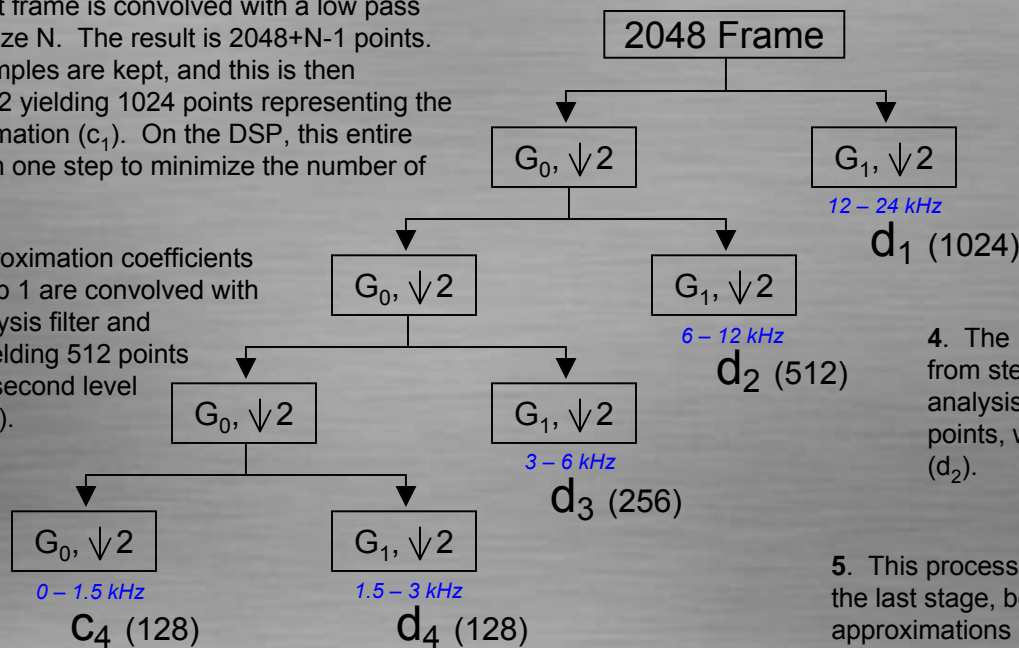


Each half of a 1024 point frame gets mirrored like a double door opening. This diminishes the “edge effects” of the DWT.

Now, the 2048 point frame is processed using Mallat’s algorithm. After the reconstruction, the mirrored parts are thrown away.

1. The 2048 point frame is convolved with a low pass analysis filter of size N . The result is $2048+N-1$ points. The first 2048 samples are kept, and this is then downsampled by 2 yielding 1024 points representing the first level approximation (c_1). On the DSP, this entire process is done in one step to minimize the number of multiplications.

3. The 1024 approximation coefficients resulting from step 1 are convolved with the low pass analysis filter and downsampled, yielding 512 points representing the second level approximation (c_2).



2. The 2048 point frame is similarly convolved with the high pass analysis filter and downsampled yielding 1024 points, which are stored as the first level details (d_1).

