Frequency Domain in the FXCore

Introduction

Most programs in the FXCore are concerned with the time domain, i.e. the length of a delay, how big a reverb sounds, etc. Even those concerned with frequency like a filter are still done in the time domain as it is the most straight forward way to deal with real time audio data and it is rare we really care about the exact amplitude of a specific frequency in audio effects. We normally care about a range of frequencies and add a filter to affect such range.

However there are times that being able to detect a specific frequency in audio is helpful, in most cases a simple bandpass filter will suffice and can be implemented using a state variable structure (see AN-7) or a second order all pass filter (see AN-8). Additionally, as these filters are run real time there is little to no delay in processing.

In cases where filters cannot be used due to the number of bands we need to examine and that some delay is acceptable we can look at converting the audio data to the frequency domain to examine it.

Fourier Transforms

When people begin to talk about working in the frequency domain you will often hear them mention the Fourier Transform, this is a mathematical operation that will decompose a signal into a series of sine waves at specific frequencies, phases and amplitudes that can summed together to reproduce the original signal. Note that the Fourier Transform (called FT from here on) actually can work in both directions; time domain to frequency domain as well as frequency domain to time domain. In this app note we are going to focus on the time to frequency domain aspect of the transform.

While the FT has great use it requires a very large number of multiplications and when used in a sampled system has additional requirements that can limit it. For a sampled system (sometimes called a discrete time environment or system) the basic FT equation looks like:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi jkn/N}$$

Equation 1

for 0 ≤ k≤ N−1, that is one ugly equation so let's break it down and see what is really going on. First the term $e^{-\frac{2\pi jkn}{N}}$ is based on Euler's identity:

$$e^{\frac{2\pi jkn}{N}} = cos\left(\frac{2\pi kn}{N}\right) + jsin\left(\frac{2\pi kn}{N}\right)$$

Equation 2

"j" represents the square root of -1.0 (note in math "i" is typically used but as we electronic engineers use "i" for current we use "*j*" for the square root of -1.0)

Now we can start to consider a few things, first as we are working with real, sampled data there is no imaginary component to it as a result we have no need for "j" and the "-" in the initial equation is actually unnecessary. If we substitute Equation 2 into Equation 1 and drop the "j" and the "-" we get:

$$X_k = \sum_{n=0}^{N-1} x_n * cos\left(\frac{2\pi kn}{N}\right) + \sum_{n=0}^{N-1} x_n * sin\left(\frac{2\pi kn}{N}\right)$$

Equation 3

We now have an equation that can be solved with normal math functions on a calculator, in a spread sheet or in a program. To those that say what I did was too simplistic and I skipped steps/proofs, oh well, we want solutions in an app note not 10 pages of proofs.

NOTE: The "+" between the cos summation and the sin summation is not really a simple addition, it is left over from representation of the values in a complex manner and will be addressed in a bit, for now just realize it is separator between the sin and cos summations.

To continue the breakdown of the equation, we need to understand the variables in it which are:
N : the number of samples we are going to evaluate
n : the sample number we are evaluating, ranges 0 to N-1
k : the "bin" (maps to frequency) we are testing for, ranges 0 to $\left(\frac{N}{2}\right)$-1
X : an array of the bins (so we address it as X_k)
x : an array of the samples (so we address it as x_n)

Those of you that have taken DSP courses in the past may recognize what is happening here, and that is that the FT is nothing more than correlation between the sampled data and a sine and cosine wave at the frequency of interest.

