Sifft VST SOURCE Manual

Sample Inverse Fast Fourier Transform

http://www.xoxos.net

The precedent stages to this release are available separately -

http://www.xoxos.net/sem/samplemodules.zip

Sample0 to Sample4 are increasingly complex sources for a loop modulating wav player. Sample0 contains my simple yet effective script for opening wav files. The code is not particularly informed and can certainly be improved. Reviewing Sample0 will simplify review of the source for Sifft.

http://www.xoxos.net/sem/fft0.zip

This source demonstrates FFT analysis and resynthesis with cartesian to polar coordinate conversion (for processing as magnitude and phase). The method was adapted from dspguide.com and has a few improvements, eg. the use of a precomputed sine array instead of using transcendental functions. Do not attempt to proceed unless you have reviewed dspguide or a similar introduction to fourier transforms. Be aware that this method was selected for comprehension. I believe that there are FFT libraries that can halve or better the cpu performance of this method.

It may also be worth reviewing my pdf on dsp methods on the SEM page in order to recognise eg. my interpolation scripts.

What have I done?!

I truly despise reading other peoples' source code, and expect you will loathe mine.. my nasty, minimal variable names, my inlining. Comparatively, I expect you will find my methods rudimentary. I will intend to explain as best I can so that referencing the source can be kept to a minimum, as it is truly "knocked together".

Most variables are grouped and described in the header. If you are familiar with the algorithms, then you will eg. immediately recognise why FFT buffer variables are in pairs, because I use two overlapping windows et al.

If you are not familiar with eg. cubic interpolation, then that section of the script will not easily discretise itself.

First you will find the process functions.. there are two almost identical functions, one for play mode and one for scan mode. Pick through these last.

The next group of functions are common to all SynthEdit SEM sources and are fairly recogniseable to the VST SDK iirc.. eg. the description of pins. Following this, we have OnPlugStateChange, which includes the wav loading function, which you can ignore unless you wish to improve it to handle more types of wav files.

Following this we handle buffer length changes, and then the basic FFT function. As noted in the header file, my variable 'n' is the integer length of the buffer. Within the process I also use 'hb' which is half n, and 'nm1' which is n minus 1.

The last function in the file is my fftanalysis() which reads the data from *left and *right, performs the fourier analysis, and stores it in rl, il, rr and ir, which are obviously the real and imaginary data for left and right. The last loop in this function converts all the phase data but the first frame to the difference between it and the last frame.

This function features an embarassing mishandling of the length of the data - all the analysed frames are stored in one array. The length of this array is limited in my implementation by the 2^32 indexing cap - you can't index a larger array with a 32 bit INT variable because it can't have higher values ;) You will observe that there is almost no attempt to correctly gauge the maximum possible length given the buffer size. My brain just didn't want to do it. I'm a lazy person. I settled on the hacked 'cap' method because it crashed otherwise - possibly due to using signed 32 bit INTs, which halves the possible range. If you can devise a more accurate cap value, or another way to store the analysed frames, you should do so. I'm real new to handling lots of data and mallocs.

So up to this point we have analysed the sample and created arrays of the data. The rest is simple, but messy.

Onto the process function

Itfp, you will have to forgive me for using the terms 'frame' 'block' and 'buffer' interchangeably... these all mean n number of samples in the FFT window. I'm casual.

If you have previously implemented FFTs or reviewed the FFT0 source, you can anticipate what happens next: the process function detects gate events and then initiates sample playback. Playback is a bit tricky with FFT because you have to stagger two overlapping functions that assemble the data.. if you've done this with an FFT effect process, you get the idea. If you haven't.. build one first so you feel confident. A simple frequency or pitch shifter, or a simple filter/equaliser should only require a few lines of modification to the fft0 source.

You will observe I track the frame position with p0 (the pointer for the first buffer) and derive the value for p1 from it on every sample. I've never seen anyone else do this.. there may be better ways..

...Because this algorithm is a sampler and these buffers are stepped through at variable rates, I use a buffer to record the previous value of each pointer, p0 and p1, to detect "zero crossings" if you will of when the pointers wrap.. if you zoom down to line 335 you will note that the actual indexed pointer is a float value (bpf for 'buffer pointer float') and that p0 is derived from it.

...This is why you should definately build an FFT effect first.. because there are two incrementors here.. one for pitch, and one for time.... just smoke more sativa and do it your way, in steps.

When we detect that a frame/buffer/block/whatever has wrapped, we build the next block of samples for it. It is only then that we need to discern the index to the analysed data p ('pointer'). In the same way as the buffer pointer, this is derived from a float pointer 'pf' which is incremented by the time incrementor 'tinc'. Now we know which block of analysed data to use.

The entire algorithm is very simple, it is only expanded to 300 lines because of my inline duplication of functions to handle different cases. If there were only one case, you'd be looking at 50 lines.. so note the application of each case in switch statements (such as the almost useless 'block' option which doesn't interpolate between FFT frames).

You will note that 'o' is my all-purpose handy dandy float variable. In line 88 it is used to calculate the frequency shift coefficient.. in line 128 it is used to discern the fractional position between two frames..

...In lines 101-102 linear interpolation is used to find the 'phase difference' in the near-useless 'block' performance mode.. the equivalent to this in the normal mode (where the audio isn't overtly jumping from block to block) in lines 134-141 several interpolations must be made.. to find the values then find the values between these :) like bilinear interpolation.. which should be easy to reference if you've never used it.

Once we've used interpolation to determine the magnitude and increment the phase vector, we perform the reverse FFT to convert the data back to a time signal at lines 169-170. We're almost at the end of the mess for dealing with constructing one block of time data..

The first block is complicated by a function inside the IF statement at 172... normally with FFT resynthesis we'd just chug along constructing windowed blocks of time data, but we have a special case - say the sample is a snare drum and we're handling the first block.. we can't window it because then the sample would fade in, and there's no overlap, so I handled this by only windowing the 2nd half of the very first window.

You've probably discerned that the first block is written to the array ol0[] and or0[] ('out left,' 'out right') and ol1[] or1[] are for the "overlapping window".

...Then we deal with the case of the overlapping window.. then we handle the time domain waveform interpolation options, with all the boring checks to make sure the interpolate data isn't past the constraints of the array

The scan process is much simpler. I obviously copy/pasted and made a few quick edits.. there's no special case for the first block, only the buffer pointers are incremented, the time incrementor is removed (woops, it seems I didn't remove the function to calculate the time incrementor, I hope no one modulates it constantly in this mode as it doesn't do anything). I hope the digging has been worth it and makes you feel better about your own code :)