# Dr. dB: The Digital Guitar Amplifier
# EGR 423 Final Project

Dustin Matthews and Lucas VanAssen

EGR 423

04/22/2024

Professor Dunne

## Table of Contents

Table of Figures ..... 3
Objectives ..... 4
Part 1: Background ..... 4
Part 2: Design Overview ..... 5
Part 3: Input Filtering ..... 6
Part 4: Distortion Stage ..... 6
Part 5: Tonestack ..... 11
Part 6: Modulation Effects ..... 13
Part 6.a: Reverb ..... 13
Part 6.b: Phaser ..... 14
Part 7: Output ..... 15
Part 8: GUI ..... 15
Conclusion ..... 16
References ..... 16
Code ..... 18
CCS Files Run on the Zoom Kit ..... 18
inputBPF.c ..... 18
distortionModule.c ..... 18
toneStack.c ..... 19
verbBuf.h ..... 21
verbBuf.c ..... 21
Main.c ..... 22
ISRs.c ..... 23
EQknobs.gel ..... 25
Scripts used for LUT generation, filter generation, and plots for the report ..... 26
bandpass.m ..... 26
Lowhi_cascade.m ..... 27
LUTdistortion.m ..... 30
distortionEffects.m ..... 31
distLUTgenerator.m ..... 33
reverbTest.m ..... 35
reverbPlots.m ..... 36

phaserTest.m ..... 37

## Table of Figures

Figure 1 Basic Modern Guitar Amplifier Block Diagram [3] ..... 4
Figure 2 Dr. dB Block Diagram ..... 5
Figure 3 Input Filter Response ..... 6
Figure 4 Tube Transfer Curve [6] ..... 7
Figure 5 Boss SD-1 Asymmetrical Clipping Circuit (https://www.pedalpcb.com/product/uberdrive/) ..... 7
Figure 6 ATAN and TANH Curves Used in the Distortion LUT ..... 8
Figure 7 Asymmetrical Distortion Curve ..... 8
Figure 8 Example of the Asymmetrical Distortion Algorithm Applied to a Sine Wave ..... 9
Figure 9 Output of Distortion Stage 1 Processed by Distortion Stage 2 ..... 9
Figure 10 Asymmetrical Distortion Algorithm (distortionModule.c) ..... 10
Figure 11 Measured Output (CH1) vs. Input (CH2) When Distortion Bypassed ..... 10
Figure 12 Measured Output (CH1) vs. Input (CH2) When Distortion at Minimum Setting ..... 11
Figure 13 Measured Output (CH1) vs. Input (CH2) When Distortion at Maximum Setting ..... 11
Figure 14 Fender Tonestack [2] ..... 11
Figure 15 Filter Design Script ..... 12
Figure 16 Overall Tonestack Response ..... 13
Figure 17 Example of the Reverb Algorithm Applied to a Sine Wave ..... 14
Figure 18 Reverb Implementation (verbBuf.c) ..... 14
Figure 19 LFO Plot [1] ..... 14
Figure 20 Example of the Phaser Algorithm Applied to a Sine Wave ..... 15
Figure 21 Dr. dB User Interface ..... 16

## Objectives

The objective of this project is to incorporate many of the concepts learned in EGR 423, as well as concepts from external research and general know-how, into a cohesive, functional processing program that digitally simulates a guitar amplifier with tone controls, variable distortion, and frequency modulation effects. The result is Dr. dB, the digital guitar amplifier.

## Part 1: Background

1. The following figure shows a simplified block diagram of the various stages found in a modern guitar amplifier. This was used as a general guidepost for determining some of the various toneshaping stages that would be implemented in Dr. dB.

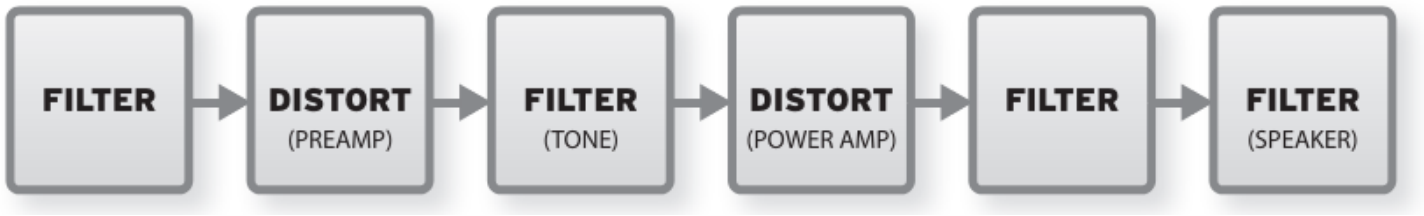


*Figure 1 Basic Modern Guitar Amplifier Block Diagram [3]*

Earlier in the semester, there were aspirations to take measurements of a specific guitar amplifier and create a basic model of it in software, but further research indicated that this endeavor was beyond the scope this project, and it was not feasible to attempt this in the time allotted. It was decided that for this project, a generic pre-amplifier design would be developed, which fits well with the limitations of the Zoom board used in this class.
In a typical guitar amplifier with an 'overdrive' or 'distortion' channel, the pre-amplifier stage is where the bulk of the distortion occurs. This channel I is also typically equipped as well with a 3-band EQ, usually labeled as some variation of 'Bass', 'Middle', and 'Treble'. Following this is the power amplification and any additional filtering before the signal is transmitted to the speaker. For this project, the design focuses on the pre-amplification region of the amplifier, as this is where the meat of the amplifier's toneshaping occurs.

2. A general note on distortion: as the reader may be aware, one form of distortion, particularly that flavor which is found in the guitar world, is the addition of harmonic content to a signal. Harmonics are integer multiples of a signal's fundamental frequency. Guitar amplifiers and effects pedals intentionally introduce varying amounts of distortion, depending on the topology of the circuit. This can be accomplished with a variety of components, such as diodes, transistors, or vacuum tubes.

## Part 2: Design Overview

1. The major elements of the design and the overall signal flow are shown in the block diagram in Figure 2. These will be briefly described below before diving into the detailed design.

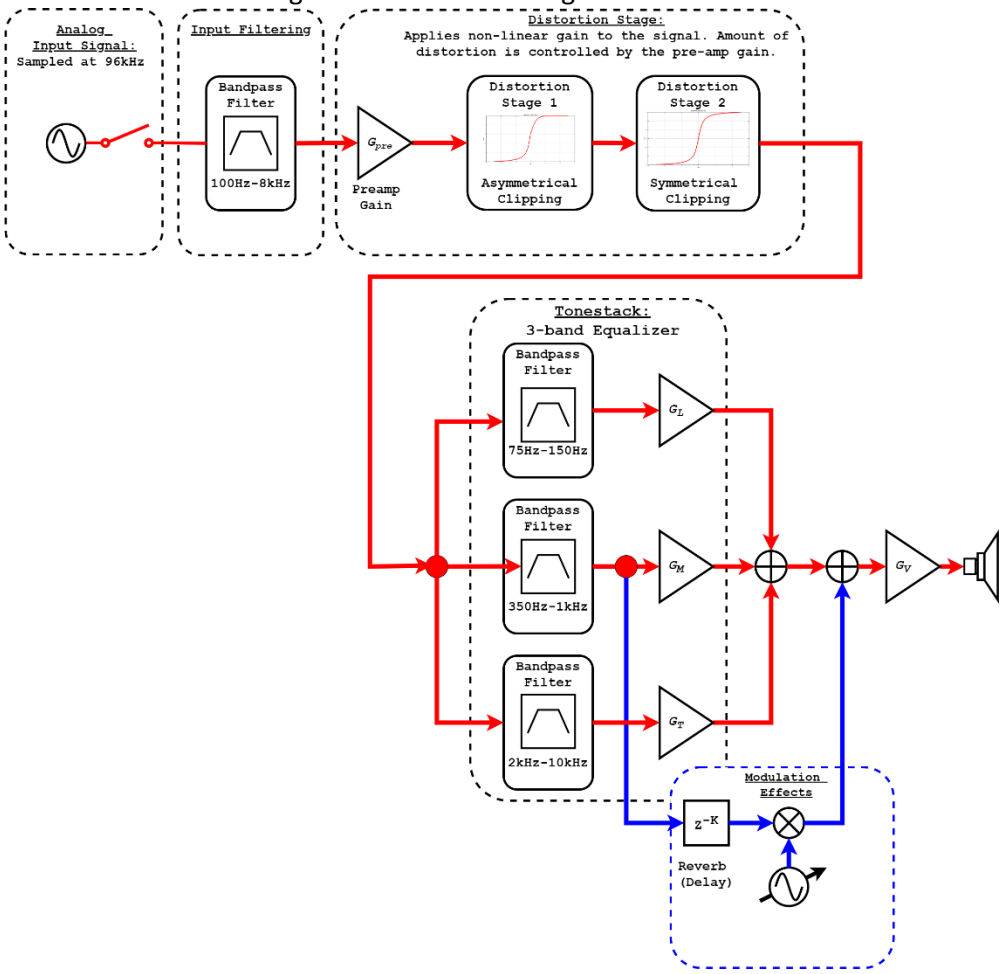


*Figure 2 Dr. dB Block Diagram*

1.1. Research indicated that, with digital guitar effects, it is common practice to oversample the signal ([2], page 7). Early experimentation with the design resulted in a lot of extra 'hiss' in the output, despite the presence of filtering. Given the capabilities of the Zoom board, it was decided to pursue a 96kHz sample rate with the intent to reduce any possible aliasing effects.

1.2. The first processing stage is a simple bandpass filter, meant to limit the frequency band of the processed signal.

1.3. Following the input filter is the distortion stage, where the amount of distortion is user controllable by varying the pre-amp gain.

1.4. The distorted signal is run through a 3-band 'tonestack', which is a set of three parallel bandpass filters, which have user controllable output gain and provide a great deal of control over the sound.

1.5. Following the tonestack is a set of optional modulation effects. This consists of the 'reverb', which is essentially an attenuated and delayed version of the Mid Frequency filter. This can be mixed with a phaser effect if the user chooses, which multiplies the delayed signal with a low frequency oscillator. The LFO frequency is user-configurable, with integer options from 5-10Hz.