As we are summing values across a potentially large number of samples, we could end up with a very large result and this can be an issue in a fixed-point processor. As a result we will want to

scale the samples so that the final result is in the range the processor can handle. This is done by simply dividing each sample by N resulting in the equation:

$$X_k = \sum_{n=0}^{N-1} \frac{x_n}{N} * cos\left(\frac{2\pi kn}{N}\right) + \sum_{n=0}^{N-1} \frac{x_n}{N} * sin\left(\frac{2\pi kn}{N}\right)$$

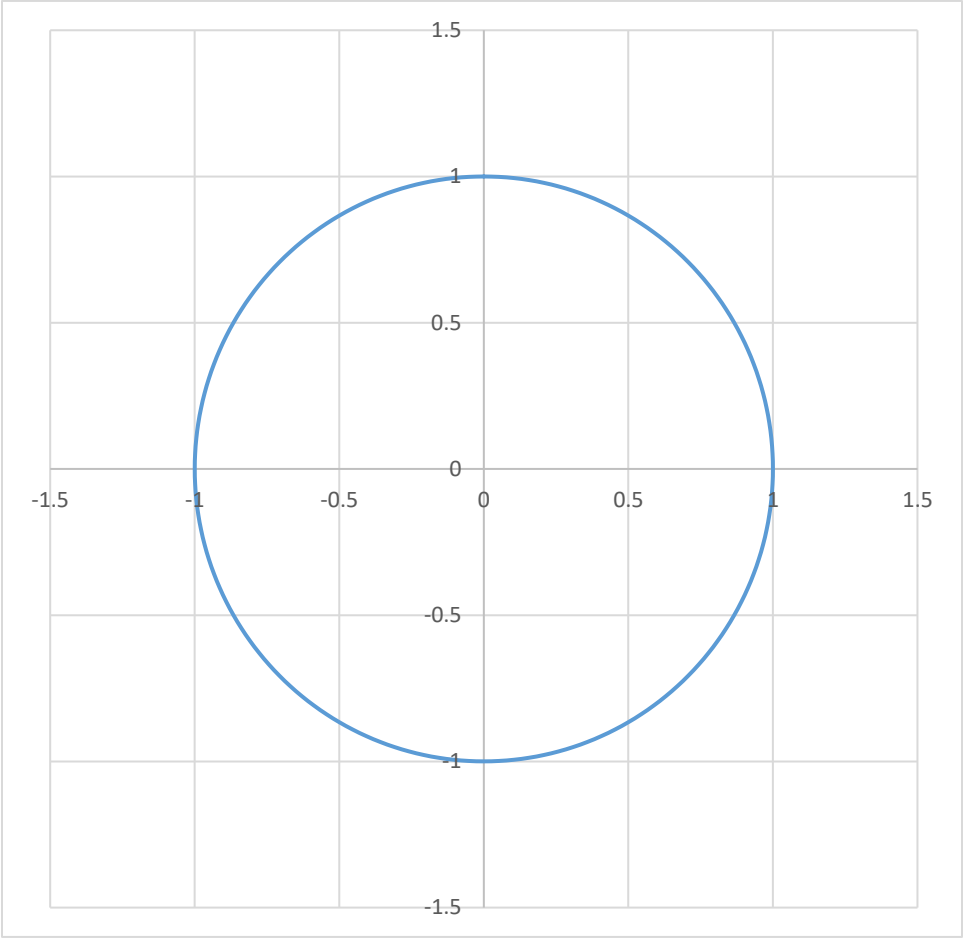
Equation 4

If we choose N to be a power of 2 then the division is a simple bit shift which is typically a very fast operation to perform in a DSP. This is why you see examples using sample length like 64 or 128.

