Signal Processing

This semester I have been manipulating WAV files on the computer to replicate sounds based on distortion theory and current guitar effects. I chose to use a programming language called Sonic [1] because I thought it would make programming easier due to it's setup 'designed' for signal processing. Basically one can create a Sonic program, use the included converter to change the code to C++ and then compile it with any C++ compiler. While Sonic is limited in some ways, I have been able to replicate many types of sound effects with the language. It took me a few weeks to tweak the program out (one of the Sonic libraries was buggy, and I had trouble acclimating my self to the language), but over the course of the semester I have been able to replicate and create examples of sound based on their algorithms.

I was first able to accomplish some interesting things with the software such as pitch shifting [this was really easy with Sonic], creating an ideal fuzz filter, removing the imaginary or real components of a WAV file, and cutting out selected frequencies of a WAV file. I was able to integrate these effects into one Sonic program in the end (pitch.s). Although it's pretty short it took me a while to get out all of the bugs, and I had to spend some time thinking about implementation.

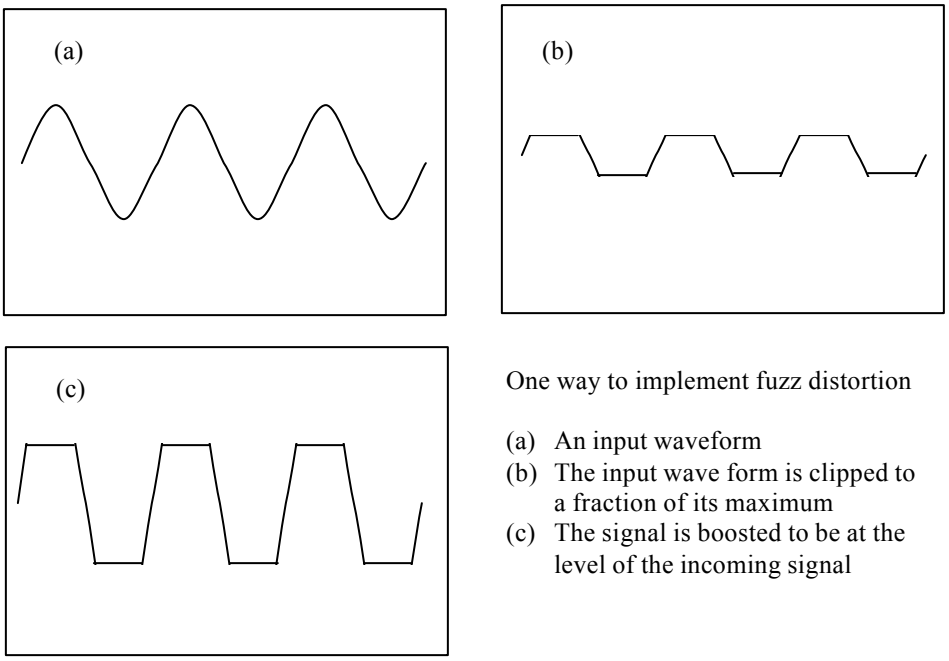
After that I spent some time creating WAV files that are based on Professor Steve Errede's "Theory of Distortion" lectures [2,3]. These include Quadratic and Cubic Non-Linear distortion (distortion.s), exponential distortion (expdist.s), along with intermodulation distortion (intermodulation.s). Then I proceeded to emulate some classic guitar effects such as ring modulation (ringmod.s), flanging (flange.s), and phasing (phase.s).

Included are descriptions of these effects and a few examples of what I have been able to do with Sonic to replicate those effects. They are of course more interesting when you can hear them, but I have made snapshots of the WAV files that I used as inputs and outputs, so that visible differences can be noticed.

All of the Sonic programs are listed on pages A1-A7. The WAV file snapshots are on pages B1-B16.

Fuzz Distortion

Basic "Fuzz Box" distortion is created when clipping an input waveform by limiting it's maximum amplitude. The end result is an output that retains some of the sound qualities of the original waveform but has extra odd harmonics added to it. The limit of maximum clipping on a sine wave for example would be a square wave. Overdriving an amplifier or using a circuit with diodes to clip the incoming signal are often the methods used to achieve this type of distortion. The program that I wrote, "pitch.s" (page A1), allows me to take an input signal and clip it at some fraction of the maximum amplitude. Sonic programs when compiled usually renormalize the output wave file, replicating a gain stage. The end result is a waveform that looks reminiscent of the original, but has higher odd harmonics (3 times the fundamental, 5 times, etc…) added. This gives the original waveform a harsher, but not unappealing, sound.



