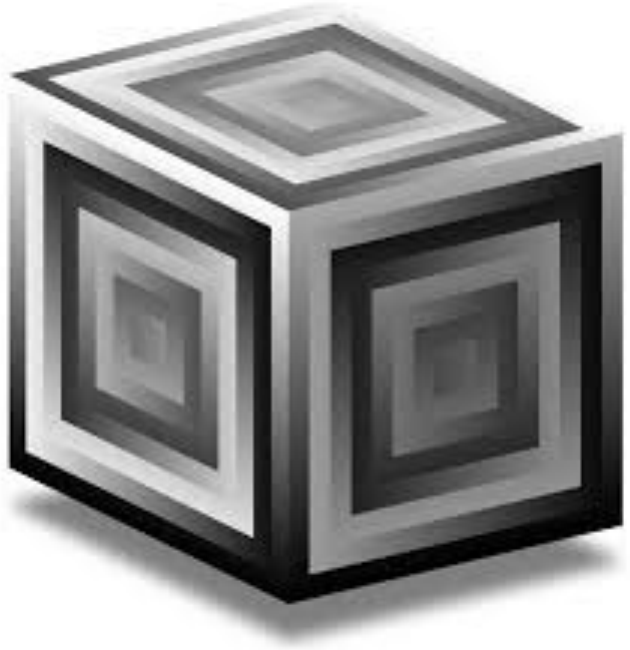
FFT in SuperCollider

Topics Addressed

- FFT with Signal
- The FFT and IFFT classes
- Phase Vocoding UGens



Computing FFT on `Signal`

- Recall that `Signal` is a class similar to a float array but used to hold samples.
- We can fill a signal with samples from a real signal or we can fill the signal using a variety of different methods (like `.sineFill`)
- The class has a method called .fft that requires three things:
  - A `Signal` of real numbers (i.e., your samples)
  - A `Signal` of complex numbers. The FFT allows for the input signal to be complex numbers. In regard to music, this will never be the case because our signals only consist of real numbers. Nevertheless, we need to provide a Signal of all zeroes.
  - A `Signal` that fills out part of a cosine table. This is used to efficiently calculate the FFT behind the scenes.

Computing FFT on Signal

```
~fft = {
        arg inputSignal, windowSize = 512;
        var complex, imag, cosTable;

        // Create a signal for the imaginary components of the input signal (required by fft)
        // Since we always use real signals, this will be an array of zeroes
        imag = Signal.newClear(windowSize);

        // Signal's fft requires a cosine table for efficient calculations
        cosTable = Signal.fftCosTable(windowSize);

        complex = inputSignal.fft(imag, cosTable);
        complex // Returns a complex number whose real part is a Signal and imag part is a Signal
};
```

Computing FFT on Signal

```
~postFrequencyBins = {
    arg complexFreqs, windowSize, sampleRate;
    var mags, counter, threshold;
    mags = complexFreqs.magnitude;
    threshold = 0.0001;
    counter = 0;
    while({counter < (windowSize/2 + 1)}, {
        if(mags[counter] > threshold, {
            "Frequency Bin: ".post;
            (counter/windowSize * sampleRate).post;
            ", Magnitude: ".post;
            (mags[counter]*2/windowSize).postln; // Note we need to scale by 2/N
            }, {}
         );
         counter = counter + 1;
    });
}
```

Working with FFT and IFFT

- Using the class `Signal` serves us well for analysis purposes of pre-recorded/generated samples.
- The classes `FFT` and `IFFT` allow for real-time frequency domain processing (via the Short-Time Fourier Transform) that can be used to produce powerful results.
  - See the *FFT Overview* from the SC doc guides.

FFT

- The class `FFT` accepts a buffer to store spectral data and an input signal to convert.
  - The buffer is usually a `LocalBuf`, essentially a buffer just for the `SynthDef` and not accessible by other synths. Think of it as the equivalent to a local variable. Could also use `Buffer`.
  - The input signal can be any number of input channels; however, you will need to provide buffers for the spectral data for every input channel of signal
- The class returns a signal (not the class `Signal`) of a constant -1, except when a new FFT window starts, in which case the size of window is returned.
  - This is how subsequent UGens know when the an FFT block has been fully placed in the buffer.
  - Remember the FFT information is in the buffer.
  - You can think of the chain as your variable to pass into other UGens so they know where and when to find the spectral data.
- FFT allows you to select different window types. We will stick with the default for real-time audio processing.

IFFT

- The class `IFFT` performs the inverse Short-Time Fourier Transform on the spectral data stored in the buffer passed into `FFT` and converts it to a time-domain audio signal.
- The `IFFT` requires the chain from the return value of the FFT in order to convert to the time domain.
- Generally `IFFT` is used at the end of series of processing UGens to convert the final result.

Simple FFT/IFFT Conversions

```
(
SynthDef(\noProcMonoFFT, {
    arg out = 0;
    var in, chain, sig;
    in = Saw.ar(440, 0.2);
    chain = FFT(LocalBuf(2048), in);
    // Convert to frequency domain and convert back
    // No processing here
    // You can add processing here before you convert back
    sig = IFFT(chain);
    Out.ar(out, sig);
}).add;
)
```

Mono Version

Stereo Version

```
(
SynthDef(\noProcStereoFFT, {
    arg out = 0;
    var in, chain, sig;
    in = Saw.ar([440, 440], 0.2); // Stereo signal
    // Need two buffers for spectral data!  Use function.
    chain = FFT(LocalBuf([2048, 2048]), in);
    // Convert to frequency domain and convert back
    // No processing here
    // You can add processing here before you convert back
    sig = IFFT(chain);
    Out.ar(out, sig);
}).add;
)
```

See Code Examples for Interesting FFT effects