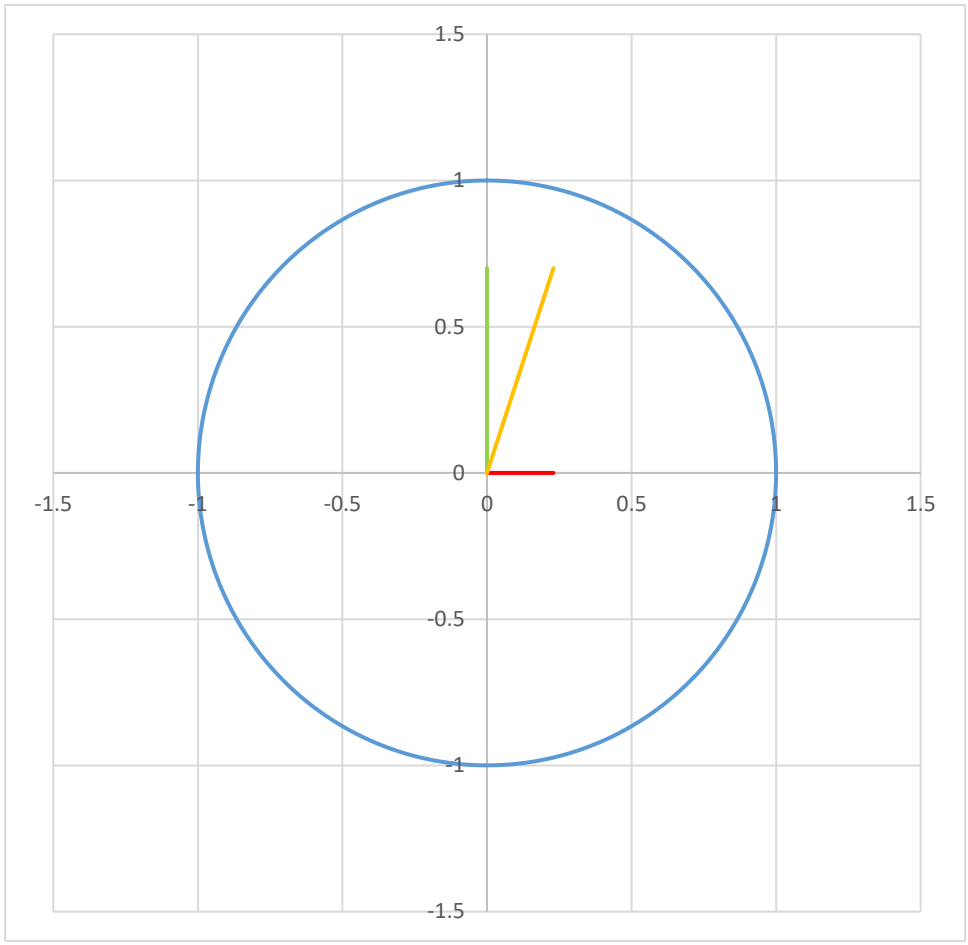
We have now created equations we can solve, scaled the sample data to remain in range of the processor but we still have that pesky “+” that is not really an addition, so what is it? What does it mean?

When we looked at Euler’s identity, Equation 2, we see the “+” between the *cos* part and the *jsin* part. This is not an addition but is used to both separate the “real” portion from the “imaginary” portion of a complex value as well as indicate that they are two parts of a single value. These two parts are also the X and Y coordinates of the value where cos is the X axis and sin is the Y axis.

At this point you may ask: “Why do we need both sin and cos? Isn’t one enough as they are both checking the same frequency?” And the answer to that is “no”. As we have no idea of the phase of the signal we are testing for in the sampled data we need to compare it to references we can use to determine the magnitude and phase. These references must have known phases and a fixed phase relationship of 90° (π/2 radians), using sin and cos satisfies both these requirements. If we look at a graph with cos as the X axis and sin as the Y axis as mentioned above and we make a circle with radius = 1 we see a graph like:



If after calculating the cos and sin summations for a specific X_k and were to plot them on the graph (just random values used here of cos = 0.23 and sin=0.7) it would look like:



:

Where the red bar along the horizontal axis is cos and the green bar along the vertical axis is sin we can calculate the magnitude of X_k from:

Magnitude for X_k = $\sqrt{cos^2 + sin^2}$

Which is drawn in orange. In this case it is equal to 0.737

Phase angle of X_k is a little more complex to calculate as it depends on both the cos and sin values, as we are really just interested in magnitude here we will ignore phase.

