Documentation of Changes to MATLAB Organization and Functionality:

Dynamic Audio Compression

June/August 2019, E. Bailey Galacci

Abstract

The purpose of this documentation is to describe the changes to the MATLAB codes made by University of Texas - Dallas on the Dynamic Audio Compression to realize real-time low latency analysis on a Field Programmable Gate Array (FPGA). Ideally, any person reading this documentation will be able to make the same changes and have identical code to the version used to generate an HDL compatible Simulink model.

Included before describing the details of the changes are graphs showing the differences between the two models, and a section discussing their potential effects.

The changes are documented in two sections: Organization and Functionality. Organizational changes are purely for clearer readability, and ideally do not affect the output in any way. Functional changes directly affect the output as described in two subsections:

- Initialization Changes
- Real-Time Analysis (RTA) Changes and Refactors

Most initialization changes were to simplify the model as well as fully define the map for inputs and outputs of dynamic compression. Many other changes were necessary to allow for real-time analysis (RTA), mainly involving full removal or fundamental redefinitions of variables based on knowledge of the full incoming signal. This was necessary because RTA systems cannot be aware of future inputs, and must instead be refactored to function only on memory and current inputs. This subsection will also identify the sources used to make these changes and refer to models describing the desired outputs.

Table of Contents

Quick Model Breakdown

Step 1: Applying a Prescription FIR

Before any compression is done, the model applies a prescription for the user to the input signal with a Finite Impulse Response (FIR) filter. The prescription is defined similarly to a hearing test: the gain needed for a given frequency is based on the user's ability to hear sounds at that frequency. Shown below in Figure 1 is an example audiogram.

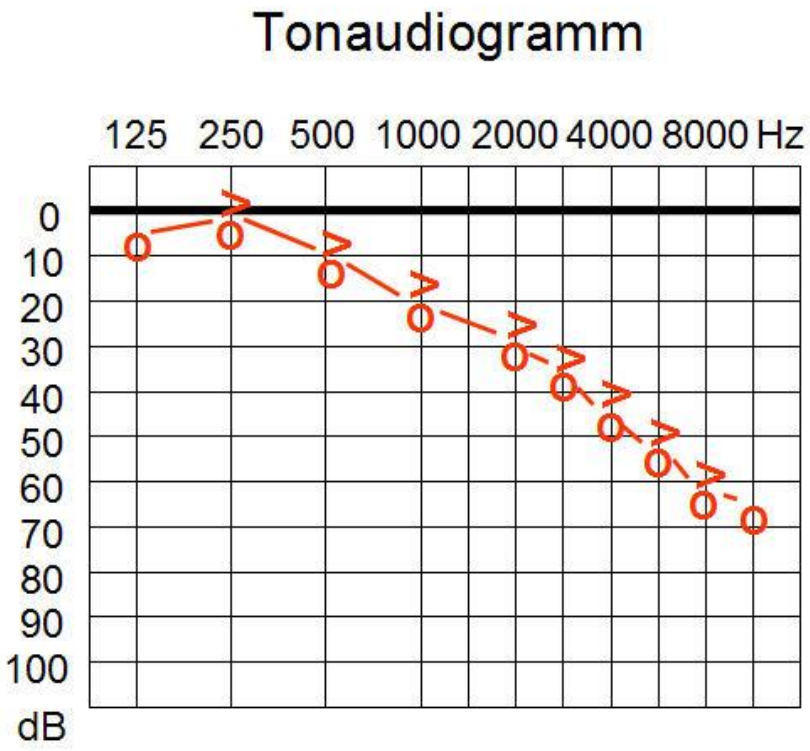

Figure 1 – Example Audiogram

Tonaudiogramm, Welleschik, Feb 2007

The marks identify the user's minimum audible volume heard at a given set of frequencies. This map is then translated into a prescription FIR filter that is applied to the input signal. Circles or the color red typically represent a user's right ear, while X's or the color blue typically represent the left ear. The frequency response of a filter fitting this example audiogram is shown in Figure 2 below.

The blue line shows the desired frequency response, and green the achieved frequency response of the filter. This FIR is calculated using a few factors outside the audiogram including sampling rate (fs) and desired delay of the filter in milliseconds.