One way to implement fuzz distortion

(a) An input waveform
(b) The input wave form is clipped to a fraction of its maximum
(c) The signal is boosted to be at the level of the incoming signal

I printed out one example of this type of distortion applied. Page B1 of the appendix shows the unmodified WAV file "chord.wav". Page B2 shows the chord file clipped to a tenth of the maximum amplitude. Listening to the distorted WAV file one can easily tell that it is made from the original "chord.wav" file, but now has a pleasant sounding distortion.

Pitch Shifting

Pitch shifting in sonic was fairly simple because it is part of a built-in-function. I implemented this function into "pitch.s". The way it works is by taking the Fourier transform of the input waveform and then simply shifting the frequency scale. This type of effect is useful when one wants to create bass sounds from a guitar or to transpose an instrument to a different key.

On page B3, "Notes12.wav" is shown. Page B4 illustrates this WAV file pitch shifted up 25 Hz.

Other ventures in the frequency domain

In the frequency domain one thing I tried is to eliminate the imaginary frequency components of "Notes12.wav", the result is on page B5. One can hear a warbling to the output sound. Removing the real parts of the frequency components produced similar sounding results. I also tried creating an ideal low pass filter (page B6) and high pass filter (page B7) by cutting off frequencies above and below 250 Hz, respectively. I also used "pitch.s" here, but I had to change the cutoffs by hand (I couldn't figure out a way to pass a user parameter to the transfer function.)

Quadratic Distortion

The simplest linear output response O for an input stimulus S would simply be:

$$O(S) = K * S$$

where the magnitude of K is just a gain factor, and the phase of K changes the phase of the output.

The output response of a system with a quadratic non-linearity is [2]:

$$O(S) = K * (S + e\ S^2) = K * S * (1 + e\ S)$$

where S is the input signal and O is the output signal.

For a simple input $S(t) = A \cos(\omega t)$, we can write this function using trigonometric identities as:

$$O(t) = K A \cos(wt) + \tfrac{1}{2} e\ K A^2 + \tfrac{1}{2} e\ K A^2 \cos(2wt)$$

Thus the input signal will have a shift in its average value and also a second harmonic component dependent on the size of ε.

The program I wrote to add non-linear distortion to a WAV file is called "distortion.s" (page A2). On page B8, I show a 220 Hz sine wave (it was named 440 accidentally, but it is a 220 Hz sine wave). Examples of quadratic distortion with ε = .25 and ε = -.25 are shown below it [K = A = 1]. The altered waves clearly have a shifted average value (no longer zero) and also rounded on some peaks and sharper on others because of the non-linear response. When listening to the two distorted files, they sound exactly the same. The second harmonic and fundamental components are both still there in equal magnitude, the only difference is the second harmonic component is flipped over, so there is simply a change in the phase of the second harmonic component.

Cubic Distortion

The output response of a system with a cubic non-linearity is [2]:

$$\mathbf{O(S) = K * (S + \varepsilon S^3) = K * S * (1 + \varepsilon S^2)}$$

For a simple input S(t) = A cos (ωt), we can write this function using trigonometric identities as:

$$\mathbf{O(t) = K A (1 + \tfrac{3}{4} \varepsilon A^2) \cos(\omega t) + \tfrac{1}{4} \varepsilon K A^3 \cos(3\omega t)}$$

Thus the input signal will have a change in amplitude and also a third harmonic component dependent on the size of ε.

I once again used "distortion.s". On page B8, I show a 220 Hz sine wave (it was named 440 accidentally, but it is a 220 Hz sine wave). Examples of cubic distortion with ε = .25 and ε = -.25 are shown on the page [K = A = 1]. Notice that the ε = .25 cubic distorted wave has sharp peaks, while the ε = -.25 have rounded peaks. It is very clear that these waveforms are going to sound different! Unlike the quadratic distortion case, the sign change does not only induce a phase shift in the higher order harmonics, but also changes the amplitude of the fundamental component. When listening to the two distorted files, the third harmonic is louder (just barely but one can tell) in the ε = -.25 file since the relative amplitudes of the fundamental and third harmonic are closer together.