At this point we have cleaned up the equation to something that makes sense, defined the terms like N,k, etc. and shown how to calculate the magnitude for a specific bin in the FT so we can create a pseudo code example of implementing the FT:

```
; N = number of samples, we will use 128 here
; n = counter that goes from 0 to N-1
; k = counter that goes 0 to (N/2) - 1
; x[n] : an array of samples
; X[k] : an array to hold the calculated magnitude of each bin
```

```
For k = 0 to 63
      cosSum = 0
      sinSum = 0
      For n = 0 to 127
            cosSum = cosSum + (x[n]/128)*cos(2*pi()*k*n/128)
            sinSum = sinSum + (x[n]/128)*sin(2*pi()*k*n/128)
      next n
      X[k] = sqrt(cosSum*cosSum + sinSum*sinSum)
next k
```

You may notice that in Equation 1 we define k to go between 0 and N-1 but here we are limiting it to 0 to $\left(\frac{N}{2}\right)$-1, this is because the FT also includes negative frequencies and we are only interested in real, positive frequencies.

Now that we have the pseudo code for an FT we could easily write it in a real programming language but we still have some things to address, like what are the actual frequencies we are looking at? So far we are using indexes like k and n but no actual frequencies, well the frequencies depend on the sampling rate we are using and the length of the array containing the samples. As an example, assume we have a sampling rate of 32KHz and we are still using 128 samples for the transform, we need to have at least 1 full cycle of a sine wave in the samples to detect it so that means the lowest frequency we can detect is:

$$(32000\frac{samples}{second}) * (\frac{1\ cycle}{128\ samples}) = 250\ cycles/second = 250Hz$$

So the lowest frequency we can detect is 250Hz, the highest would be:

$$(N/2) * 250Hz = 64 * 250Hz = 16KHz$$

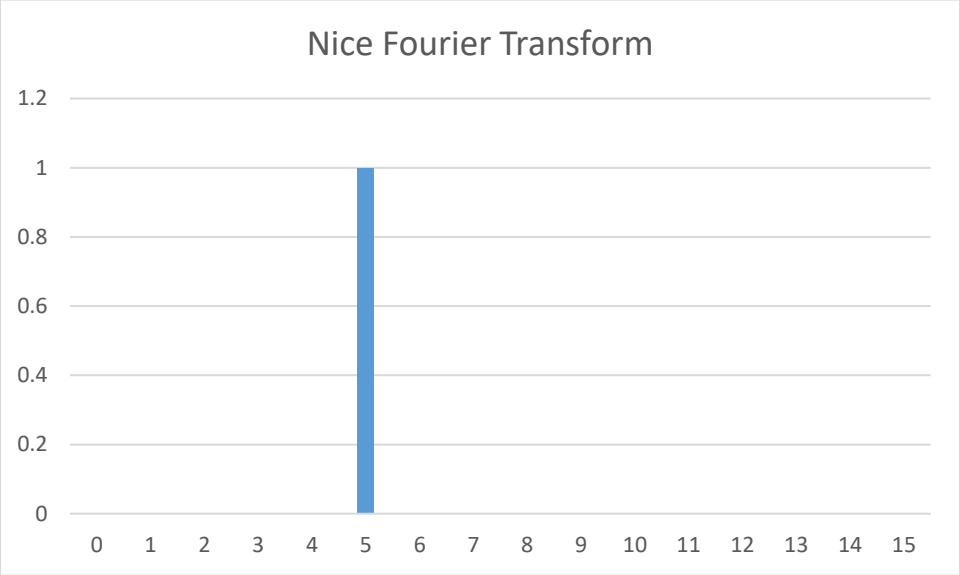
Let's make a few observations about the above pseudo code and frequency range:

1. The FT requires a very large number of multiplies
2. A sample length of 64 is too small as we really need to detect even lower frequencies in audio but to detect a lower frequency means increasing the number of samples which means even more multiplies. Additionally the more samples we collect prior to processing the longer our latency before calculating results.
3. Our frequency resolution is poor, examining the FT equations and pseudo code above we see that we are always looking at an integer multiple of the lowest frequency. In this example we would be examining 250Hz, 500Hz, 750Hz, etc.