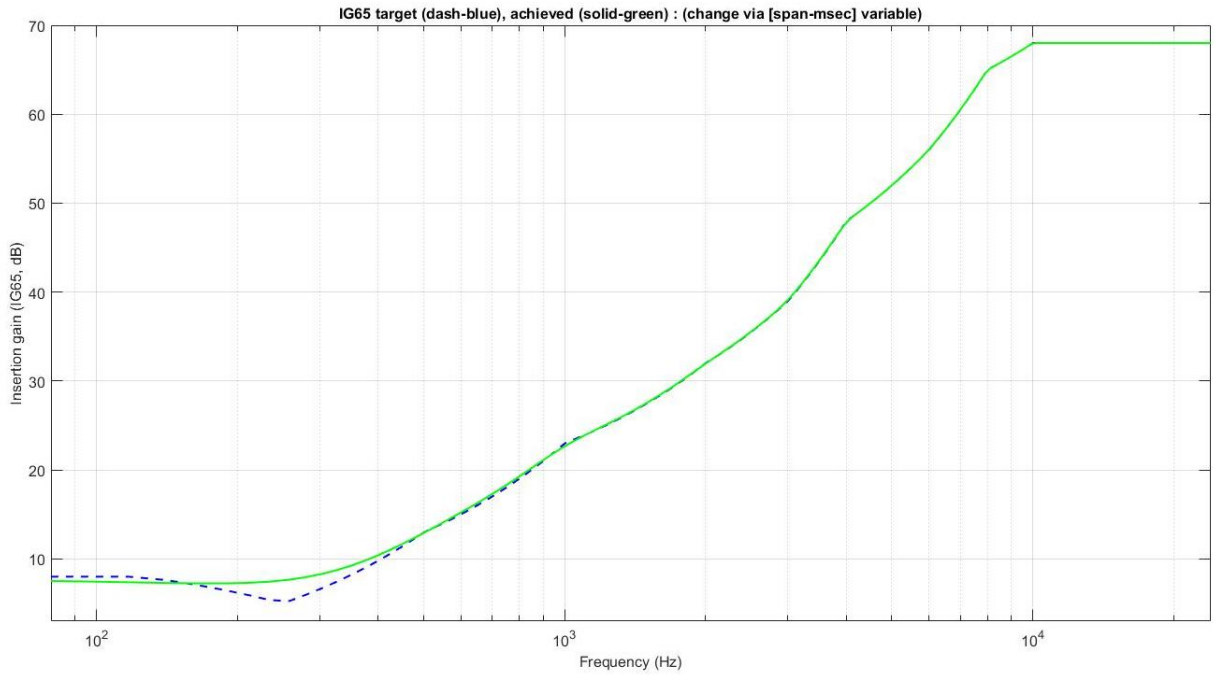

Figure 2 - The frequency response of the example audiogram

The Simulink model uses the same process to find a prescription FIR, with **no modifications**.

Step 2: Separate the Signal Into a Set of Frequency Bands

In order to improve performance of the compression to maximize a user's ability to hear another person's voice, as well as help separate different sounds. (citation needed)

The original codes do this by generating a series of bandpass filters equal to the number of channels desired by the user (NChans). Then, a set of filters is created to allow only some frequencies through, and ideally, the sum of the frequency responses of all filters will be unity gain. Nchan_FbankDesign and Nchan_FbankAGCAid are used to generate these filters using the center frequencies provided by AidSettingsXChans. Center frequencies are *generally* between 500 Hz and 8 kHz, geometrically centered at 2 kHz. In order to follow these guidelines for any number of channels, I wrote an algorithm to generalize center frequencies.

```
r = nthroot(100, NChans+5);
a = 100*(r^3);
chan_cfs = a*r.^(1:NChans);
```

Code Snippet 1: Algorithm to Define Center Frequencies of Bandpass Filters

One of the problems with this algorithm is that it tends to hug lower frequencies more than the predefined settings. Original 5 channel settings had center frequencies [500 1000 2000 4000 8000], this algorithm produces a set [562, 1000, 1778, 3162, 5623], as it is loosely centered at 1500 Hz. This is because settings for 22 channels had center frequencies bound between 125 Hz and 9161 Hz, so in order to fit this model the "center" filter couldn't be set at 2000 Hz while keeping geometric spacing between center frequencies, as the upper ceiling of 9000 Hz would be hit before a filter would be centered below 300 Hz.

The frequency response of 5 filter bands (with generalized parameters) is shown in Figure 3, after applying correction from Nchan_FbandAGCAid. Notice the sum of the responses closely resemble 0dB. Also note the first filter is actually a lowpass filter, as opposed to a bandpass. This is to allow low frequency signals through the first filter so they aren't ignored by the system. High frequency signals can be ignored since the sample rate of the system limits the upper frequency range to the Nyquist rate. In the original configuration, a highpass is also applied to the input signal to ignore frequencies below hearing range.