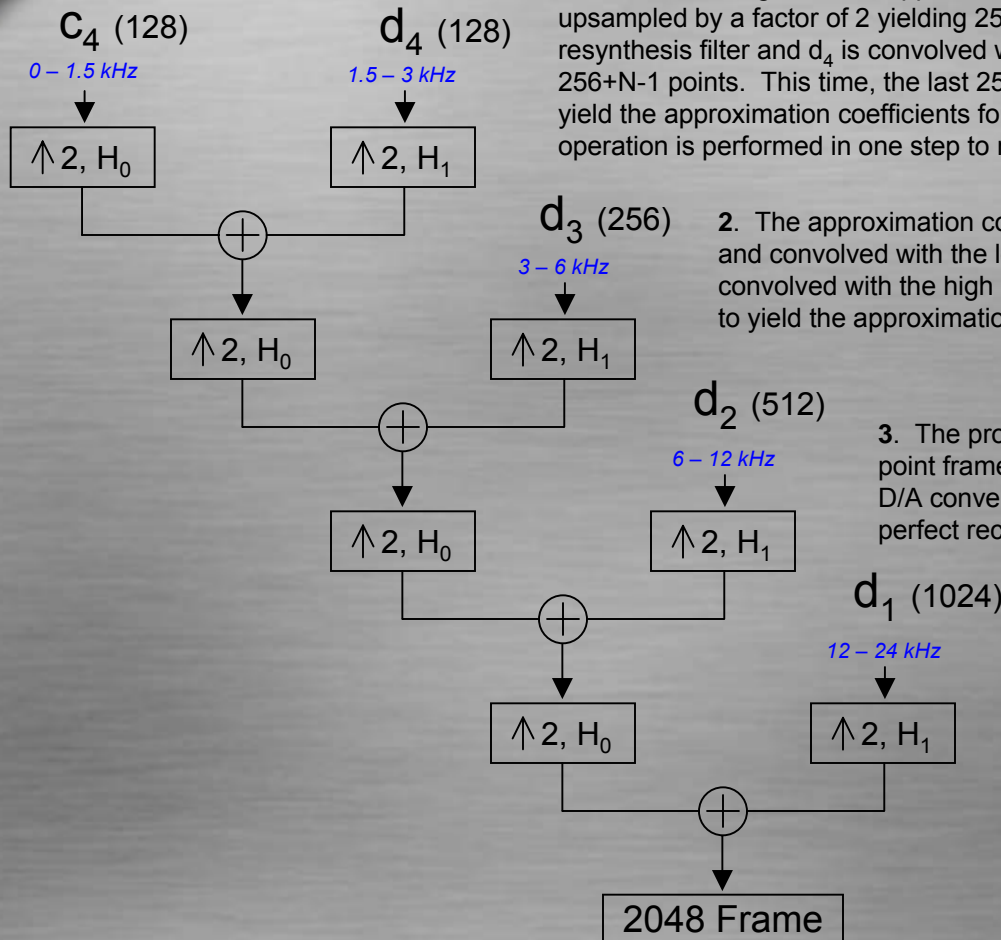
4. The 1024 approximation coefficients resulting from step 1 are also convolved with the high pass analysis filter and downsampled yielding 512 points, which are stored as the second level details (d_2).

5. This process is repeated two more times and after the last stage, both the fourth level details (d_4) and the approximations (c_4) are stored. Therefore, the total number of points stored is still 2048, but now they are in the wavelet domain.

DENOISING

- Linear Denoising is performed by zeroing out an entire vector (i.e. d_1, d_2, d_3, d_4, c_4) in the wavelet domain. This assumes that the noise is only present in one of the frequency bands.
- Non-linear denoising assumes that noise is equally present in all frequency bands.
 - Hard thresholding zeros out all the coefficients that are below a certain threshold.
 - Soft thresholding then reduces the remaining coefficients by the threshold.

PERFECT RECONSTRUCTION



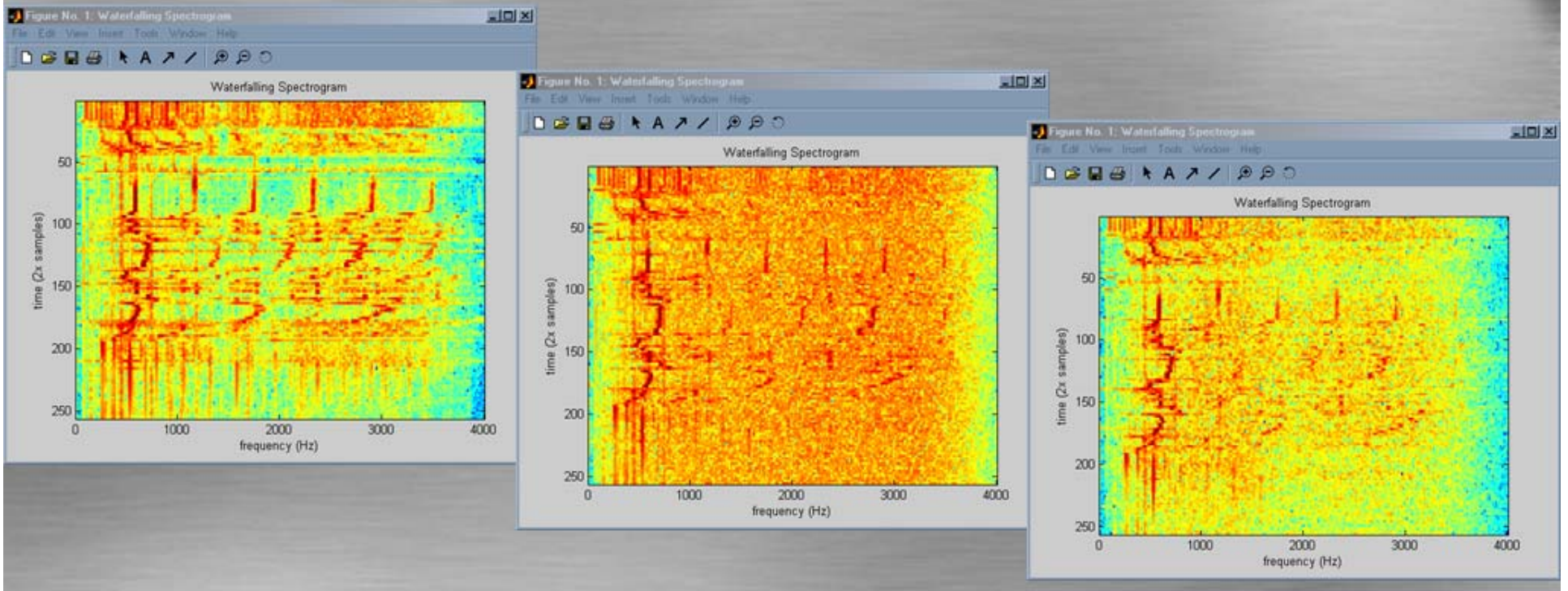
1. After denoising, the final approximation (c_4) and detail (d_4) coefficients are both upsampled by a factor of 2 yielding 256 points. c_4 is then convolved with the low pass resynthesis filter and d_4 is convolved with the high pass resynthesis filter yielding $256+N-1$ points. This time, the last 256 points are kept from each and then summed to yield the approximation coefficients for the next level (c_3). On the DSP, this entire operation is performed in one step to minimize storage requirements.