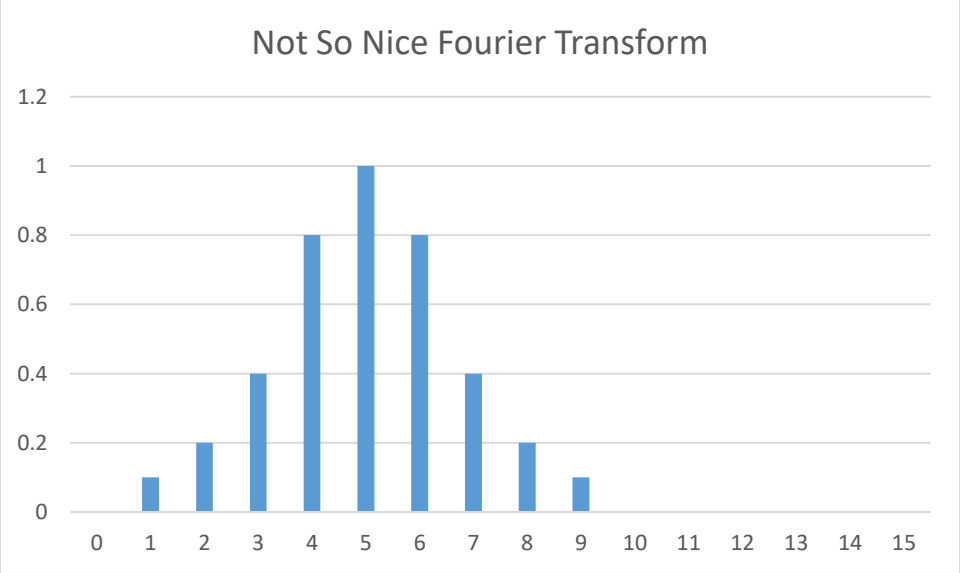
For items 1 and 2, there are tricks we can use to lower the impact of additional samples, but 3 is a real issue. Western music does not use integer multiples of a singular frequency, each increase in frequency is by a factor of $\sqrt[12]{2}$. By not using integer multiples of a singular frequency we create an entirely new set of issues.

Spectral Leakage

When we execute the FT on a set of samples and only check frequencies that are an integer multiple of the lowest frequency we can test we get nice, clean results with peaks at the frequencies that are in the sample data and near zero results at frequencies that are not. We have seen these nice looking graphs like:



But if we examine a frequency that is not an integer multiple of the lowest frequency we get graphs that look like:



If we are examining a single frequency why are we seeing peaks in the sidelobes? The FT assumes that there is some number of full cycles of the frequency being examined in the sampled data (it is a multiple of the lowest frequency that fits in the sampled data), basically if we “wrapped” around from the end of the sampled data back to the beginning the waveform

would perfectly connect and there would be no abrupt changes. If we are examining a frequency that does not have an exact number of wave forms in the sampled data (it is not a multiple of the lowest frequency) then we have an abrupt change in the waveform if we wrap around from the end of the sampled data to the beginning. To the FT this abrupt change is part of the signal so we see it as sidelobes to the bin that is closest to the frequency. There are things we can do to minimize these sidelobes like windowing the data but we can never eliminate them.

Windowing

“Windowing” the samples means to run them through a function that effects the amplitude of the samples. Basically we want to minimize or eliminate the discontinuity we see if we wrap around from the end of the samples to the beginning, we do this by using a function that smoothly increases from 0 to 1.0 and back down to 0 so the sample values at the start and end are 0 and the middle values are close to their original values. This has both positive and negative effects, while it helps decrease the sidelobes it does not eliminate them and it will cause the calculated amplitudes to have an error in them as the sample amplitudes have been changed. In many cases this is acceptable as we may not need to know the exact amplitude just which bins have the highest amplitudes. Each windowing function has advantages and disadvantages, in audio we often use the Hann (aka Hanning or raised cosine) window.