A few other settings were generalized and are functionally identical to the predefined settings for 5 channels, and loosely match predefined settings for 22 channels. These include chan_crs, chan_thrs, and deltaFSdB. Shown below are the changes for clarity, although much of this code was cut when implementing the Simulink model.

The goal of multiple bands is to help separate compression gain applied to different sounds. By applying compression in a set of bands, it is also possible to more accurately map different frequencies to different volumes compared to single band compression.

Note that after the prescription is applied, there are some parts of the bandpass that reach above the desired 0dB relative value. This difference is saved in a variable *Calib_recomb_dB*, and applied after compression to help avoid clipping.

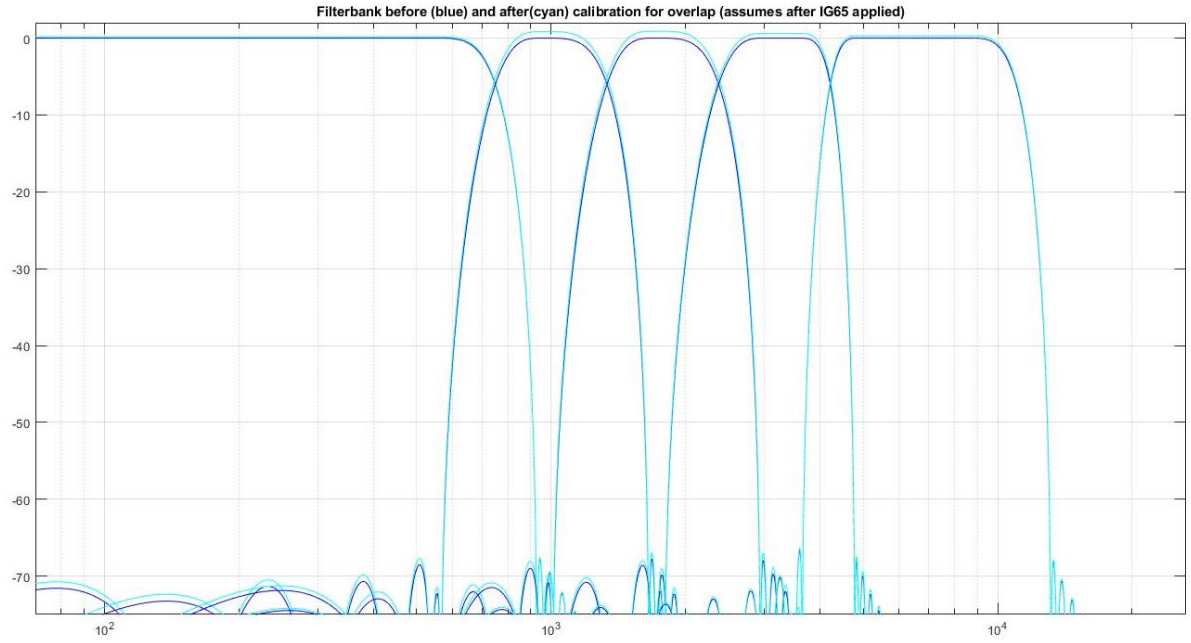

Figure 3 – Bandpass Frequency Responses

Full alterations for AidSettingsXChans compared to AidSettings5Chans:

X Chans, generalized

```
r = nthroot(100, NChans+3);
a = 100*(r^2);
chan_cfs = a*r.^(1:NChans);
```

5 Chans

```
chan_cfs = [500 1000 2000 4000 8000]
```

(Listed although functionally identical)

```
chan_crs(1:NChans) = 5;
```

```
chan_crs = [5 5 5 5 5];
```

```
chan_thrs(1:ceil(NChans/2)) = -15;
chan_thrs(floor(NChans/2)+1:…
   ceil(3*NChans/4)) = -10;
chan_thrs(ceil(3*NChans/4):NChans) = -5;
chan_thrs(NChans)=0;
```

```
chan_thrs = [-15 -15 -10 -5 0];
```

```
deltaFSdB(1:ceil(NChans/2)) = 13;
deltaFSdB(floor(NChans/2)+1:…
   ceil(3*NChans/4)) = 12;
deltaFSdB(ceil(3*NChans/4):NChans) = 11;
deltaFSdB(NChans)=10;
```

```
deltaFSdB = [13 13 12 11 10];
```

Step 3: Apply Compression to Each Band

Compression is a function designed to limit the effect of high volume sounds from overtaking the rest of the signal. Effectively, once a band reaches a certain threshold volume, the output is tapered, but not reduced below that threshold volume. Shown in Figure 4 is an example static characteristic.