I also show a special case on page B8, when $\varepsilon = -4/3$. In the interesting case where A = 1 as well, the fundamental component completely disappears:

$$O(t) = K\left(1 + \tfrac{3}{4} * -4/3\right)\cos(\omega t) + \tfrac{1}{4} * -4/3\ K\cos(3\omega t)$$

$$O(t) = -1/3\ K\cos(3\omega t)$$

In fact looking at the output WAV file, indeed only the third harmonic component remains when $\varepsilon = -4/3$.

Exponential Distortion

The output response of a system with an exponentially growing non-linearity is [2]:

$$O(S) = K * (e^{a|S|} - 1) \qquad \text{for } S \geq 0$$

$$O(S) = -K * (e^{a|S|} - 1) \qquad \text{for } S < 0$$

while the output response of a system with a exponentially decaying non-linearity is [2]:

$$O(S) = K * (1 - e^{-a|S|}) \qquad \text{for } S \geq 0$$

$$O(S) = -K * (1 - e^{-a|S|}) \qquad \text{for } S < 0$$

$\alpha > 0$ in both exponentially-growing and exponentially-decaying cases.

Looking on page B9, a 220 Hz sine wave has exponentially-growing and exponentially-decaying non-linearities applied [α = K = A = 1]. After renormalization, the sound of wave file with the exponential-growing non-linearity seems to have less evident higher harmonic content than the exponential-decaying non-linearity.

The program I wrote to add exponential distortion to a WAV file is called "expdist.s" (page A3). On pages B10 and B11, I applied these distortions to the "chord.wav" file. I used a large value of α = 5 [K = A = 1]. The respective sharpening and rounding are very evident here. On page B11 one can particularly notice that after renormalization, that small amplitudes are quenched by a exponentially-growing non-linearity (sound starts to crackle) while small amplitudes are enhanced by a exponentially-decaying non-linearity (sustain increased).

Intermodulation Distortion

Two sinusoidal tones of different frequencies simply combine by addition [3]:

$$S = S_1 + S_2 = A_1 \cos(w_1 t) + A_2 \cos(w_2 t)$$

If a completely linear output response O is applied to S then:

$$O(S) = K * S = K * (S_1 + S_2) = O(S_1) + O(S_2)$$

The program I wrote to create and add non-linear distortions to two sin waves is called "intermodulation.s" (page A4). Examples of combined waves are shown on page B12. The first wave is a plain normalized 1000 Hz sinusoidal wave. The second has this wave combined with a 950 Hz wave and renormalized. Notice that when the two frequencies are close together there are beats formed. When the 1000 Hz wave is combined with a 150 Hz wave, then the 1000 Hz wave is amplitude modulated along the form of the 150 Hz wave.

If we now apply quadratic distortion, the overall output response is:

$$O(t) = K(A_1 \cos(w_1 t) + A_2 \cos(w_2 t)) + \varepsilon K[A_1 \cos(w_1 t) + A_2 \cos(w_2 t)]^2$$

Once again we can apply trigonometric identities and realize an interesting result:

$$O(t) = K(A_1 \cos(w_1 t) + A_2 \cos(w_2 t)) + \tfrac{1}{2}\varepsilon K A_1^2 + \tfrac{1}{2}\varepsilon K A_2^2 + \tfrac{1}{2}\varepsilon K A_1^2 \cos(2w_1 t) + \tfrac{1}{2}\varepsilon K A_2^2 \cos(2w_2 t) + \varepsilon K A_1 A_2 [\cos((w_1 - w_2) t) + \cos((w_1 + w_2) t)]$$

In fact there is now not only second harmonics of each input wave frequency, but there are also components of the final waveform at the sum and difference frequencies. This component is called intermodulation distortion and results from the non-linear mixing of the two waveforms under distortion.

I again used "intermodulation.s". On page B13, I compare clean mixed signals to distorted ones. For $\varepsilon = .25$ [K = A = 1], the resulting output looks similar to the original, but the lower part of the signal is scrunched. The distorted sounds have more buzz due to the additional frequency components.

If we apply cubic distortion, the overall output response is:

$$O(t) = K(A_1 \cos(w_1 t) + A_2 \cos(w_2 t)) + \varepsilon K[A_1 \cos(w_1 t) + A_2 \cos(w_2 t)]^3$$

Again we can apply trigonometric identities:

$$O(t) = K A_1 [1 + \tfrac{3}{2} e (\tfrac{1}{2} A_1^2 + A_2^2)] \cos(w_1 t) + K A_2 [1 + \tfrac{3}{2} e (\tfrac{1}{2} A_2^2 + A_1^2)] \cos(w_2 t) + \tfrac{1}{4} e K A_1^3 \cos(3w_1 t) + 1/4 e K A_2^3 \cos(3w_2 t) + \tfrac{3}{4} e K A_1^2 A_2 [\cos((2w_1 - w_2) t) + \cos((2w_1 + w_2) t)] + \tfrac{3}{4} e K A_1 A_2^2 [\cos((w_1 - 2w_2) t) + \cos((w_1 + 2w_2) t)]$$

Once again there are not only third harmonics of each input wave frequency, but there are also components of the final waveform at certain sum and difference frequencies.