Sample Collection

In all the above we treated the samples as a set of N samples that we used to calculate the results for all k bins and if we were analyzing satellite data or information from a deep space probe we would do it exactly like that but we are looking at real time audio that is constantly changing as a user plays their instrument. Instead of capturing N samples then doing the calculations for k bins we can do the calculations as each sample comes into the FXCore and update the sin and cos summations accordingly. If we do this then the pseudocode from above becomes:

```
; N = number of samples, we will use 128 here
; n = counter that goes from 0 to N-1
; k = counter that goes 0 to (N/2) − 1
; X[k] : an array to hold the calculated magnitude of each bin
For k = 0 to 63
      cosSum = 0
      sinSum = 0
```

```
      For n = 0 to 127
            z = sample_in/128
            cosSum = cosSum + z*cos(2*pi()*k*n/128)
            sinSum = sinSum + z*sin(2*pi()*k*n/128)
      next n


      X[k] = sqrt(cosSum*cosSum + sinSum*sinSum)
next k
```

Instead of collecting 128 samples then analyzing them we simply do the calculations each time a new sample comes in. We do not need to store N samples just use N sequential samples.

Some may notice that for each bin k we are using a different set of 128 samples, this is perfectly fine in an audio application designed for real time. We basically care about what is happening “now” rather than “what were all the amplitudes in a specific set of samples”.

But we can see that latency may be an issue, if we are sampling at Fs=32K then it takes 64*128 sample periods to do all the bins, that is 8192 sample periods and at 32K is 256mS. That latency between updates may be too long, so to speed things up we can do multiple bins at the same time like:

```
; N = number of samples, we will use 128 here
; n = counter that goes from 0 to N-1
; k = counter that goes 0 to (N/2) - 1
; X[k] : an array to hold the calculated magnitude of each bin
For k = 0 to 63 step 4
      cosSum0 = 0
      sinSum0 = 0
      cosSum1 = 0
      sinSum1 = 0
      cosSum2 = 0
      sinSum2 = 0
      cosSum3 = 0
      sinSum3 = 0
      For n = 0 to 127
            z = sample_in/128
            cosSum0 = cosSum0 + z*cos(2*pi()*k*n/128)
            sinSum0 = sinSum0 + z*sin(2*pi()*k*n/128)
            cosSum1 = cosSum1 + z*cos(2*pi()*(k+1)*n/128)
            sinSum1 = sinSum1 + z*sin(2*pi()*(k+1)*n/128)
            cosSum2 = cosSum2 + z*cos(2*pi()*(k+2)*n/128)
            sinSum2 = sinSum2 + z*sin(2*pi()*(k+2)*n/128)
            cosSum3 = cosSum3 + z*cos(2*pi()*(k+3)*n/128)
            sinSum3 = sinSum3 + z*sin(2*pi()*(k+3)*n/128)
```

```
      next n
      X[k] = sqrt(cosSum0*cosSum0 + sinSum0*sinSum0)
      X[k+1] = sqrt(cosSum1*cosSum1 + sinSum1*sinSum1)
      X[k+2] = sqrt(cosSum2*cosSum2 + sinSum2*sinSum2)
      X[k+3] = sqrt(cosSum3*cosSum3 + sinSum3*sinSum3)
next k
```

We are now doing 4 bins at a time so the total time will be 64mS for all 64 bins.

Shifting bin center frequencies

In all of the above we have focused on each bin k being the result of an integer multiple of the lowest frequency we can detect. But we also recognized that the frequencies that interest us in audio are not integer multiples of a single base frequency and we will get spectral leakage as a result. Spectral leakage is not the only issue, we are also unsure of the actual frequency as we may see a peak in a specific bin but that does not mean the frequency is centered in that bin. For example, in the above we determined that for Fs=32K and N = 128 then the lowest frequency we can detect is 250Hz, the next bin would be 500Hz but in audio we may be interested in 440Hz.