2. The approximation coefficients resulting from step 1 are then upsampled and convolved with the low pass resynthesis filter. d_3 is upsampled and convolved with the high pass resynthesis filter. The outputs are then summed to yield the approximation coefficients for the next level (c_2).

3. The process is repeated two more times until the output yields a 2048 point frame. The middle 1024 points of this frame are then sent to the D/A converter. If no denoising was performed, this process yields perfect reconstruction of the original frame.

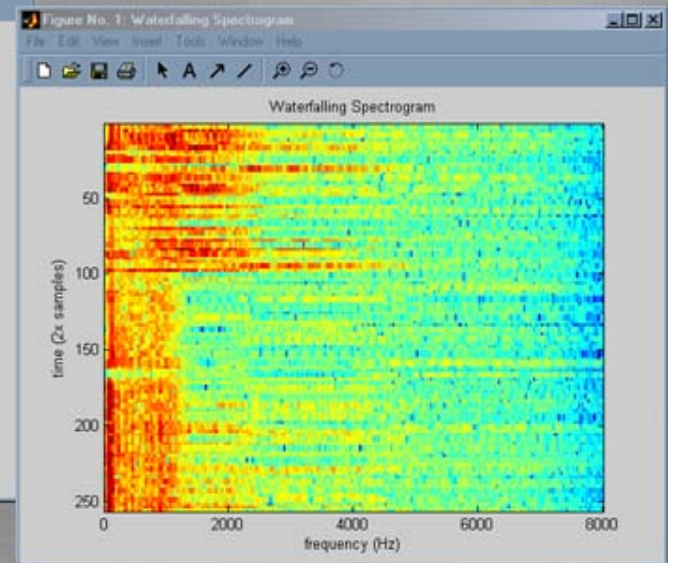
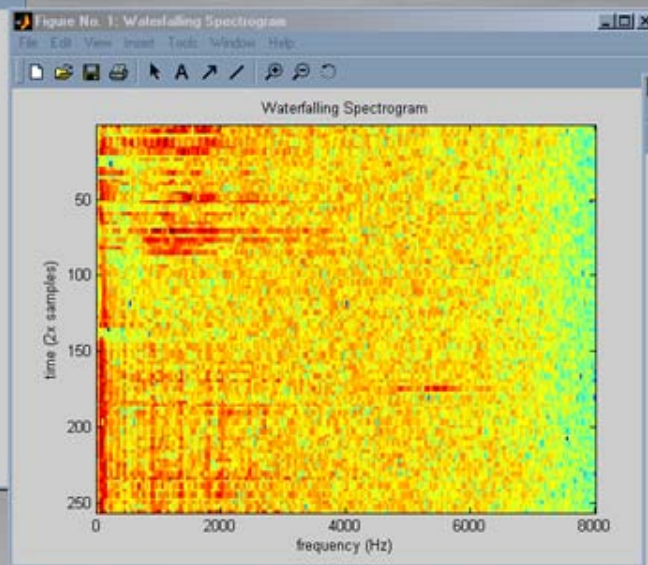
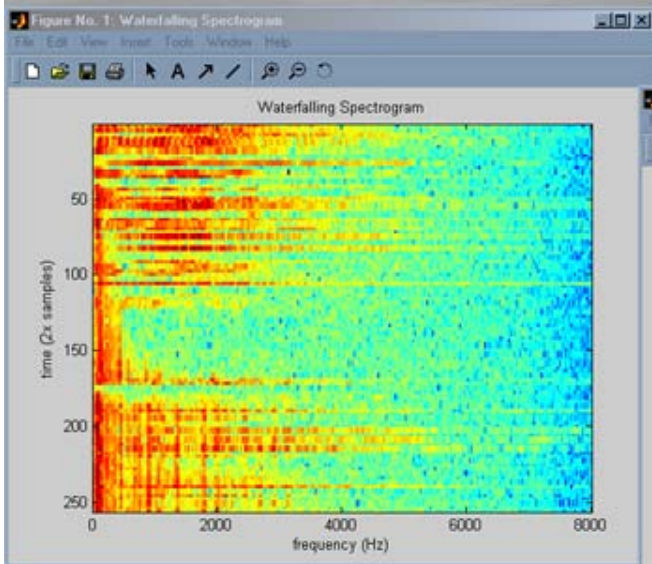
REAL-TIME RESULTS