I again used "intermodulation.s". On page B14, I compare clean mixed signals to cubic-distorted ones. For $\varepsilon = 1$ [K = A = 1], the resulting output has an envelope that seems to be made up of straight lines rather than curved. For $\varepsilon = -1$ [K = A = 1], the resulting output has it's peaks squashed down. Similarly to cubic distortion as we saw earlier with A = 1, an ε value of 1 has a greater amount of the fundamental than for ε value of -1. The distorted sounds again have more buzz due to the additional frequency components, with the third harmonic and sum and difference frequencies more evident for $\varepsilon = -1$.

Ring Modulation

Ring Modulation [4] is simply a multiplication of two signals together to make the output signal.

$$out(t) = in_1(t) * in_2(t)$$

Usually one of the inputs is a simple sine wave and the other is the sound to be ring modulated.

$$out(t) = in(t) * \sin(2\pi ft)$$

The program I wrote to ring modulate an input WAV file is called "ringmod.s" (page A5). The WAV files I ring modulated were "220.wav" and "Notes12.wav" and can be seen on page B15.

The output tones produced for a simple input wave (say a sine wave) are the sum and difference of the original frequencies.

$$\mathbf{out(\ t\ ) = \sin(\ 2\pi\ f_1 t\ ) * \sin(\ 2\pi\ f_2 t\ )}$$

$$\mathbf{out(\ t\ ) = \tfrac{1}{2}\ [\ \cos(\ 2\pi\ [f_1 - f_2]t\ ) - \cos(\ 2\pi\ [f_1 + f_2]t\ )\ ]}$$

If f_1 = 220Hz & f_2 = $73^1/_3$Hz then out(t) will have $146^2/_3$Hz and $293^1/_3$Hz signals. (example: 220-73.wav, page B15)

If f_1 = 220Hz & f_2 = 220Hz then out(t) will have 0Hz ["DC"-type] and 440Hz signals. (example: 220-220.wav, page B15)

Flanging

Flanging [5] is an effect created by mixing a signal with a slightly delayed copy of itself, where the length of the delay is constantly changing. It was discovered when a tape machine was being used for a delay and someone touched the rim of a tape reel, changing the pitch. With some more tinkering and mixing of signals, the characteristic flanging sound was created. The rim of the reel was known as the flange, thus the name flanging.

The process of adding the delayed copy of the signal to the original is basically what is called a comb filter.

The typical type of comb filter looks like:

$$\mathbf{Out[\ t\ ] = In[\ t\ ] + A * In\ [\ t - F(t)]}$$

Although I used this type of response for the output in my program [basically the same]:

$$\mathbf{Out[\ t\ ] = In[\ t\ ] + A * In\ [\ t + F(t)]}$$

Where F(t) is the delay time and A is the mix factor (A = 1 means the amplitude of the delayed signal is the same as the original signal).

The average delay time (D) is varied by a low frequency oscillator (LFO). Typical LFO's include sinusoidal waves and triangle waves. So generally F(t) is defined as:

$$F(t) = D + LFO$$

In my program I used a cosine wave as my LFO so:

$$F(t) = D + S * \cos(2\pi ft)$$

where S is the amplitude of the LFO and f is the frequency of the LFO

A comb filter creates notches in the frequency response of the output. For any given amount of delay, some frequencies will be passed through and others will not. This can be seen by taking a look at the comb filter in frequency space [6]. Assuming A=1:

$$Out[\,t\,] = In[\,t\,] + In\,[\,t - F(t)]$$

$$Out[\,s\,] = In\,[\,s\,] + e^{sF(t)} * In\,[\,s\,] \qquad s = i\omega$$

$$Out[\,s\,] = (1 + e^{sF(t)}) * In\,[\,s\,] = H\,[s] * In\,[s]$$

$$H[\,s\,] = e^{sF(t)/2}\,(e^{-sF(t)/2} + e^{sF(t)/2})$$

$$H[\,i\omega\,] = e^{i\omega f(t)/2}\,2\cos(\omega F(t)/2)$$

$$|\,H[\,i\omega\,]\,| = |\,2\cos(\omega F(t)/2)\,| = |\,2\cos(2\pi f' F(t)/2)\,|$$

For example if the delay time was 1ms, then frequencies of f'= 500 Hz, 1500 Hz, 2500 Hz, etc would be eliminated in the output response. The name comb filter comes from the comb shape of the graph of $|\,H[\,i\omega\,]\,|$.

