Introduction to Computer Music

Week 8

Instructor: Prof. Roger B. Dannenberg

Topics Discussed: FFT, Inverse FFT, Overlap Add, Reconstruction from Spectral Frames

Chapter 8

Spectral Processing

Topics Discussed: FFT, Inverse FFT, Overlap Add, Reconstruction from Spectral Frames

8.1 FFT Analysis and Reconstruction

Previously, we have learned about the spectral domain in the context of sampling theory, filters, and the Fourier transform, in particular the fast Fourier transform, which we used to compute the spectral centroid. In this chapter, we focus on the details of converting from a time domain representation to a frequency domain representation, operating on the frequency domain representation, and then reconstructing the time domain signal.

We emphasized earlier that filters are *not* typically implemented in the frequency domain, in spite of our theoretical understanding that filtering is effectively multiplication in the frequency domain. This is because we cannot compute a Fourier transform on a infinite signal or even a very long one. Therefore, our only option is to use short time transforms as we did with computing the spectral centroid. That *could* be used for filtering, but there are problems associated with using overlapping short-time transforms. Generally, we do not use the FFT for filtering.

Nevertheless, operations on spectral representations are interesting for analysis and synthesis. In the following sections, we will review the Fourier transform, consider the problems of long-time analysis/synthesis using short-time transforms, and look at spectral processing in Nyquist.

8.1.1 Review of FFT Analysis

Here again are the equations for the Fourier Transform in the continuous and discrete forms:

Continuous Fourier Transform
Real part:

$$R(\omega) = \int_{-\infty}^{\infty} f(t)\cos(\omega t)dt \tag{8.1}$$

Imaginary part:

$$X(\omega) = -\int_{-\infty}^{\infty} f(t)\sin(\omega t)dt \tag{8.2}$$

Discrete Fourier Transform

Real part:

$$R_k = \sum_{x=0}^{N-1} x_i \cos(2\pi ki/N) \tag{8.3}$$

Imaginary part:

$$X_k = -\sum_{x=0}^{N-1} x_i \sin(2\pi ki/N) \tag{8.4}$$

Recall from the discussion of the spectral centroid that when we take FFTs in Nyquist, the spectra appear as floating point arrays. As shown in Figure 8.1, the first element of the array (index 0) is the DC (0 Hz) component,[1] and then we have alternating cosine and sine terms all the way up to the top element of the array, which is the cosine term of the Nyquist frequency. To visualize this in a different way, in Figure 8.2, we represent the basis functions (cosine and sine functions that are multiplied by the signal), and the numbers here are the indices in the array. The second element of the array (the gray curve labeled 1), is a single period of cosine across the duration of the analysis frame. The next element in the array (the black curve labeled 2) is a single sine function over the length of the array. Proceeding from there, we have two cycles of cosine, two cycles of sine. Finally, the Nyquist frequency term has $n/2$ cycles of a cosine, forming alternating samples of +1, -1, +1, -1, +1, ... (The sine terms at frequency 0 and the Nyquist frequency $N/2$ are omitted because $\sin(2\pi ki/N) = 0$ if $k = 0$ or $k = N/2$.)

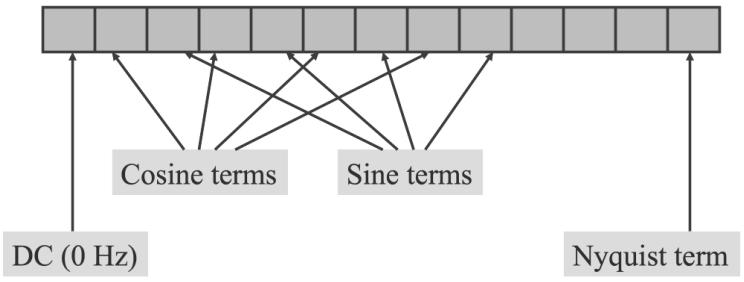

Figure 8.1: The spectrum as a floating point array in Nyquist. Note that the real and imaginary parts are interleaved, with a real/imaginary pair for each frequency bin. The first and last bin have only one number (the real part) because the imaginary part for these bins is zero.

Following the definition of the Fourier transform, these *basis functions* are multiplied by the input signal, the products are summed, and the sums are the output of the transform, the so-called Fourier *coefficients*. Each one of the basis functions can be viewed as a frequency analyzer—it picks out a particular frequency from the input signal. The frequencies selected by the basis functions are $K/duration$, where index $K \in \{0, 1, ..., n/2\}$.