## Part 3: Input Filtering

1. The guitar does not, of course, generate clean, single frequency sinusoids when it is played. It generates a great deal of harmonic content, with the bulk of the frequency range spanning from roughly 82Hz up to 5kHz [5]. One additional tip that was found through research is that reducing frequencies above 10kHz can reduce any excessive harshness in a guitar's tone [5]. The harmonic content of the guitar can reach up to 15 kHz, but to limit the amount of additional frequency content and keep the signal free from any potential noise or aliasing prior to distortion, the input filtering was designed with a passband from 100Hz – 8kHz.
    1.1. The digital filter was designed in Octave, which is a similar software to MATLAB, and is free. After experiencing stability issues with IIR bandpass filters in a previous lab and throughout the earlier stages of development of this project, all bandpass filters in the design were created by cascading low and high pass IIR filters, which have proved to be more stable, and did not seem to require significantly more processing time.
2. The frequency response of the two Butterworth filters comprising the input filter stage are shown in the plot below.

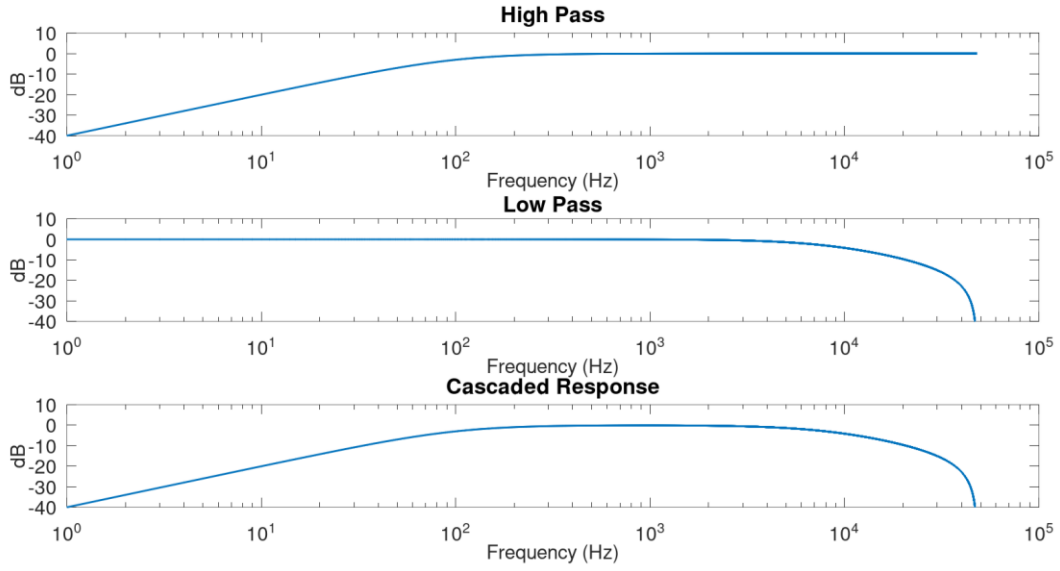


*Figure 3 Input Filter Response*

## Part 4: Distortion Stage

1. In guitar amplifiers, distortion is a non-linear effect. To paint with a broad brush, the harder you drive the input, the harder the amplifier will distort. Research indicated that one feasible method to simulate distortion is through a common waveshaping technique, which used trigonometric function such as tanh(x) and arctan(x) ([2], page 10). The curves of these trigonometric functions resemble a reflected version of some vacuum tubes' or diodes' typical voltage/current curves, which are shown in Figure 4.

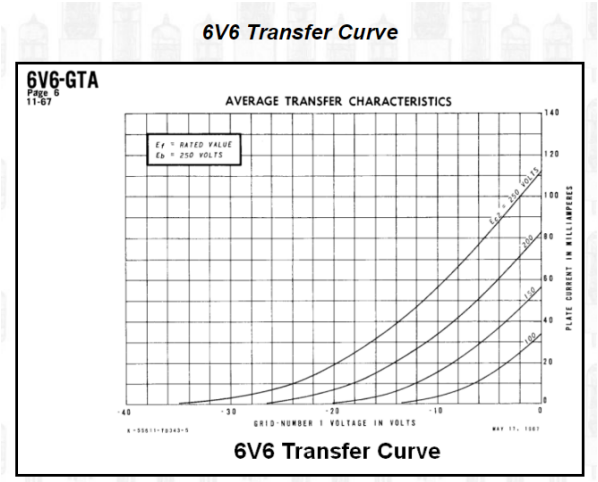


*Figure 4 Tube Transfer Curve [6]*

2. Another property of some guitar equipment is asymmetrical clipping, which gives 'softer' distortion effect, often referred to as 'overdrive'. One well-known example of this is the feedback loop of a Boss SD-1 guitar pedal with an asymmetrical set of diodes for clipping, as shown in Figure 5

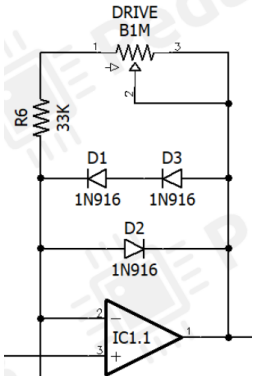


*Figure 5 Boss SD-1 Asymmetrical Clipping Circuit (https://www.pedalpcb.com/product/uberdrive/)*

It should be noted that the distinctions between different types of distortion as described by guitarists are often a little blurry, and there is not a quantitative limit that separates overdrive from what might typically be referred to as 'modern' distortion, or fuzz, which is an even 'harder' distortion. Generally speaking, the different types of distortion involve varying amounts and degrees of harmonic content being added to the signal, such as, for example, a circuit resulting in more odd harmonics or even harmonics.

3. Given that this program is running with a 96kHz sample rate, there is a limited amount of processing time available. Rather than use potentially costly trigonometric calculations, it was decided to generate a set of lookup tables that could be used to process the input signal. Given the maximum amplitude of the input signal is capped at 32767, the algorithm pursued uses a simple bit shift to scale the input down to a number between 0 and 1023, which is the size of the lookup tables. This bitshifted value becomes the selected index, and the lookup table output is multiplied by a scalar that leaves sufficient headroom for further processing. The two curves representing the lookup table values are shown below:

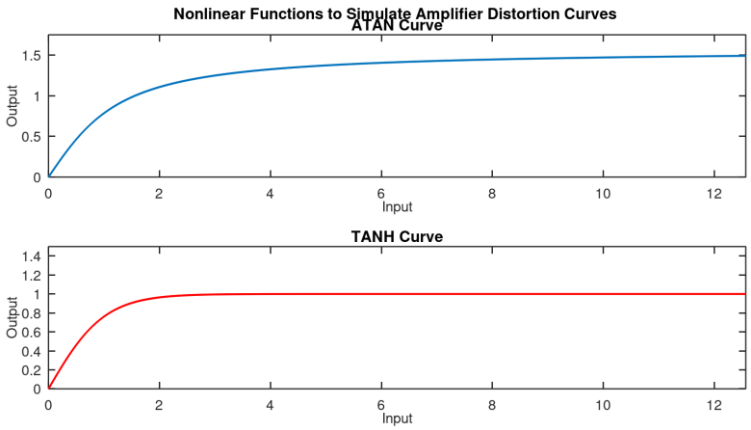


*Figure 6 ATAN and TANH Curves Used in the Distortion LUT*

The distortion in Dr. dB is accomplished in two stages. The first stage applies asymmetrical distortion by processing the positive half of the input through the hyperbolic tangent lookup table, while the negative half of the signal is processed by the arctangent lookup table. The resulting gain curve is shown in Figure 7.

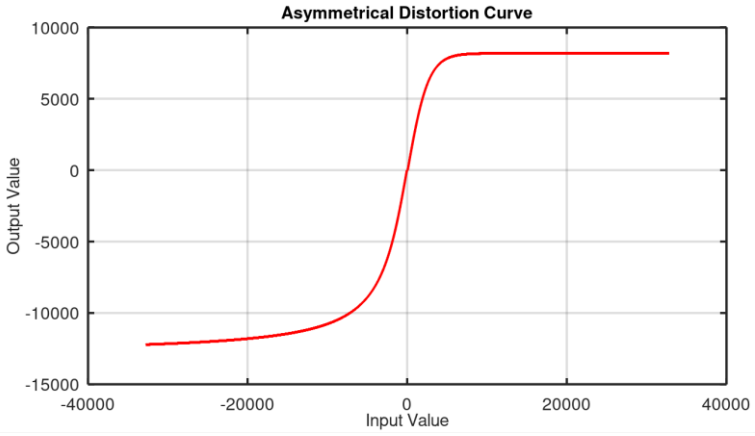


*Figure 7 Asymmetrical Distortion Curve*

An example of a sinusoid being processed in this method is shown in Figure 8. Clearly, the resulting waveform has different amounts of harmonic content added to its positive and negative portions. Further analysis might involve performing an FFT on a sample waveform to dissect example how much of which harmonics are injected into the signal through this processing.

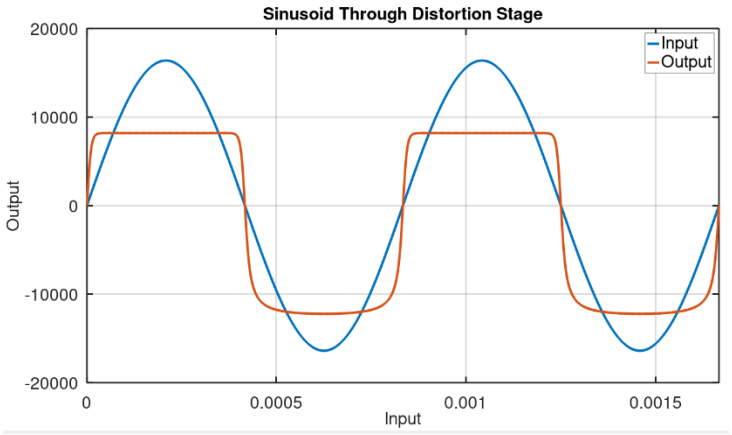


*Figure 8 Example of the Asymmetrical Distortion Algorithm Applied to a Sine Wave*

4. To increase the amount of distortion, a 'preamp gain' value is applied to the input signal, which is user selectable between unity and 10. For input samples greater than 32767 after the gain is applied, the index is automatically set to the max value of 1023 for maximum compression/distortion.
5. Following the asymmetrical clipping stage, the signal is put through a symmetrical clipping stage, which applies a harder distortion (again, this description is more qualitative than quantitative). An example of the output of Figure 8 being fed through this next distortion stage is shown in Figure 9.