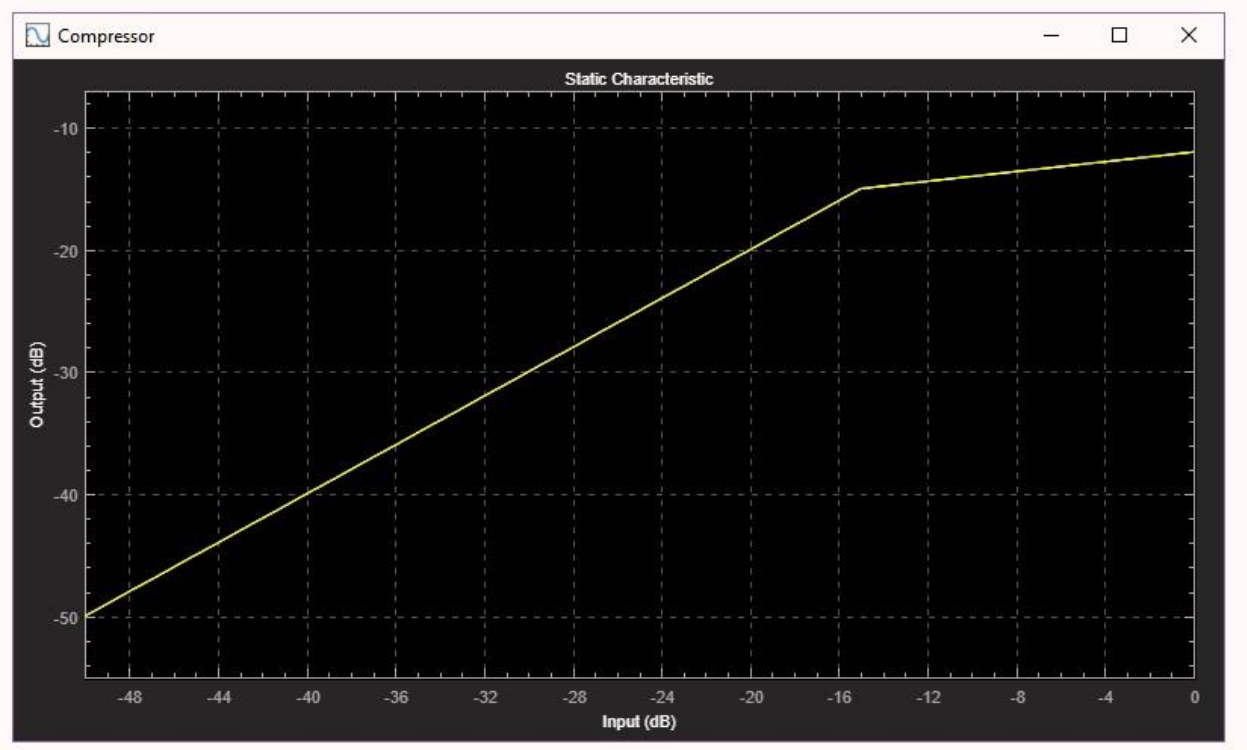

Figure 4 – Example Compression Static Characteristic, 0 dB set to maximum input value

In this example, the threshold volume is set at -15 dB, and the compression ratio is set at 5. This means volumes below -15 dB are unaffected, and anything above that volume is reduced by 5 times the dB difference to -15 dB (ex: -10 dB results in [-15 dB + (15-10)/5 = -14 dB]). This limits the loudest signal possible to play at -12 dB.

Step 4: Apply an Envelope Function to Compression

To improve comprehensibility in audio, compression must not be applied immediately to each sample, instead an envelope is applied to attenuate volume without significantly changing frequency content. Shown in figure 5 is an example of this concept applied.