Knowing the frequencies of basis functions is very important for interpreting or operating on spectra. For example, if the analysis window size is 512 samples, the sample rate is 44100 Hz, and value at index 5 of the spectrum is large, what

[1] This component comes from the particular case where $\omega = 0$, so $\cos \omega t = 1$, and the integral is effectively computing the average value of the signal. In the electrical world where we can describe electrical power as AC (alternating current, voltage oscillates up an down, e.g. at 60 Hz) or DC (direct current, voltage is constant, e.g. from a 12-volt car battery), the average value of the signal is the DC component or DC offset, and the rest is "AC."

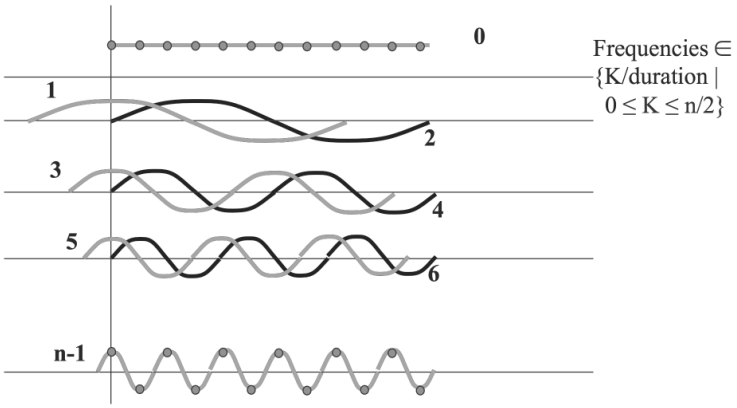

Figure 8.2: The so-called basis functions for the Fourier transform labeled with bin numbers. To compute each Fourier coefficient, form the dot product of the basis function and the signal. You can think of this as the weighted average of the signal where the basis function provides the weights. Note that all basis functions, and thus the frequencies they select from the input signal, are harmonics: multiples of a fundamental frequency that has a period equal to the size of the FFT. Thus, if the N input points represent 1/10 second of sound, the bins will represent 10 Hz, 20 Hz, 30 HZ, ..., etc. Note that we show sin functions, but the final imaginary (sin) coefficient of the FFT is negated (see Equation 8.4.)

strong frequency does that indicate? The duration is $512/44100 = 0.01161$ s, and from Figure 8.2, we can see there are 3 periods within the analysis window, so $K = 3$, and the frequency is $3/0.01161 = 258.398$ Hz. A large value at array index 5 indicates a strong component near 258.398 Hz.

Now, you may ask, what about some near-by frequency, say, 300 Hz? The next analysis frequency would be 344.531 Hz at $K = 4$, so what happens to frequencies between 258.398 and 344.531 Hz? It turns out that each basis function is not a perfect "frequency selector" because of the finite number of samples considered. Thus, intermediate frequencies (such as 300 Hz) will have some correlation with more than one basis function, and the discrete spectrum will have more than one non-zero term. The highest magnitude coefficients will be the ones whose basis function frequencies are nearest that of the sinusoid(s) being analyzed.

8.1.2 Perfect Reconstruction

Is it possible to go from the time domain to the spectral domain and back to the time domain again without any loss or distortion? One property that works in our favor is that the FFT is information-preserving. There is a Fast Inverse Short-Time Discrete Fourier Transform, or IFFT for short, that converts Fourier coefficients back into the original signal. Thus, one way to convert to the spectral domain and back, without loss, is shown in Figure 8.3.

In general, each short-time spectrum is called a *spectral frame* or an *analysis frame*.

The problem with Figure 8.3 is that if we change any coefficients in the spectrum, it is likely that the reconstructed signal will have discontinuities at the boundaries between one analysis frame and the next. You may recall from our early discussion on splicing that cross-fades are important to avoid clicks due to discontinuities. In fact, if there are periodic discontinuities (every analysis frame),

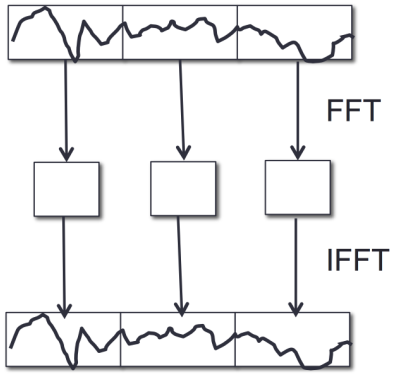

Figure 8.3: One (flawed) approach to lossless conversion to the frequency domain and back to the time domain. This works fine unless you do any manipulation in the frequency domain, in which case discontinuities will create horrible artifacts.

a distinct buzz will likely be heard.