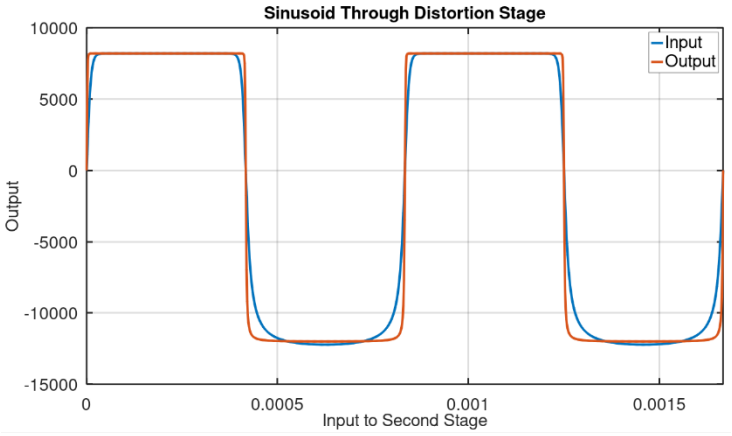


*Figure 9 Output of Distortion Stage 1 Processed by Distortion Stage 2*

6. To illustrate how all of this is accomplished in software, Figure 10 shows the function distModule1, which applies the asymmetrical distortion algorithm.

```
Int16 distModule1(int* inSig){
    Int16 outSig;
    // determine sign of input
    sigSign = (*inSig < 0) ? -1 : 1;

    // if input > 32767, set indx to 1023 (max)
    // else divide input by 32, giving max value of 1023 when gain = 1
    distLUTindx = (*inSig >= MAXINPUT) ? (DISTLUTSIZE - 1) : abs(*inSig) >> INDXSHIFT;

    // apply asymmetrical clipping
        // use tanh for neg
        // use atan for positive
    outSig = (sigSign) ? (Int16)(sigSign*(sigLevel2*TANH[distLUTindx])):
                         (Int16)(sigSign*(sigLevel2*ATAN[distLUTindx]));

    return outSig;
```

*Figure 10 Asymmetrical Distortion Algorithm (distortionModule.c)*

The 'sigLevel2' variable that is multiplied by the lookup table's output is set to 6200, yielding a maximum magnitude of 9300 on the output. This value could certainly use more optimization but was chosen after some experimentation to leave sufficient headroom for additional processing on the magnitude of the output. As it was, when the signal is fully distorted and all effects are in play, the output of the ISR was near the maximum.

7. The following figures show oscilloscope measurements of the output signal of Dr. dB with varying levels of distortion applied to an input sinusoid. Figure 11 shows the output with the distortion stage bypassed, so the signal is only being run through a set of IIR filters. There is some attenuation but otherwise the signal appears only to have been shifted somewhat in phase.

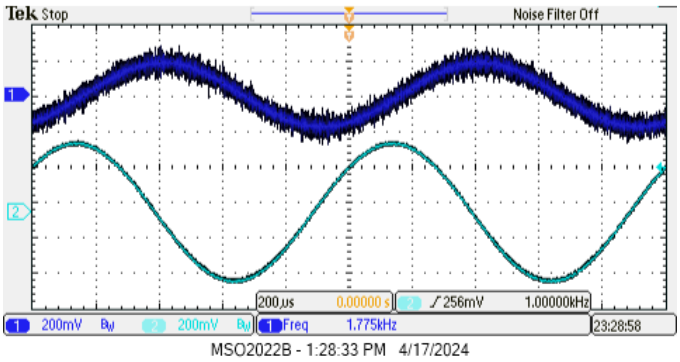


*Figure 11 Measured Output (CH1) vs. Input (CH2) When Distortion Bypassed*

Figure 12 shows the signal with the minimum distortion applied (pre-amp gain set to 1), which resembles the theoretical waveform shown in Figure 8.

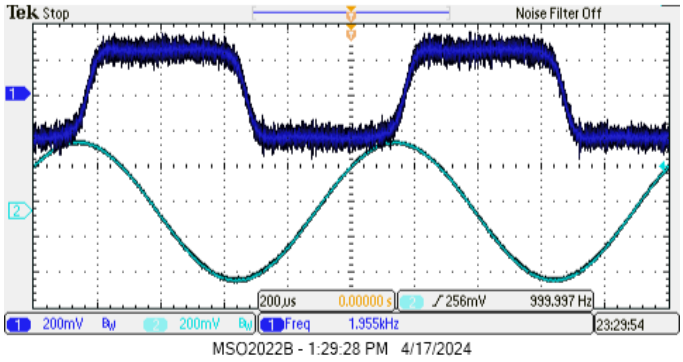


*Figure 12 Measured Output (CH1) vs. Input (CH2) When Distortion at Minimum Setting*

Figure 13 shows the signal with maximum distortion applied (pre-amp gain set to 10). This signal is heavily altered, with a curious jagged-ness in the negative portions of the signal. This may be due to some overflow occurring, as most of the signal variables used in the code were of type Int16. This is another area in which the code could be optimized further.

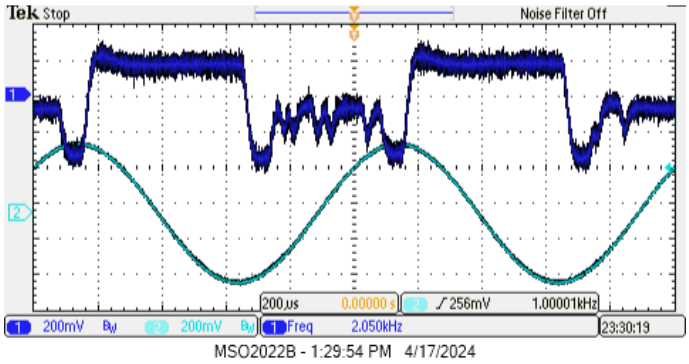


*Figure 13 Measured Output (CH1) vs. Input (CH2) When Distortion at Maximum Setting*

Audibly, however, the distortion generally sounded good and was not overly harsh. Lower levels of distortion had a similar 'overdrive' sound to circuits like the aforementioned Boss SD-1 pedal.

## Part 5: Tonestack

1. Initially, the plan was to utilize a set of filters based upon an actual amplifier's EQ circuitry, as found in David Te-Mao Yeh's analysis of the Fender Blackface amplifier tonestack ([2], p. 24).

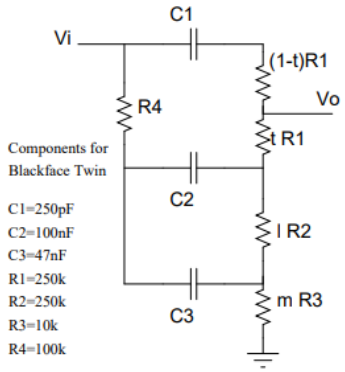


*Figure 14 Fender Tonestack [2]*

David's analysis shows this filter to have the following continuous-time transfer function:

$$H(s) = \frac{b_1 s + b_2 s^2 + b_3 s^3}{a_0 + a_1 s + a_2 s^2 + a_3 s^3}$$

1.1. The coefficients of this filter are parameterized such that they change when the bass, mid, or treble potentiometers are adjusted. It was attempted to implement this filter digitally by performing Bilinear Transformations and generating a set of lookup tables for different potentiometer settings, but the resulting filter taps proved to be unstable. This likely could have been solved by breaking the filter down into second order sections, but due to time constraints an alternative set of filters was pursued.

2. Pivoting to a new approach, it was decided to implement the tonestack section of Dr. dB by creating three IIR Butterworth bandpass filters. To select the passbands, further research was performed. As a starting point, the frequencies in the audible spectrum can be loosely categorized into the following [5]:
    2.1.1. Low frequencies: 20 Hz – 250Hz
    2.1.2. Mid frequencies: 250 Hz – 4kHz
    2.1.3. Treble: 4kHz - 20kHz
3. As in the input filtering stage, the three bandpass filters were implemented by cascading a low and high pass IIR Butterworth filter together. Each individual filter was designed as a second order filter using the 'butter' command in Octave, which allows a digital filter to be directly designed by giving it $\theta$ values ranging from 0 to 1. Figure 15 is given as an example.

```
% generate 2nd order
fs = 96000;
N = 1;
% digital filter ->fs/2
WL = 75/(fs/2);
WH = 150/(fs/2);

[bzH, azH] = butter(N, WL, "high");
```

*Figure 15 Filter Design Script*

The passbands chosen for the filters were informed by the above list and are described below. The passbands were chosen with some separation between them to allow for dips in the overall response and to give a generally flat response when all gain knobs are set to unity.

3.1. Low: passband from 75Hz – 150 Hz. The guitar's lowest fundamental frequency is around 80Hz, so this passband was chosen to control this region of the spectrum.
3.2. Mids: passband from 350Hz – 1kHz. The 1kHz region is one that is often altered by the tone knob of guitar pedals.
3.3. Treble: passband from 2kHz – 10kHz. This is clearly the widest passband and contains some 'upper mid' frequencies as well as high frequencies.

4. Figure 16 shows the overall response of the three filters summed together with three combinations of gain applied.
4.1. The first shows the response when the 'bass' knob is at its maximum attenuation while the 'mid' and 'treb' knobs are at unity gain.
4.2. The next is the response when the 'mid' knob is at maximum attenuation while the 'bass' and 'treb' knobs are at unity gain.
4.3. The third is the response when the 'treb' knob is at maximum attenuation while the 'bass' and 'mid' knobs are at unity gain.