If we input a signal and see a peak at 500Hz but also sidelobes with the ones at 250Hz and 750Hz being rather high we might guess it is 440Hz but we can't tell for sure. There are ways to increase the resolution of the data with zero padding but that will only increase the amount of processing it takes and will result in an unacceptably long latency. We could try curve fitting around the sidelobe peaks and attempt to find the peak of the fitted curve but that will also take lots of calculations and increase latency. As we will need to deal with sidelobes and non-multiples of a base frequency let's forget about the rule of integer multiples.

If we accept that the sidelobes are going to be there as we are interested in frequencies that are not integer multiples of a base frequency then why not at least change the center frequencies of the bins to something closer to the frequency of interest? If each bin is centered on a semitone then we can get a better idea if we are on pitch by the peak in the bin and the peak values of the sidelobes around it.

From the above pseudocode we see:

```
cosSum0 = cosSum0 + z*cos(2*pi()*k*n/128)
sinSum0 = sinSum0 + z*sin(2*pi()*k*n/128)
```

But what if k, as used in the equations, was non-integer? We could still use the integer k as an index into memory pointing to where we want to save the result but in the equation we could say "if integer k = 2 then equation k = 1.76" then we would be checking 440Hz instead of 500Hz. This will result in spectral leakage but we are going to have spectral leakage in any case as 440Hz is not an integer multiple of 250Hz so we can at least make it a little easier on us.

Putting it together

We have so far defined an equation for the FT we can solve, seen the issue on sample length and sample rate that will cause spectral leakage in our results and have a method to decrease these side lobes though it will affect amplitude calculation accuracy. We also came to the conclusion that if we have to live with spectral leakage we should make the entire thing easier on us and just set the bin center frequencies to semitones as that is what we care about in audio. All these things are done in the example code an-10.fxc, it operates across 42 semitones (open low E string on a guitar to fret 17 on the high E string), windows the samples using the

Hanning window, does 6 bins at a time and uses 512 samples per block. It has 36 triangle generators (one for each semi tone) where the amplitude for each generator is based on the extracted amplitude for the corresponding semitone. It uses lookup tables for finding sin and cos.

We are also using a look up table for the calculation of $\frac{2\pi kn}{N}$ for each bin because all the terms, except for “n”, are known in advance so they can be precomputed and included in the code therefor all we need to do is add a fixed value each sample period then use the result to look up the sin or cos in a table.

While we are minimizing things like the sidelobes they are still there so we hear the associated triangle generators for those bins. Additional processing of the bin amplitudes could be done, values could be sent to a micro over the second I2S channel for this additional processing and either sent on to a second FXCore or used for additional controls (i.e. send MIDI messages based on the bin and amplitude).

Conclusion

Frequency domain processing can be done on the FXCore but working with signals that are not integer multiples of a base frequency complicates the situation as it does for any DSP. Post processing of the data is going to be required and tradeoffs made between speed, number of bins, latency, etc.

Experimental Noize Inc. reserves the right to make changes to, or to discontinue availability of, any product or service without notice.

Experimental Noize Inc. assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using any Experimental Noize Inc. product or service. To minimize the risks associated with customer products or applications, customers should provide adequate design and operating safeguards.

Experimental Noize Inc. make no warranty, expressed or implied, of the fitness of any product or service for any particular application.

In no event shall Experimental Noize Inc. be liable for any direct, indirect, consequential, punitive, special or incidental damages including, without limitation, damages for loss and profits, business interruption, or loss of information arising out of the use or inability to use any product or document, even if Experimental Noize Inc. has been advised of the possibility of such damage.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Experimental Noize Inc. products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death ("Safety-Critical Applications"). Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Experimental Noize Inc. products are not designed nor intended for use in military or aerospace applications or environments. Experimental Noize Inc. products are not designed nor intended for use in automotive applications.

Experimental Noize Inc.

Scottsdale, AZ USA

www.xnoize.com

sales@xnoize.com