1. Guitar solo in “Chicago Blues” by the Boston Funk Band c/o Data Acquisition Toolbox in MATLAB.
2. 20% White Gaussian Noise added.
3. Denoised with the discrete Meyer wavelet using soft thresholding at 0.01 (coeffs range 0 to 1).



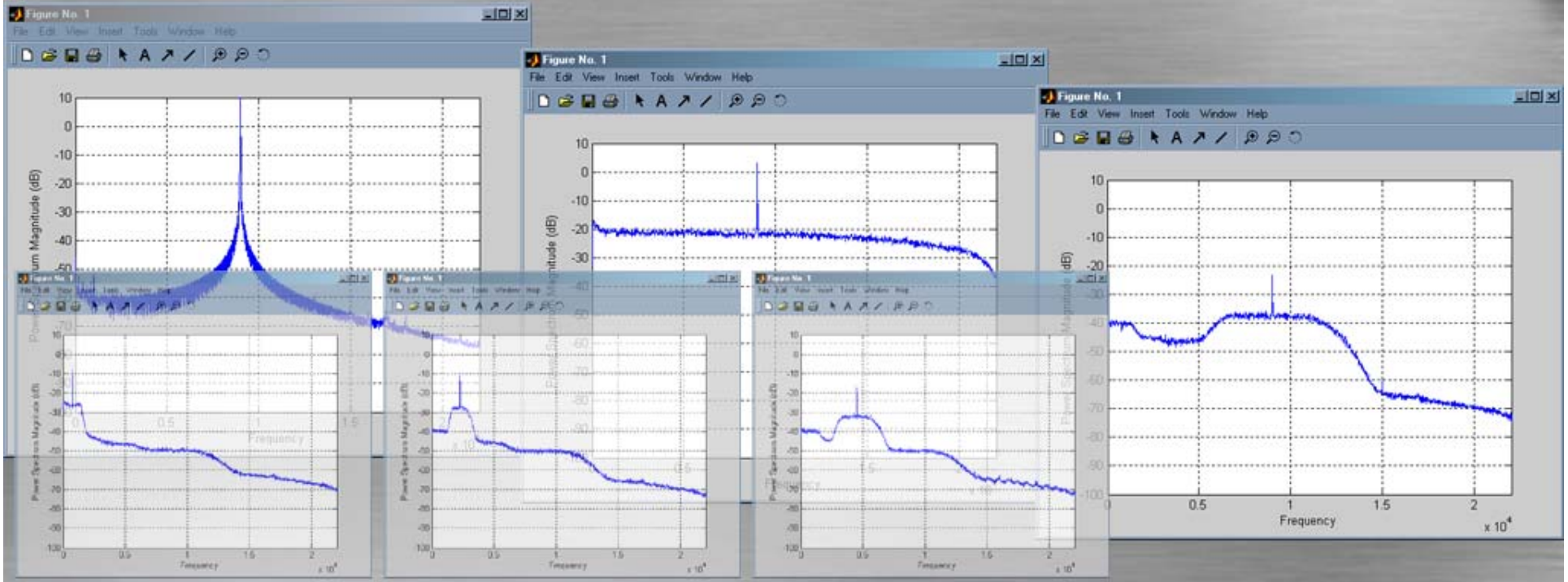
REAL-TIME RESULTS

1. End of live performance of “Everlong” by the Foo Fighters followed by applause.
2. 10% White Gaussian Noise added.
3. Denoised with the discrete Meyer wavelet using soft thresholding at 0.03.