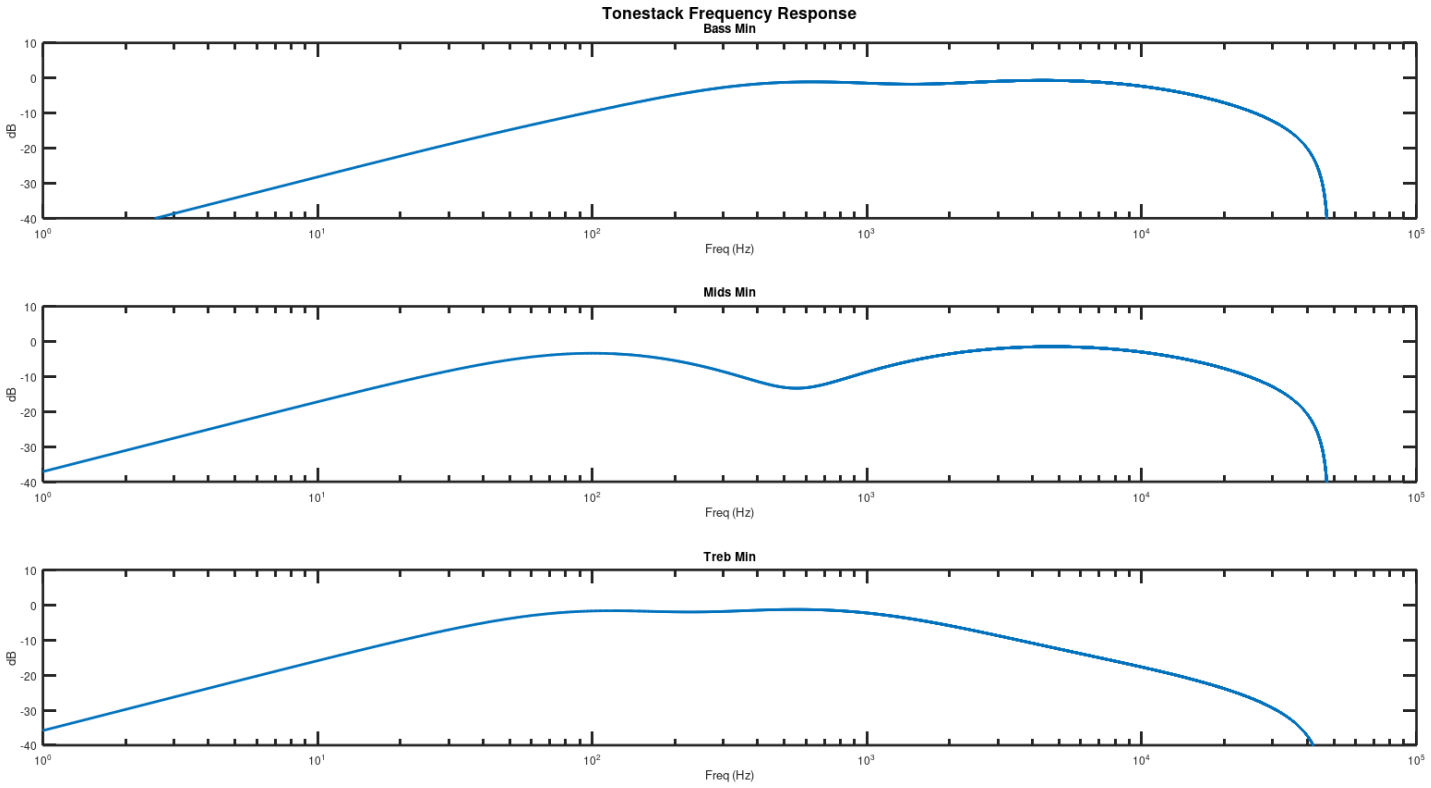


*Figure 16 Overall Tonestack Response*

5. The resulting tonestack allowed a great deal of control over the sound of the guitar. Accentuating the 'mids', for example, while attenuating the bass and treble somewhat, gave a sound more reminiscent of British amplifiers, which tend to be mid-heavy.

## Part 6: Modulation Effects

1. Two common effects, often implemented in guitar pedals, are reverb and phaser. A relatively simple version of each was implemented in Dr. dB to give some more options for the sounds that could be created.

## Part 6.a: Reverb

1. There are many, many ways to create a reverb effect, but the initial idea for the reverb algorithm used in Dr. dB was actually found in a homework problem. This problem showed reverb being implemented as a simple addition of a delayed sample to a current sample. The problem did state that the implementation as such was not necessarily suitable due to a lack of constant gain over all frequencies. As an experiment, however, this basic idea was implemented in Octave, and it yielded a pleasing delay/reverb effect, so it was implemented on the Zoom board.
2. Since the code operates with a 96kHz sample rate, a fairly large buffer is required to yield any significant amount of time-domain delay. To facilitate the delay, a circular buffer of 16384 samples was created and allocated to memory using the #pragma command. Every 6$^{th}$ sample is loaded into this buffer in the ISR, yielding a total available delay of

$$\frac{16384\ delay\ samples}{\frac{96000}{6}(\frac{samples}{\text{sec}})} = 1.024\ seconds$$

To attempt to add some more realism to the delay, the samples loaded into the circular buffer were the output samples of the 'mid' range bandpass filter, which were attenuated by half before being loaded into the buffer. The reasoning here is that some frequency content is absorbed when reflected off different types of surfaces, so limiting

the frequency band of the delayed signals to the midrange seemed appropriate and would avoid any additional harsh high frequencies being added back into the output signal.

3. Figure 17 shows an example of the reverb algorithm developed being applied to a sinusoidal input signal, where the output is clearly shifted and has a greater amplitude than the input due to the addition of the delayed samples.

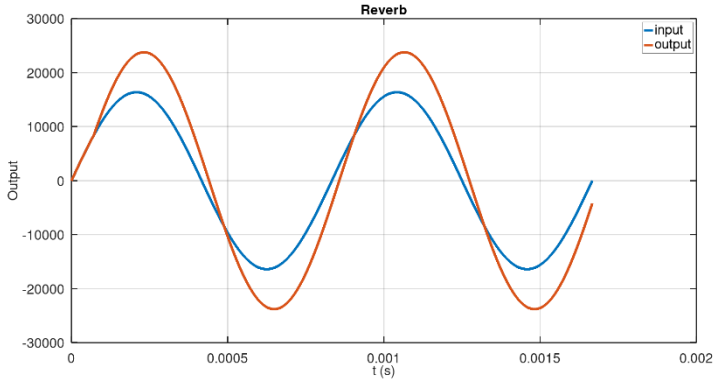


*Figure 17 Example of the Reverb Algorithm Applied to a Sine Wave*

4. In practice, the length of the delay is user-selectable via a knob on the GUI. Fourteen levels of delay are available in multiples of 1/16. Figure 18 shows how the sample is read from the buffer, where *cbuf.head* is the index containing the newest sample, *VBUFLEN* is 16384, *verbLength* is an array of 1/16 multiples of 16384, and *verbIndex* is the integer selected by the user to control the reverb length. Essentially, the sample read from the buffer is at the 'tail' index, but the number of samples between the head and the tail varies depending on the user's choice of reverb length.

```
// if verb indx = 0, disable reverb on output
    // else read samples from the buffer
        // sample read is dictated by the user-selected verbIndx -> see verbBuf.c file
sampleOut = (verbIndx == 0) ? 0 : (cbuf.buf[(cbuf.head + 1 + (VBUFLEN - verbLength[verbIndx])) % VBUFLEN]);
```

*Figure 18 Reverb Implementation (verbBuf.c)*

## Part 6.b: Phaser

1. The idea for how to implement the phaser effect was found in a research paper published as part of the 19th International Conference on Digital Audio Effects, in which a phaser effect pedal was measured and analyzed. One plot in particular illustrated the idea of using a low frequency oscillator for the phaser effect, as shown in Figure 19.

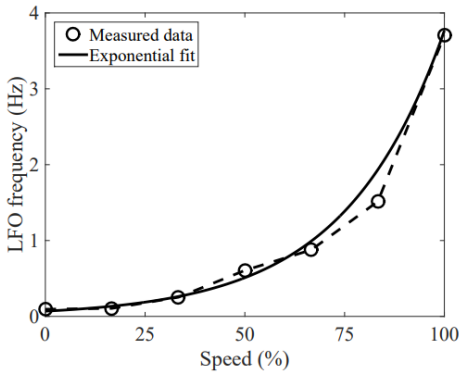


*Figure 19 LFO Plot [1]*

2. It should be noted that the phaser effect should also incorporate some sort of all-pass filtering; however, due to time constraints this additional filter was not developed for the final implementation of this project.
3. In practice, an 'enable' signal and a knob/slider varying from 5-10 (integers) were developed to allow the user to enable the phaser effect and vary the speed of the low frequency oscillator.
4. The low frequency oscillator was generated using the lookup table method, with a base frequency of 5Hz and a maximum frequency of 10Hz. To save processing time, the LFO samples were only generated when the reverb sample was being loaded into the circular buffer. The phaser is only usable when the reverb is enabled. The phaser is implemented as a mixer, where the LFO sample is multiplied by the sample that is loaded into the circular buffer.
5. Figure 20 shows an example of the phaser algorithm thus described being applied to a simple sinusoidal input. In practice it yields some very interesting effects.

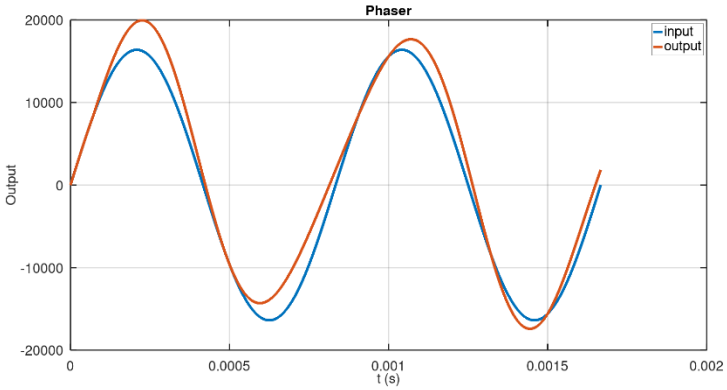


*Figure 20 Example of the Phaser Algorithm Applied to a Sine Wave*

## Part 7: Output

1. To complete the signal chain, a simple user-selectable gain is applied to the output signal, which is a sum of the tonestack's output signal, and the sample read from the reverb/phaser buffer. Volume knobs are often logarithmic, but to leave some noticeable volume at lower levels it was decided use quadratically determined volume levels, where the gain applied to the output is:

$$\left(\frac{x}{10}\right)^2, \qquad x = 0, 1, 2, \ldots .10$$

## Part 8: GUI

1. The GUI was designed to look a bit like a guitar amp, with the user interface broken into a few different groups. The top left three knobs of the GUI are grouped to control the tonestack, ordered from low frequencies to high. Below them is distortion and volume, with the distortion enable having a switch connected as well. The right-hand side contains the phaser and reverb, with a phaser disable option as well. The background is a simulated wood panel, imported as an image. The basic instructions in order to implement the GUI on the OMAP board came from Justin and Hector's video.

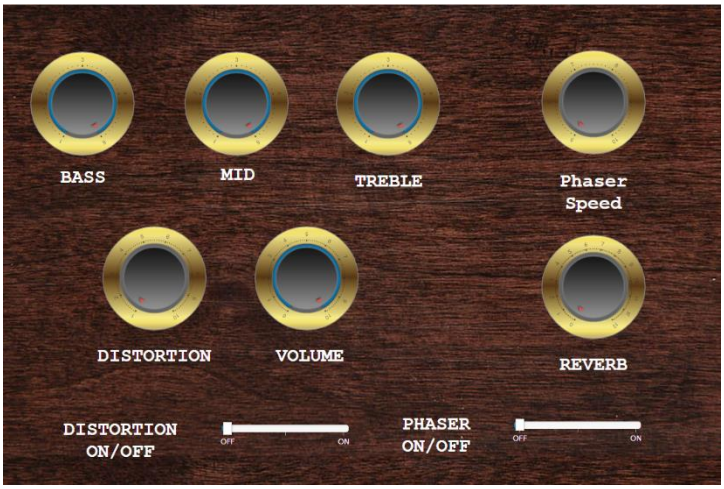


*Figure 21 Dr. dB User Interface*

## Conclusion

This was a very exciting project to undertake and has inspired interest in further pursuing development of audio processing software. The overall design of the project incorporated many aspects of DSP covered in this course and in Lab, including testing out designs in MATLAB/Octave, IIR filter design, FIR processing (the reverb), creating circular buffers, allocating large arrays to memory, and utilizing lookup tables for various tasks such as variable frequency sinusoid generation.