The program I wrote to flange an input WAV file is called "flange.s" (page A6). Page B16 shows examples of flanging at D = 3.5 ms [a typical flanging delay] and D = 1ms [I'll use this to compare phasing to flanging].

Phasing

Phasing is similar to flanging, but uses an allpass filtered input signal together with the original input signal to crate an output. An allpass filter passes each frequency unaltered, but causes changes in the phase of the original signal.

An allpass filter in the time domain looks like [7]:

$$\mathbf{Out(\ t\ ) = -A * In(\ t\ ) + (1 - A*A) * (\ In(\ t - F(t)\ ) + A * Out(\ t - F(t)\ )\ )}$$

Where F(t) is once again

$$\mathbf{F(t) = D + S * \cos(\ 2\pi\ ft\ )}$$

Notice that the (In(t - F(t)) + A * Out(t - F(t))) part is basically the comb filter we saw earlier. So the final output, R, is simply

$$\mathbf{R(\ t\ ) = Out(\ t\ ) + In(\ t - F(t)\ )}$$

Similar to a comb filter, there are notches in the frequency response, but they are usually limited and not in the comb-like structure of the comb filter, which creates an effect that is subtler than the flanger [8].

The program I wrote to phase an input WAV file is called “phase.s” (page A7). Page B16 shows examples of both flanging and phasing at D = 1ms and S = 0.5 ms. While the phaser effect definitely has a different look from the flanger effect and the original waveform, it sounds less intrusive than the flanging effect. There is actually a noticeable difference between flanging and phasing here.

Other Effects

Many types of effects such as chorus, simple echo, reverberation, slap echo, and delay can be built from comb and allpass filters. In fact chorus and simple echo can easily be achieved using a flanger (comb filter) but with a delay time of about 5 to 20 ms for chorus and greater than 30 ms for a simple echo [9]. I was able to replicate these sounds using values of D = 20 ms and 200 ms respectively.

Other Thoughts

I learned a lot by completing this project and I think it is nice that there are accompanying WAV files so that the sounds produced can be heard as well as seen. The WAV files are currently on the Water_drop computer in 364 Loomis, but will hopefully be moved to the Physics 398 Home Page [10] and possibly changed to MP3 Format. Thanks to Steve Clayton for his helpful ideas and assistance and to Steve Errede for his suggestion to try replicating sounds based on the "Theory of Distortion" lecture notes (which kept me going on this project) and of course for offering this course.

REFERENCES

[1] "Sonic - a Digital Audio Programming Language", Don Cross
(http://www.intersrv.com/~dcross/sonic/)

[2] "Theory of Distortion I", Steve Errede
(http://webug.physics.uiuc.edu/courses/phys398/fall00/Lecture_Notes/Distortion/PDF_FIles/Theory_of_Distortion1.pdf)

[3] "Theory of Distortion II", Steve Errede
(http://webug.physics.uiuc.edu/courses/phys398/fall00/Lecture_Notes/Distortion/PDF_FIles/Theory_of_Distortion2.pdf)

[4] "Audio Effects: Frequently Asked Questions", Mike Currington
(http://www.harmony-central.com/Effects/audio-effects-faq-10.txt)

[5] "Flanging", Scott Lehman
(http://www.harmony-central.com/Effects/Articles/Flanging/)

[6] "Derivation of the Flanger's Frequency Response", Scott Lehman
(http://www.harmony-central.com/Effects/Articles/Flanging/derivation.html)

[7] "EMS News", Brett Terry (Editor)
(http://ems.music.uiuc.edu/news/fall94/news.html)

[8] "Sound Processing and Effects", Sampo Syreeni
(http://www.helsinki.fi/~ssyreeni/dsound/dsndc08)

[9] "Time-related sound processors", Heidrun Speckmann
(http://www.dwelle.de/rtc/infotheque/sound_processors/soundprocessors.html)

[10] "UIUC Physics 398 PEMI Homepage", Steve Errede
(http://webug.physics.uiuc.edu/courses/phys398/)