REAL-TIME RESULTS

1. 9kHz sinusoid.
2. 80% white Gaussian noise added.
3. Denoised with the discrete Meyer wavelet using soft thresholding at 0.15.
- 4,5,6. Denoiser “adapts” to the sinusoid changing frequencies (4.5kHz, 2.25kHz, and 750Hz) without parameter modification.





CONVLEFTDOWN

```
void convleftdown(float *DOut, float *COut, float *CIn, int count)
{
    float dsum, csum;
    int i, j, k;

    // convolve, keep left, and downsample
    for(i=0; i<=count-1; i+=2) {
        dsum = 0; // keep left
        csum = 0; // initialize the D coefficient
                // initialize the C Coefficient

        // dot product
        for(j=0, k=i; j<=N; j++, k--) {
            if ((j<=i) && (k<=count-1)) {
                dsum += CIn[k] * H1FLIP[j]; // convolve input with H1flip
                csum += CIn[k] * H0FLIP[j]; // convolve input with H0flip
            }
        }

        // store downsampled
        j = i>>1;
        DOut[j] = dsum;
        COut[j] = csum;
    }
}
```

ZEROCOLUMNS

```
void zerocolumns(float *DFirst, float *DSecond, float *DThird, float *DFourth, float *CFourth)
{
    int i;

    if ((short)HPI_Block.Col1)
        for (i=0; i<BUFFER_COUNT; i++)
            DFirst[i] = 0;
    if ((short)HPI_Block.Col2)
        for (i=0; i<(BUFFER_COUNT>>1); i++)
            DSecond[i] = 0;
    if ((short)HPI_Block.Col3)
        for (i=0; i<(BUFFER_COUNT>>2); i++)
            DThird[i] = 0;
    if ((short)HPI_Block.Col4)
        for (i=0; i<(BUFFER_COUNT>>3); i++)
            DFourth[i] = 0;
    if ((short)HPI_Block.Col5)
        for (i=0; i<(BUFFER_COUNT>>3); i++)
            CFourth[i] = 0;
}
```

THRESHOLDING

```
void thresholding(float *DFirst, float *DSecond, float *DThird, float *DFourth, float *CFourth)
{
    float adjthreshold = HPI_Block.Threshold*32768.0;
    int i;

    for (i=0; i<BUFFER_COUNT; i++)
        if ((DFirst[i] >= -adjthreshold) && (DFirst[i] <= adjthreshold))
            DFirst[i] = 0;
        else if ((short)HPI_Block.SoftThresholding) {
            if (DFirst[i] > 0)
                DFirst[i] = DFirst[i] - adjthreshold;
            else
                DFirst[i] = DFirst[i] + adjthreshold;
        }

    for (i=0; i<(BUFFER_COUNT>>1); i++)
        if ((DSecond[i] >= -adjthreshold) && (DSecond[i] <= adjthreshold))
            DSecond[i] = 0;
        else if ((short)HPI_Block.SoftThresholding) {
            if (DSecond[i] > 0)
                DSecond[i] = DSecond[i] - adjthreshold;
            else
                DSecond[i] = DSecond[i] + adjthreshold;
        }

    .
    .
    .
}
```

UPCONVRIGHT

```
void upconvrightright(float *COut, float *DIn, float *CIn, int count)
{
    float dsum, csum;
    int i,j,k,test;

    // upsample, convolve, and keep right
    for(i=N; i<=count+N-1; i++) {
        dsum = 0; // keep right
        csum = 0; // initialize the D coefficient
                // initialize the C coefficient

        // dot product
        for(j=0,k=i; j<=N; j++,k--) {
            test = (k << 31) >> 31; // test if k is even, this does "upsampling"
            if ((j<=i) && (k<=count-1) && (test==0)) {
                dsum += DIn[k>>1] * H1[j]; // convolve upsampled with H1
                csum += CIn[k>>1] * H0[j]; // convolve upsampled with H0
            }
        }

        // store
        csum += dsum;
        COut[i-N] = csum;
    }
}
```