## References

[1]

R. Kiiski, F. Esqueda, and V. Välimäki, "TIME-VARIANT GRAY-BOX MODELING OF A PHASER PEDAL," in *Proceedings of the 19th International Conference on Digital Audio Effects*, Brno University of Technology: The Faculty of Electrical Engineering and Communication, Sep. 2016, pp. 31–39. Accessed: Apr. 19, 2024. [Online]. Available: https://dafx16.vutbr.cz/files/DAFx16_proceedings.pdf

[2]

D. Te-Mao Yeh, "DIGITAL IMPLEMENTATION OF MUSICAL DISTORTION CIRCUITS BY ANALYSIS AND SIMULATION A DISSERTATION SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND THE COMMITTEE ON GRADUATE STUDIES OF STANFORD UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY," 2009. Available: https://ccrma.stanford.edu/~dtyeh/papers/DavidYehThesissinglesided.pdf

[3]

"Multipoint Iterative Matching & Impedance Correction Technology (MIMIC™)," Fractal Audio Systems, Apr. 2013. Accessed: Apr. 19, 2024. [Online]. Available: https://www.fractalaudio.com/downloads/manuals/axe-fx-2/Fractal-Audio-Systems-MIMIC-(tm)-Technology.pdf

[4]

Perry, "Tube Amplifiers Explained, Part 10: Understanding Distortion," *Analog Ethos*, Apr. 05, 2020. https://www.analogethos.com/post/understanding-distortion (accessed Apr. 19, 2024).

[5]

“Electric guitar EQ guide,” *neuraldsp.com*. https://neuraldsp.com/articles/electric-guitar-eq-guide

[6]

R. Robinette, “Tube Guitar Amp Overdrive”, https://robrobinette.com/Tube_Guitar_Amp_Overdrive.htm#:~:text=Tube%20amplifiers’%20transfer%20curves%20are, harmonic%20and%20intermodulation%20distortion%20too. (accessed Apr. 20, 2024).

# Code

## CCS Files Run on the Zoom Kit

### inputBPF.c

```
/*
 * inputbpf.c
 *
 *  Created on: Apr 12, 2024
 *      Author: dstnm
 */

#include <inputBPF.h>

/* 2ND ORDER CASCADED BUTTERWORTH FILTERS
 */

float bpfb[4] = {0.996738, -0.996738, 0.211325, 0.211325}, bpfa[4] = {1.000000, -0.993476, 1.000000, -
0.577350};
Int16 bpfInPrev[3] = {0}, bpfOutPrev[3] = {0}, bpfOutPrev2[3] = {0};

Int16 inputBPF(Int16 *in){

    bpfInPrev[0] = *in;
    // apply HPF
     bpfOutPrev[0] = round(bpfInPrev[0]*bpfb[0] + bpfInPrev[1]*bpfb[1]
                   - (bpfOutPrev[1]*bpfa[1]));


     // apply LPF
     bpfOutPrev2[0] = round(bpfOutPrev[0]*bpfb[2] + bpfOutPrev[1]*bpfb[3]
                    - (bpfOutPrev2[1]*bpfa[3]));

     // shift inputs/outputs
    // bpfInPrev[2] = bpfInPrev[1];
     bpfInPrev[1] = bpfInPrev[0];

    // bpfOutPrev[2] = bpfOutPrev[1];
     bpfOutPrev[1] = bpfOutPrev[0];

    // bpfOutPrev2[2] = bpfOutPrev2[1];
     bpfOutPrev2[1] = bpfOutPrev2[0];

    return bpfOutPrev2[0];
}
```

### distortionModule.c

```
/*
 * distortionModule.c
 *
 *  Created on: Apr 10, 2024
 *      Author: dstnm
 */

#include "distortionModule.h"
#include "distortionLUT.h"

volatile int preampGain = 1;    // scale factor for the input signal
int distLUTindx = 0,            // index in the lookup table
        sigLevel = 8192,       // scale factor for the output
        sigLevel2 = 6200,
        sigSign = 1;            // sign of the current signal

Int16 distLow = 0 , distMid = 0, distTreb = 0;

////////////////////////////////////////////////////////////////
Int16 distModule1(int* inSig){
```

```
    Int16 outSig;
    // determine sign of input
    sigSign = (*inSig < 0) ? -1 : 1;

    // if input > 32767, set indx to 1023 (max)
    // else divide input by 32, giving max value of 1023 when gain = 1
    distLUTindx = (*inSig >= MAXINPUT) ? (DISTLUTSIZE - 1) : abs(*inSig) >> INDXSHIFT;

    // apply asymmetrical clipping
        // use tanh for neg
        // use atan for positive
    outSig = (sigSign) ? (Int16)(sigSign*(sigLevel2*TANH[distLUTindx])):
                         (Int16)(sigSign*(sigLevel2*ATAN[distLUTindx]));

    return outSig;

}

////////////////////////////////////////////////////////////
Int16 distModule2(Int16* inSig){
    Int16 outSig;

    // determine sign of input
    //sigSign = (*inSig < 0) ? -1 : 1;     // uncomment if distModule1 is not run before calling distModule2

    // if input > 32767, set indx to 1023 (max)
    // else divide input by 32, giving max value of 1023 when gain = 1
    distLUTindx = (*inSig >= MAXINPUT) ? (DISTLUTSIZE - 1) : abs(*inSig) >> INDXSHIFT;

    // apply symmetrical clipping
        // use atan
    outSig = (sigSign) ? (Int16)(sigSign*sigLevel*ATAN[distLUTindx]) :
                         (Int16)(sigSign*sigLevel*ATAN[distLUTindx]);

    return outSig;

}
```

toneStack.c

```
/*
 * toneStack.c
 *
 *  Created on: Apr 3, 2024
 *      Author: dstnm 'n' lcsass
 */

#include "toneStack.h"


#define EQMAX 10
#define EQLEN 5

volatile int  eqB = 5, bPrev = 5, eqM = 5, mPrev = 5, eqT = 5, tPrev = 5;

// LUT of gain levels applied to filter outputs
float gainLUT[5] = {0.04, 0.16, 0.36, .64, 1};

// gain levels for filter outputs (unity is max)
float bassGain = 1, midGain = 1, trebGain = 1;

///// TAPS
// 75 - 150
float blowEQ[EQLEN] = {0.997552, -0.997552, 0.004885, 0.004885}, alowEQ[EQLEN] = {1.000000, -0.995103,
1.000000, -0.990230};

// 350 - 1000
float bmidEQ[EQLEN] = {0.988675, -0.988675, 0.031699, 0.031699}, amidEQ[EQLEN] = {1.000000, -0.977351,
1.000000, -0.936602};

// 2000 - 10000
```

```
float bhiEQ[EQLEN] = {0.938488, -0.938488, 0.253427, 0.253427}, ahiEQ[EQLEN] = {1.000000, -0.876976, 1.000000,
-0.493145};

// filter buffers
Int16 toneStackInBuf[3] = {0},
      lowBuf[3] = {0}, lowOut[3] = {0}, lowOut2[3] = {0},
      midBuf[3] = {0}, midOut[3] = {0}, midOut2[3] = {0},
      hiBuf[3]  = {0}, hiOut[3]  = {0}, hiOut2[3]  = {0};

Int16 maxInput = 0;

Int16 lowFilt(Int16* inSig){
    lowBuf[0] = *inSig;

    // apply HPF
    lowOut[0] = round(toneStackInBuf[0]*blowEQ[0] + toneStackInBuf[1]*blowEQ[1]
                - (lowOut[1]*alowEQ[1]));


    // apply LPF
    lowOut2[0] = round(lowOut[0]*blowEQ[2] + lowOut[1]*blowEQ[3]
                 - (lowOut2[1]*alowEQ[3]));

    // shift inputs/outputs

    lowOut[1] = lowOut[0];

    lowOut2[1] = lowOut2[0];


       return lowOut2[0];

}

Int16 midFilt(Int16* inSig){
    midBuf[0] = *inSig;

    // apply HPF
    midOut[0] = round(toneStackInBuf[0]*bmidEQ[0] + toneStackInBuf[1]*bmidEQ[1]
                 - (midOut[1]*amidEQ[1]));


    // apply LPF
    midOut2[0] = round(midOut[0]*bmidEQ[2] + midOut[1]*bmidEQ[3]
                 - (midOut2[1]*amidEQ[3]));

    // shift inputs/outputs

    midOut[1] = midOut[0];

    midOut2[1] = midOut2[0];

       return midOut2[0];

}


Int16 highFilt(Int16* inSig){
    hiBuf[0] = *inSig;

    // apply HPF
    hiOut[0] = round(toneStackInBuf[0]*bhiEQ[0] + toneStackInBuf[1]*bhiEQ[1]
                - (hiOut[1]*ahiEQ[1]));


    // apply LPF
    hiOut2[0] = round(hiOut[0]*bhiEQ[2] + hiOut[1]*bhiEQ[3]
                 - (hiOut2[1]*ahiEQ[3]));

    // shift inputs/outputs
```

```
    hiOut[1] = hiOut[0];

    hiOut2[1] = hiOut2[0];

    return hiOut2[0];

}

// update the EQ gain settings
void setEQ(){
    //float tmpA, tmpB;
    bassGain = gainLUT[eqB - 1];
    midGain  = gainLUT[eqM - 1];
    trebGain = gainLUT[eqT - 1];

}
```

verbBuf.h

```
/*
 * cbuf.h
 *
 *  Created on: Feb 14, 2024
 *      Author: dstnm
 */
#include "DSP_Config.h"

#ifndef VERBBUF_H_
#define VERBBUF_H_

#define VBUFLEN      16384
#define VBUF15_16    (VBUFLEN * 15)/16
#define VBUF7_8      (VBUFLEN * 7) /8
#define VBUF13_16    (VBUFLEN * 13)/16
#define VBUF6_8      (VBUFLEN * 3)/4
#define VBUF11_16    (VBUFLEN * 11)/16
#define VBUF5_8      (VBUFLEN * 5)/8
#define VBUF9_16     (VBUFLEN * 9)/16
#define VBUF4_8      VBUFLEN / 2
#define VBUF7_16     (VBUFLEN * 7)/16
#define VBUF3_8      (VBUFLEN * 3)/8
#define VBUF5_16     (VBUFLEN * 5)/16
#define VBUF2_8      (VBUFLEN)/4
#define VBUF1_16     (VBUFLEN)/16


#define SAMPLEOFFSET 6

#define BUFLENARRAY 16

extern volatile int verbIndx;

typedef struct{
    Int16   buf[VBUFLEN];
    uint16_t head;
    uint16_t tail;
} cbuffer;

extern int verbLength[BUFLENARRAY];

extern int samplerCnt;
extern Int16 sampleOut;

// function
int loadBuf(Int16 *data);

#endif /* VERBBUF_H_ */
```