Just as we used envelopes and cross-fades to eliminate clicks in other situations, we can use envelopes to smooth each analysis frame before taking the FFT, and we can apply the smoothing envelope again after the IFFT to ensure there are no discontinuities in the signal. These analysis frames are usually called *windows* and the envelope is called a *windowing function*.

Figure 8.4 illustrates how overlapping windows are used to obtain groups of samples for analysis. From the figure, you would (correctly) expect that ideal window functions would be smooth and sum to 1. One period of the cosine function raised by 1 (so the range is from 0 to 2) is a good example of such a function. Raised cosine windows (also called Hann or Hanning windows, see Figure 8.5) sum to one if they overlap by 50%.

The technique of adding smoothed overlapped intervals of sound is related to granular synthesis. When the goal is to produce a continuous sound (as opposed to a turbulent granular texture), this approach is called *overlap-add*.

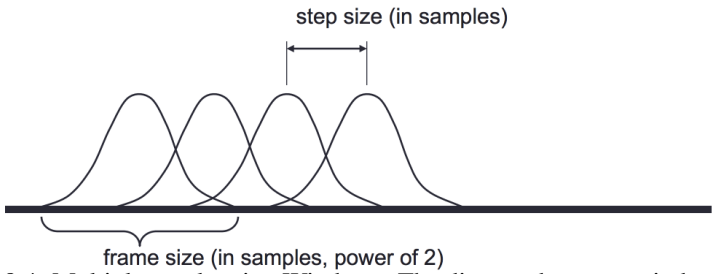

Figure 8.4: Multiple overlapping Windows. The distance between windows is the step size.

But windows are applied twice: Once *before* FFT analysis because smoothing the signal eliminates some undesirable artifacts from the computed spectrum, and once *after* the IFFT to eliminate discontinuities. If we window twice, do the envelopes still sum to one? Well, no, but if we change the overlap to 75% (i.e. each window steps by 1/4 window length), then the sum of the windows is one![2]

[2]The proof is straightforward using trigonometric identities, in particular $\sin^2(t)+\cos^2(t)=1$. If you struggled to learn trig identities for high-school math class but never had any use for them, this proof will give you a great feeling of fulfillment.

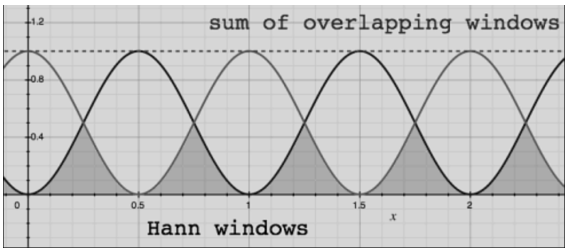

Figure 8.5: The raised cosine, Hann, or Hanning window, is named after the Mathematician Von Hann. This figure, from katja (www.katjaas.nl), shows multiple Hann windows with 50% overlap, which sum to 1. But that's not enough! In practice, we multiply by a smoothing window twice: once before the FFT and once after the IFFT. See the text on how to resolve this problem.

With windowing, we can now alter spectra more-or-less arbitrarily, then reconstruct the time domain signal, and the result will be smooth and pretty well behaved.

For example, a simple noise reduction technique begins by converting a signal to the frequency domain. Since noise has a broad spectrum, we expect the contribution of noise to the FFT to be a small magnitude at every frequency. Any magnitude that is below some threshold is likely to be "pure noise," so we set it to zero. Any magnitude above threshold is likely to be, at least in part, a signal we want to keep, and we can hope the signal is strong enough to mask any noise near the same frequency. We then simply convert these altered spectra back into the time domain. Most of the noise will be removed and most of the desired signal will be retained.

8.2 Spectral Processing

In this section, we describe how to use Nyquist to process spectra.

8.2.1 From Sound to Spectra

Nyquist has a built-in type for sound together with complex and rich interfaces, however, there is nothing like that in Nyquist for spectra, which are represented simply as arrays of floats representing *spectral frames*. Thus, we have to design an architecture for processing spectra in Nyquist (Figure 8.6). In this figure, data flows right-to-left. We first take input sounds, extract overlapping windows, and apply the FFT to these windows to get spectral frames. We then turn those spectral frames back into time domain frames and samples, and overlap add them to produce an output sound. The data chain goes from time domain to spectral domain and back to time domain.