verbBuf.c

```
/*
 * cbuf.c
```

```
 *
 *  Created on: Feb 14, 2024
 *      Author: dstnm
 */

#include "verbBuf.h"

#pragma DATA_SECTION (cbuf, "CE0");

cbuffer cbuf = {{0}, (VBUFLEN-1), 0};

volatile int verbIndx = 0;
int samplerCnt = 0;
Int16 sampleOut = 0;

int verbLength[BUFLENARRAY] = {0, VBUF1_16, VBUF2_8, VBUF5_16, VBUF3_8, VBUF7_16, VBUF4_8, VBUF9_16, VBUF5_8,
VBUF11_16, VBUF6_8, VBUF13_16, VBUF7_8, VBUF15_16, VBUFLEN};


/*  loadBuf(float *data)
 *      Load data into circular buffer.
 *      Increment head and tail, overwriting
 *      oldest element when buffer is full.
 */
int loadBuf(Int16 *data){
    // load data element
    cbuf.buf[cbuf.head] = *data;
    // increment head for next load
    cbuf.head = (cbuf.head + 1) % VBUFLEN;

    cbuf.tail =  (cbuf.head + 1) % VBUFLEN;

    return 1;
}
```

Main.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

//////////////////////////////////////////////////////////////////////////
// Filename: main.c
//
// Synopsis: Main program file for demonstration code
//
//////////////////////////////////////////////////////////////////////////

#include "DSP_Config.h"
#include "toneStack.h"

int main()
{

    // initialize DSP board
    DSP_Init();

    // call StartUp for application specific code
    // defined in each application directory
    StartUp();

    // main stalls here, interrupts drive operation
    while(1) {
        // update EQ settings if user changes a knob in the GUI
      if((eqB != bPrev) || (eqM != mPrev) || (eqT != tPrev)){
        bPrev = eqB;
        mPrev = eqM;
        tPrev = eqT;
        setEQ();
      }

    }
```

```
}
```

ISRs.c

```
// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

/////////////////////////////////////////////////////////////////////////
// Filename: ISRs.c
//
// Synopsis: Interrupt service routine for codec data transmit/receive
//
// Original ISRs file is a simple pass-through routine. This was modified
// to include all stages necessary to implement a basic digitial guitar amplifier
//
// input filter -> distortion -> 3-band EQ -> modulation effects -> volume attenuation -> output
/////////////////////////////////////////////////////////////////////////
#include <stdbool.h>
#include "inputBPF.h"
#include "DSP_Config.h"
#include "distortionModule.h"
#include "toneStack.h"
#include "verbBuf.h"
#include "sinLUT5Hz.h"

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word.  The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.

#define LEFT  0
#define RIGHT 1

volatile union {
    Uint32 UINT;
    Int16 Channel[2];
} CodecDataIn, CodecDataOut;


// volatile variables for user control
volatile int bypassDist = 0,    // allows user to bypass the distortion stage
             ampVolume  = 10;   // 11 selectable volume levels, 10 is max
volatile int phaseFreq = 5,     // default phaser freq = 5Hz
             phaseEn   = 0;     // enable/disable the phaser effect


/* add any global variables here */
Int16 input;
// ratio of user selected frequency to phaser default frequency
float phaseRatio = 0.0;
// scalars applied to output signal to manipulate the volume level
float volLevels[11] = {0.0, 0.01, 0.04, 0.09, 0.16, 0.25, 0.36, 0.49, 0.64, 0.81, 1.0};
// tracker variable
int maxOut = 0;
// circular buffer for delay/reverb
extern cbuffer cbuf;

interrupt void Codec_ISR()
/////////////////////////////////////////////////////////////////////////
// Purpose:   Codec interface interrupt service routine
//
// Input:     None
//
// Returns:   Nothing
//
// Calls:     CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes:     None
/////////////////////////////////////////////////////////////////////////
{
    /* add any local variables here */
```

```
    float xRight;
    Int16 distOut, tonestackOut, verbOut, verbIn;
    int bpfOut, distIn;
    static Int16 output = 0;


    if(CheckForOverrun())                   // overrun error occurred (i.e. halted DSP)
        return;                             // so serial port is reset to recover

    CodecDataIn.UINT = ReadCodecData();     // get input data samples

    /* add your code starting here */

    // this example simply copies sample data from in to out
    input  = CodecDataIn.Channel[ LEFT];
    xRight = CodecDataIn.Channel[ RIGHT];

    ////////////
    // debug- track magnitude of input
    //maxInput = (abs(input) > maxInput) ? abs(input) : maxInput;
    ////////////

    ////////////////////////////////////////
    ///////    INPUT FILTERING    /////////
    ////////////////////////////////////////
    // bandlimit the input btwn 100Hz & 8kHz
    // and attenuate signal for additional headroom

    bpfOut = ((inputBPF(&input)) >> 1);

    ////////////////////////////////////////
    ///////    DISTORTION STAGE   /////////
    ////////////////////////////////////////

    // apply preamp gain
    distIn = bpfOut * preampGain;

    // run asymmetrical distortion - more even harmonics
    distOut = (distModule1(&distIn));
    // run symmetrical distortion - more odd harmonics, additional clipping
    distOut = distModule2(&distOut);

    ////////////////////////////////////////
    ///////     EQUALIZER STAGE   /////////
    ////////////////////////////////////////
    // allow user to bypass distortion and feed original input signal into tonestack
    toneStackInBuf[0] = (bypassDist) ? distOut :  (Int16)bpfOut;

    // PERFORM LOW, MID, AND HIGH FREQ FILTERING
    distLow  = lowFilt(&toneStackInBuf[0]);
    distMid  = midFilt(&toneStackInBuf[0]);
    distTreb = highFilt(&toneStackInBuf[0]);
    // shift buffer
    toneStackInBuf[1] = toneStackInBuf[0];

    // apply user selected gain on the bands
    tonestackOut = (Int16)(bassGain * distLow) +
                    (Int16)(midGain * distMid) +
                   (Int16)(trebGain * distTreb);

    ////////////////////////////////////////
    /////     MODULATION STAGE      ///////
    ////////////////////////////////////////

    // MODULATION INCLUDES 'REVERB' WITH OPTIONAL PHASER EFFECT

    // sinusoid generator with LUT for phaser LFO
    static float phIndx = 0;
    phaseRatio = (float)phaseFreq / f0;

    // gather every x-th sample
```

```
    if(samplerCnt == 0){
        // calculate the LFO LUT index
        phIndx += phaseRatio;
        if(phIndx > (PHSIZE-1)){
            phIndx = phIndx - (PHSIZE);
        }
        // use 'dampened' signal -> the output of mid freq filter as input to delay buffer
            // apply phaser if enabled
            verbIn = (phaseEn) ?  (Int16)((distMid >> 1)*sLUT[(int)phIndx]) :
                                          distMid >> 1;
        // load the signal into the circular buffer
        loadBuf(&verbIn);
    }

    // increment the sampler counter
    samplerCnt = (samplerCnt + 1) % SAMPLEOFFSET;

    // if verb indx = 0, disable reverb on output
        // else read samples from the buffer
            // sample read is dictated by the user-selected verbIndx -> see verbBuf.c file
    sampleOut = (verbIndx == 0) ? 0 : (cbuf.buf[(cbuf.head + 1 + (VBUFLEN - verbLength[verbIndx])) %
VBUFLEN]);


    ////////////////////////////////////
    ////////     OUTPUT STAGE     / ///////
    ////////////////////////////////////

    // add the delayed sample to the current output signal
    verbOut = (Int16)(tonestackOut + sampleOut);

    ///////////
    // debug variable to monitor the magnitude of the output signal:
    //maxOut = (abs(verbOut) > maxOut) ? abs(verbOut) : maxOut;
    ///////////

    // apply user-selected volume attenuation
    output = (Int16)(volLevels[ampVolume] * verbOut);

    // route signal to output channels
    CodecDataOut.Channel[ LEFT] = output;
    CodecDataOut.Channel[RIGHT] = output;

    /* end your code here */

    WriteCodecData(CodecDataOut.UINT);      // send output data to  port
}
```

EQknobs.gel

GEL sliders can be used to play with the amplifier if GUI is not loaded

```
menuitem "EQ"

slider bassKnob(1, 5, 1, 5, bk)
{
    eqB = bk;
}

slider midKnob(1, 5 ,1, 5, mk)
{
    eqM = mk;
}

slider trebKnob(1, 5 ,1, 5, tk)
{
    eqT = tk;
}
```

```
slider enable(0, 1 ,1, 1, e)
{
    bypassDist = e;
}

slider gain(1, 10 ,1, 1, g)
{
    preampGain = g;
}

slider verb(0, 14 ,1, 1, v)
{
    verbIndx = v;
}

slider vol(0, 10 ,1, 1, vol)
{
    ampVolume = vol;
}

slider phaseEn(0, 1 ,1, 1, pe)
{
    phaseEn = pe;
}

slider phaseFreq(5, 10 ,1, 1, pf)
{
    phaseFreq = pf;
}
```

---

## Scripts used for LUT generation, filter generation, and plots for the report

Note: These scripts were written and run in Octave software, which may require slight modifications to function properly in MATLAB due to syntax differences

### bandpass.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Authors: Dustin
% Course: EGR 423
% Instructor: Prof. Bruce Dunne
% Date: Apr. 2024
% File written for and run in Octave software
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
pkg load signal;  % required to run filter cmds in Octave
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% INPUT FILTER

% generate 2nd order filters
fs = 96000;
N = 1;
% digital filter ->fs/2
WL = 100/(fs/2);
WH = 8000/(fs/2);

% octave allows a digital filter to be generated directly by
% selecting frequencies < 1
[bzH, azH] = butter(N, WL, "high");

figure(1)
s1 = subplot(3, 1, 1);
wz = 1: 1: fs/2;
HzH2 = freqz(bzH, azH, wz, fs);
semilogx(wz, 20*log10(abs(HzH2)), "linewidth",2);
set(gca, "linewidth", 2, "fontsize", 32);
set(s1, 'title', 'High Pass',"fontsize", 12);
ylabel('dB',  "fontsize", 16);
xlabel('Frequency (Hz)',  "fontsize", 16);
```

```
ylim([-40 10]);

s2 = subplot(3, 1, 2);
[bzL, azL] = butter(N, WH);
HzL2 = freqz(bzL, azL, wz, fs);
semilogx(wz, 20*log10(abs(HzL2)), "linewidth",2);
set(gca, "linewidth", 2, "fontsize", 32);
set(s2, 'title', 'Low Pass',"fontsize", 12);
ylabel('dB',  "fontsize", 16);
xlabel('Frequency (Hz)',  "fontsize", 16);


s3 = subplot(3, 1, 3);
Ht2 = HzL2.*HzH2;
semilogx(wz, 20*log10(abs(Ht2)), "linewidth",2);
set(gca, "linewidth", 2, "fontsize", 32);
set(s3, 'title', 'Cascaded Response',"fontsize", 12);
ylabel('dB',  "fontsize", 16);
xlabel('Frequency (Hz)',  "fontsize", 16);
ylim([-40 10]);

if(1)
fileID = fopen('bandpass.txt','w');

fprintf(fileID,'b\n');
fprintf(fileID,'%f, ', bzH(1,:));
fprintf(fileID,'%f, ', bzL(1,:));

fprintf(fileID,'\na\n');
fprintf(fileID,'%f, ', azH(1,:));
fprintf(fileID,'%f, ', azL(1,:));
fprintf(fileID,'\n\n');

fprintf(fileID,'\n\n');
fclose(fileID);
endif
```

Lowhi_cascade.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Authors: Dustin
% Course: EGR 423
% Instructor: Prof. Bruce Dunne
% Date: Apr. 2024
% File written for and run in Octave software
%
% This file is to generate plots & tap values for three sets of
% cascaded low-and-high pass filters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
pkg load signal;

%%%% NEW TONESTACK :(
%%%%%%%% FILT 1 for low freq band

% generate 2nd order
fs = 96000;
N = 1;
% digital filter ->fs/2
WL = 75/(fs/2);
WH = 150/(fs/2);

[bzH, azH] = butter(N, WL, "high");

figure(1)
wz = 1: 1: fs/2;
HzH1 = freqz(bzH, azH, wz, fs);
semilogx(wz, 20*log10(abs(HzH1)));
hold on;
ylim([-40 10]);
```

```
[bzL, azL] = butter(N, WH);
HzL1 = freqz(bzL, azL, wz, fs);
semilogx(wz, 20*log10(abs(HzL1)));
hold off;

figure(2)
Ht1 = HzL1.*HzH1;
semilogx(wz, 20*log10(abs(Ht1)));
ylim([-40 10]);

if(1)
fileID = fopen('lowfreqCascade.txt','w');

fprintf(fileID,'b\n');
fprintf(fileID,'%f, ', bzH(1,:));
fprintf(fileID,'%f, ', bzL(1,:));

fprintf(fileID,'\na\n');
fprintf(fileID,'%f, ', azH(1,:));
fprintf(fileID,'%f, ', azL(1,:));
fprintf(fileID,'\n\n');

fprintf(fileID,'\n\n');
fclose(fileID);
endif


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% FILT 2 for mid freq band

% generate 2nd order
fs = 96000;
N = 1;
% digital filter ->fs/2
WL = 350/(fs/2);
WH = 1000/(fs/2);

[bzH, azH] = butter(N, WL, "high");


figure(3)
wz = 1: 1: fs/2;
HzH2 = freqz(bzH, azH, wz, fs);
semilogx(wz, 20*log10(abs(HzH2)));
hold on;
ylim([-40 10]);


[bzL, azL] = butter(N, WH);
HzL2 = freqz(bzL, azL, wz, fs);
semilogx(wz, 20*log10(abs(HzL2)));
hold off;

figure(4)
Ht2 = HzL2.*HzH2;
semilogx(wz, 20*log10(abs(Ht2)));
ylim([-40 10]);

if(1)
fileID = fopen('MIDfreqCascade.txt','w');

fprintf(fileID,'b\n');
fprintf(fileID,'%f, ', bzH(1,:));
fprintf(fileID,'%f, ', bzL(1,:));

fprintf(fileID,'\na\n');
fprintf(fileID,'%f, ', azH(1,:));
fprintf(fileID,'%f, ', azL(1,:));
fprintf(fileID,'\n\n');

fprintf(fileID,'\n\n');
```

```
fclose(fileID);
endif

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% FILT 3 for hi freq band

% generate 2nd order
fs = 96000;
N = 1;
% digital filter ->fs/2
WL = 2000/(fs/2);
WH = 10000/(fs/2);

[bzH, azH] = butter(N, WL, "high");


figure(5)
wz = 1: 1: fs/2;
HzH3 = freqz(bzH, azH, wz, fs);
semilogx(wz, 20*log10(abs(HzH3)));
hold on;
ylim([-40 10]);


[bzL, azL] = butter(N, WH);
HzL3 = freqz(bzL, azL, wz, fs);
semilogx(wz, 20*log10(abs(HzL3)));
hold off;

figure(6)
Ht3 = HzL3.*HzH3;
semilogx(wz, 20*log10(abs(Ht3)));
ylim([-40 10]);

if(1)
fileID = fopen('TREBfreqCascade.txt','w');

fprintf(fileID,'b\n');
fprintf(fileID,'%f, ', bzH(1,:));
fprintf(fileID,'%f, ', bzL(1,:));

fprintf(fileID,'\na\n');
fprintf(fileID,'%f, ', azH(1,:));
fprintf(fileID,'%f, ', azL(1,:));
fprintf(fileID,'\n\n');

fprintf(fileID,'\n\n');
fclose(fileID);
endif


figure(7)
hold on;
HtT = .04*Ht1 + 1*Ht2 + Ht3;
s1 = subplot(3, 1, 1);

semilogx(wz, 20*log10(abs(HtT)), "linewidth", 2);
ylim([-40 10]);
set(s1, 'title', 'Bass Min',  "fontsize", 16);
ylabel('dB');
xlabel('Freq (Hz)');
set(gca, "linewidth", 2, "fontsize", 12);

HtT = 1*Ht1 + 0.04*Ht2 + Ht3;
s2 = subplot(3, 1, 2);

semilogx(wz, 20*log10(abs(HtT)), "linewidth", 2);
ylim([-40 10]);
set(s2, 'title', 'Mids Min',  "fontsize", 16);
ylabel('dB');
xlabel('Freq (Hz)');
```

```
set(gca, "linewidth", 2, "fontsize", 12);


HtT = 1*Ht1 + Ht2 + 0.04*Ht3;
s3 = subplot(3, 1, 3);

semilogx(wz, 20*log10(abs(HtT)), "linewidth", 2);
ylim([-40 10]);
set(s3, 'title', 'Treb Min',  "fontsize", 16 );
ylabel('dB');
xlabel('Freq (Hz)');
set(gca, "linewidth", 2, "fontsize", 12);

hold off;
S = axes( 'visible', 'off', 'title', 'Tonestack Frequency Response', "linewidth", 2,  "fontsize", 16);
```

LUTdistortion.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Authors: Dustin
% Course: EGR 423
% Instructor: Prof. Bruce Dunne
% Date: Apr. 2024
% File written for and run in Octave software
%
% This file has two purposes. The first is to validate that our distortion
% algorithm will in fact scale the input values seen on the zoom board, with its
% range of roughly +/-32767, as desired using the lookup table method.
%
% The second section of this file reads in a wav file recording and processes it
% in the same manner before renormalizing the levels back to that of the input for
% playback.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x = linspace(0, 4*pi, 1024);

% luts are the positive half of trig functions
atanLut = atan(x);
tanhLut = tanh(x);

% generate input sequence at max range of zoom board
inSeq = linspace(-32768, 32768, (32768*2)+1);
len = length(inSeq);

outSeq1 = zeros(1, len);

% test the lut - use the input divided by 32 (for a max of 1024) to
% select the index of the lut. Scale this value by some desired max amplitude
for i = 1 : len
  if(inSeq(i) < 0)
    s = -1;
  else
    s = 1;
  endif
  if((inSeq(i) == 0) || (floor(abs(inSeq(i))/32) == 0))
    outSeq(i) = 0;
  else
    outSeq1(i) = s*10000*atanLut(floor(abs(inSeq(i))/32)); %(inSeq(i)/12)
  endif

endfor

figure(1)
plot(inSeq, outSeq1);
grid on;
title('outSeq1');


%%%%%%%%%%%%%%%%%%%%%%%%
% use LUT to test distorting the guitar signal
```

```
fs = 44100;
in = audioread('guitar.wav');
% convert to mono
in = in(:, 1);

% play the unprocessed signal
sound(in, fs);

% normalize input to 32768
scale = 32768;
inn = scale * (in / max(abs(in)));

len = length(in);
out = zeros( 1, len);

% apply pre-amp gain - this increases distortion
gain = 10;
inn = gain * inn;

for i = 1 : len

  lutindx = floor(abs(inn(i))/32);
  if( lutindx > 1024 )
    lutindx = 1024;
  endif

  if(inn(i) < 0)
    s = -1;
  else
    s = 1;
  endif

  if((inn(i) == 0) || (lutindx == 0))
    out(i) = 0;
  elseif(inn(i) < 0)
    out(i) = s*10000*tanhLut(lutindx); %(inSeq(i)/12)
  else
    out(i) = s*10000*atanLut(lutindx);
  endif
endfor

% renormalize to input level
out = max(in) * (out/max(out));

% show the input vs output signal
figure(2)
subplot(2,1,1)
plot(in);
subplot(2,1,2)
plot(out);

% play the processed signal
sound(out, fs);
```

distortionEffects.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Authors: Dustin
% Course: EGR 423
% Instructor: Prof. Bruce Dunne
% Date: Apr. 2024
% File written for and run in Octave software
%
% This file is to generate plots showing our waveshaping distortion method against
% a simple sinusoidal input
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x = linspace(0, 4*pi, 1024);

% luts are the positive half of trig functions
```

```
atanLut = atan(x);
tanhLut = tanh(x);

% generate input sequence at max range of zoom board
f = 1.2e3;
t = linspace(0, 2/f, 1024);
input = 16384*sin(2*pi*f*t);
out = zeros(1, 1024);
out2 = zeros(1, 1024);