In terms of control flow, everything is done in a lazy or demand-driven manner, which means we start with the output on the left. When we need some sound samples, we generate a request to the object providing samples. It generates a request for a spectral frame, which it needs to do an IFFT. The request goes to the FFT iterator, which pulls samples from the source sound, performs the FFT, and returns the spectrum to SND-IFFT.

Given this simple architecture, we can insert a spectral processing object between SND-IFFT and FFT iterator to alter the spectra before converting them back to the time domain. We could even chain multiple spectral processors in sequence.

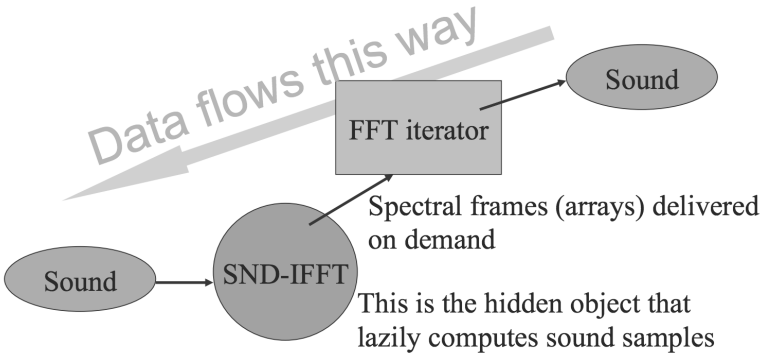

Figure 8.6: Spectral processing in Nyquist. Dependencies of sounds, unit generators, and objects are indicated by arrows. Computation is demand driven, and demand flows left-to-right following pointers to signal sources. Signal data flows right-to-left from signal sources to output signals after processing.

Nyquist represents the FFT iterator in Figure 8.6 as an object, but SAL does not support object-oriented programming, so Nyquist provides a procedural interface for SAL programmers.

The following is a simple template for spectral processing in SAL. You can find more extensive commented code in the "fftsal" extension of Nyquist (use the menu in the NyquistIDE to open the extension manager to find and install it). In particular, see comments in the files `lib/fftsal/spectral-process.lsp` and in `runtime/spectral-analysis.lsp`.

To get started, we use the `sa-init` function to return an object that will generate a sequence of FFT frames by analyzing an input audio file:

```
set sa = sa-init(input: "./rpd-cello.wav",
                 fft-dur: 4096 / 44100.0,
                 skip-period: 512 / 44100.0,
                 window: :hann)
```

Next, we create a spectral processing object that pulls frames as needed from `sa`, applies the function `processing-fn` to each spectrum, and returns the resulting spectrum. The two zeros passed to `sp-init` are additional state we have added here just for example. The `processing-fn` must take at least two parameters: the spectral analysis object `sa`, and the spectrum (array of floats) to be modified. You can have additional parameters for state that is preserved from one frame to the next. In this case, `processing-fn` will have `p1` and `p2`, corresponding in number to the two zeros passed to `sp-init`.

```
set sp = sp-init(sa, quote(processing-fn), 0, 0)
```

Since SAL does not have objects, but one might want object-like behaviors, the spectral processing system is carefully implemented with "stateful" object-oriented processing in mind. The idea is that we pass state variables into the processing function and the function returns the final values of the state variables so that they can be passed back to the function on its next invocation. The definition of `processing-fn` looks like this:

```
function processing-fn(sa, frame, p1, p2)
  begin
    ... Process frame here ...
    set frame[0] = 0.0  ; simple example: remove DC
```

```
  return list(frame, p1 + 1, p2 + length(frame))
end
```

In this case, note that `processing-fn` works with state represented by `p1` and `p2`. This state is passed into the function each time it is called. The function returns a *list* consisting of the altered spectral frame and the state values, which are saved and passed back on the next call. Here, we update `p1` to maintain a count of frames, and `p2` to maintain a count of samples processed so far. These values are not used here. A more realistic example using state variables is you might want to compute the spectral difference between each frame and the next. In that case, you could initialize `p1` to an array of zeros and in `processing-fn`, copy the elements of `frame` to `p1`. Then, `processing-fn` will be called with the current frame in `frame` as well as the previous frame in `p1`.

Getting back to our example, to run the spectral processing, you write the following:

```
play sp-to-sound(sp)
```

The `sp-to-sound` function takes a spectral processing object created by `sp-init`, calls it to produce a sequence of frames, converts the frames back to the time domain, applies a windowing function, and performs an overlap add, resulting in a sound.