% test the lut - use the input divided by 32 (for a max of 1024) to
% select the index of the lut. Scale this value by some desired max amplitude
for i = 1 : 1024
  if(input(i) < 0)
    s = -1;
  else
    s = 1;
  endif
  if(input(i) == 0 )
    out(i) = 0;
  elseif(input(i) < 0 )
    out(i) = s*8192*atanLut(abs(floor(input(i) / 16)));
  else
    out(i) = s*8192*tanhLut(abs(floor(input(i) / 16)));
  endif

endfor

figure(1)

plot(t, input, "linewidth",3);
hold on;
plot(t, out, "linewidth",3);
grid on;
hold off;
set(gca, "linewidth", 2, "fontsize", 12);
title('Sinusoid Through Distortion Stage',"fontsize", 16);
ylabel('Output',  "fontsize", 16);
xlabel('Input',  "fontsize", 16);
xlim([0 2/f]);
set(gca, "linewidth", 2, "fontsize", 12);

figure(2)
for i = 1 : 1024
  if(out(i) < 0)
    s = -1;
  else
    s = 1;
  endif
  if(out(i) == 0 )
    out2(i) = 0;
  elseif(out(i) < 0 )
    out2(i) = s*8192*atanLut(abs(floor(out(i) / 16)));
  else
    out2(i) = s*8192*tanhLut(abs(floor(out(i) / 16)));
  endif

endfor

plot(t, out, "linewidth",3);
hold on;
plot(t, out2, "linewidth",3);
grid on;
hold off;
set(gca, "linewidth", 2, "fontsize", 12);
title('Sinusoid Through Distortion Stage',"fontsize", 16);
ylabel('Output',  "fontsize", 16);
xlabel('Input to Second Stage',  "fontsize", 16);
xlim([0 2/f]);
set(gca, "linewidth", 2, "fontsize", 12);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%
% use LUT to distort the guitar signal

in = audioread('guitar.wav');
% convert to mono
in = in(:, 1);

% normalize input to 32768
scale = 32768;
inn = scale * (in / max(abs(in)));

len = length(in);
out = zeros( 1, len);

% apply pre-amp gain - this increases distortion
gain = 10;
inn = gain * inn;
if(0)
for i = 1 : len

  lutindx = floor(abs(inn(i))/32);
  if( lutindx > 1024 )
    lutindx = 1024;
  endif

  if(inn(i) < 0)
    s = -1;
  else
    s = 1;
  endif

  if((inn(i) == 0) || (lutindx == 0))
    out(i) = 0;
  elseif(inn(i) < 0)
    out(i) = s*10000*tanhLut(lutindx); %(inSeq(i)/12)
  else
    out(i) = s*10000*atanLut(lutindx);
  endif
endfor

% renormalize to input level
out = max(in) * (out/max(out));


figure(3)
subplot(2,1,1)
plot(in);
subplot(2,1,2)
plot(out);

endif
%fs = 44100;
%sound(out, fs);
```

distLUTgenerator.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Authors: Dustin
% Course: EGR 423
% Instructor: Prof. Bruce Dunne
% Date: Apr. 2024
% File written for and run in Octave software
%
% This file is to generate a header file with atan and tanh value lookup tables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% generate 1024 datapoints
x = linspace(0, 4*pi, 1024);

% LUTS are the positive half of trig functions
```

```
atanLut = atan(x);
tanhLut = tanh(x);

figure(1)
hold on;

s1 = subplot(2, 1, 1);

plot(x, atanLut, "linewidth", 2);
set(s1, 'title', 'ATAN Curve',"fontsize", 12);
set(gca, "linewidth", 2, "fontsize", 12);
ylabel('Output',  "fontsize", 16);
xlabel('Input',  "fontsize", 16);

ylim([0 1.75]);
xlim([0 4*pi]);
set(gca, "linewidth", 2, "fontsize", 12);

s2 = subplot(2, 1, 2);

plot(x, tanhLut, "linewidth", 2, 'r');
set(s2, 'title', 'TANH Curve', "fontsize", 12);
ylabel('Output',  "fontsize", 16);
xlabel('Input',  "fontsize", 16);
ylim([0 1.5]);
xlim([0 4*pi]);
set(gca, "linewidth", 2, "fontsize", 12);
S = axes( 'visible', 'off', 'title', 'Nonlinear Functions to Simulate Amplifier Distortion Curves ',
"linewidth", 2,  "fontsize", 16);


figure(2)
xn = linspace(-4*pi, 0, 512);
dn = atan(xn);
xp = linspace(4*pi/512, 2*pi, 512);
dp = tanh(xp);
dist = ([dn(1:512) dp(1:512)]);
xg = linspace(-4*pi, 4*pi, 1024);

plot(xg, dist, "linewidth", 6, 'r');
grid on;
set(gca, "linewidth", 2, "fontsize", 16);
title('Asymmetrical Distortion Curve', "fontsize", 16);
ylabel('Output',  "fontsize", 16);
xlabel('Input',  "fontsize", 16);

figure(3)

xg = linspace(-4*pi, 4*pi, 1024);
dist2 = atan(xg);

plot(xg, dist2, "linewidth", 6, 'r');
grid on;
set(gca, "linewidth", 2, "fontsize", 16);
title('Symmetrical Distortion Curve', "fontsize", 16);
ylabel('Output',  "fontsize", 16);
xlabel('Input',  "fontsize", 16);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GENERATE THE LUT FILE
% ---------------------
LUTSIZE = 1024;

generate = 0;
if(generate)

filename = "distortionLUT";

j = fopen([filename '.h'],'w');
  fprintf(j, '/* %-35s */\n\n', [filename '.h']);
  fprintf(j, '/* Distortion LUTS=s\n');
  fprintf(j, ' *------------------------\n\n');
```

```
  fprintf(j, ' * LUTs are 1024 datapoints of atan(x) and tanh(x)\n');
  fprintf(j, ' */ \n\n');


  fprintf(j, '#ifndef DISTORTIONLUT_H_\n');
  fprintf(j, '#define DISTORTIONLUT_H_\n\n');
  fprintf(j, '#include "distortionModule.h"\n\n');

  fprintf(j, '#define DISTLUTSIZE 1024\n\n');

  % use PRAGMA to store the LUT values

  % ATAN LUT
  fprintf(j, '#pragma DATA_SECTION (ATAN, "CE0");\n\n');
  fprintf(j, 'float ATAN[DISTLUTSIZE] = {\n');

  for i = 1 : 1024
    fprintf(j, '\t\t%g,\n', atanLut(i));
  endfor

  fprintf(j, ' };\n\n');   % no comma

    % TANH LUT
  fprintf(j, '#pragma DATA_SECTION (TANH, "CE0");\n\n');
  fprintf(j, 'float TANH[DISTLUTSIZE] = {\n');

  for i = 1 : 1024
    fprintf(j, '\t\t%.5f,\n', tanhLut(i));
  endfor

  fprintf(j, ' };\n\n');   % no comma


  % EOF
  fprintf(j, '#endif /* DISTORTIONLUT_H_ */\n');

  fclose(j);

endif
```

reverbTest.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Authors: Dustin and Lucas
% Course: EGR 423
% Instructor: Prof. Bruce Dunne
% Date: Apr. 2024
% File written for and run in Octave software
%
% This file is used to demonstrate the validity of our 'reverb' method
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear;
fs = 44100;
% import guitar clip
src = audioread('guitar.wav');
% convert to mono
src = transpose(src(:, 1));
smpls = length(src);

% play unprocessed signal
sound(src, fs);

%
out = src;

% create circular buffer for delayed signal
% length of the reverb can be changed with vLen
vLen = 128;
cntmax = floor(44.1*vLen); cntmin = 1;
```

```
head = cntmax;
tail = cntmin;
buf = zeros(1,cntmax);

for i = 1:smpls
  buf(head) = out(i);

  out(i) = out(i) + .75*buf(tail);
  if(tail == cntmax)
    tail = cntmin;
  else
    tail++;
  endif
  if(head == cntmax)
    head = cntmin;
  else
    head++;
  endif

endfor

% renormalize
out= out/max(abs(out));

sound(out, fs);
```

reverbPlots.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Authors: Dustin and Lucas
% Course: EGR 423
% Instructor: Prof. Bruce Dunne
% Date: Apr. 2024
% File written for and run in Octave software
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fs = 44100;
% import guitar clip
%src = audioread('guitar.wav');
%src = transpose(src(:, 1));
f = 1.2e3;
t = linspace(0, 2/f, 32767);
src = 16384*sin(2*pi*f*t);

phase = sin(2*pi*1000*t);
smpls = length(src);

out = src;%zeros(1, smpls);


cntmax = floor(44.1*32); cntmin = 1;
head = cntmax;
tail = cntmin;
buf = zeros(1,cntmax);

for i = 1:smpls
  buf(head) = (out(i)/2);

  out(i) = out(i) + .5*buf(tail)*phase(i);
  if(tail == cntmax)
    tail = cntmin;
  else
    tail++;
  endif
  if(head == cntmax)
    head = cntmin;
  else
    head++;
  endif

endfor
```

```
figure(1)
plot(t, src, "linewidth",4);
hold on;
plot(t, out, "linewidth",4);
grid on;
hold off;
set(gca, "linewidth", 2, "fontsize", 32);
title('Phaser',"fontsize", 32);
ylabel('Output',  "fontsize", 32);
xlabel('t (s)',  "fontsize", 32);
set(gca, "linewidth", 2, "fontsize", 32);
legend('input', 'output', "fontsize", 32);

%out= out/max(abs(out));


%sound(out, fs);
```

phaserTest.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Authors: Dustin and Lucas
% Course: EGR 423
% Instructor: Prof. Bruce Dunne
% Date: Apr. 2024
% File written for and run in Octave software
%
% This file is used to demonstrate the validity of our 'phaser' effect
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear;
fs = 44100;
% import guitar clip
src = audioread('guitar.wav');
src = transpose(src(:, 1));
smpls = length(src);
% play unprocessed signal
sound(src, fs);

% change the speed of the phaser with PHASERFREQ
PHASERFREQ = 6;
t = linspace(0, smpls/fs, smpls);
% low frequency oscillation signal
wv = sin(2*pi*PHASERFREQ*t);

out = src;%zeros(1, smpls);

% create circular buffer for delayed signal
cntmax = floor(44.1*125); cntmin = 1;
head = cntmax;
tail = cntmin;
buf = zeros(1,cntmax);

for i = 1:smpls
  buf(head) = out(i)* wv(i);

  out(i) = out(i) + .5*buf(tail);
  if(tail == cntmax)
    tail = cntmin;
  else
    tail++;
  endif
  if(head == cntmax)
    head = cntmin;
  else
    head++;
  endif

endfor
```

```
out= out/max(abs(out));

% play processed signal
sound(out, fs);
```