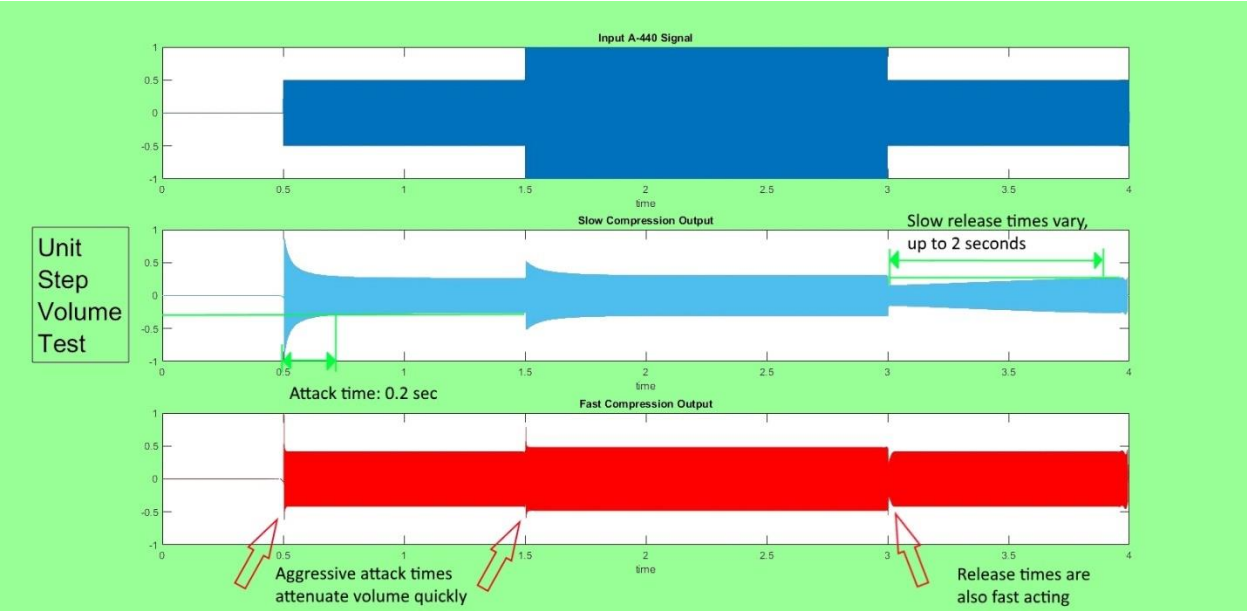

Figure 5 – Fast and Slow Compression Outputs

Input is a single 440Hz (A4) note, at stepping volumes throughout the signal. These values do not show typical results and are purely for demonstrating visually the effect of the envelope function (although the attack and release times are the same used in the updated model).

Attack time is defined as the amount of real time necessary for the envelope to change from 10% to 90% of the applied negative dB gain. Release time is defined as the amount of real time necessary for the envelope to change from 10% to 90% of positive dB gain.

When an envelope is applied, the signal attenuates much more slowly. With a sample rate of 48000 Hz, attack rates shown are on the order of 9600 and 2400 samples for slow and fast compression respectively. The purpose of these envelopes is to improve the clarity of speech while still maintaining volume control of the signal. From a paper on Signal to Noise Ratio Dynamic Compression in Hearing Aids[2], “Specifically, fast-acting compression was applied to speech-dominated T-F units where the SNR was high, while slow-acting compression was performed for noise-dominated T-F units with a low SNR.” These choices were made on subjective results, and for more details see the references section.

The updated model uses the same attack and release time settings as the original model, regardless of the number of channels. The generalized configuration is exactly consistent with all presets for these settings.

Organizational Changes

First, the MATLAB codes provided by Dallas[0] are written for an outdated version of MATLAB. These fixes are simple line by line replacements with newer commands that do the same thing.

Line 6 of *loadwavfile.m* contains the function call *wavread*. This has since been removed. For identical operation, this has been replaced with the following lines:

Code Snippet 2 – wavread replacement

```
6     %[sig, Fs, nbits] = wavread(strcat(newpath,newfile));
7     [sig, Fs] = audioread(strcat(newpath,newfile));
8     opplayer  = audioplayer(sig, Fs);
9     nbits = opplayer.bitspersample;
```

```
14    audiowrite(strcat(oppath,opfile), opwav, Fs, 'BitsPerSample',
nbits);
```

A similar issue occurs with *savewavfile.m*, where the function call *wavwrite* is used on line 13. This line can be replaced with the following functionally equivalent line:

Code Snippet 3 – wavwrite replacement

Simulink uses block models to easily show the functionality of any system. To accomplish this, any subfunction within a system must have fully defined variable inputs and outputs. Constant inputs that do not change during runtime can be used in the startup definitions of blocks, and these are applied during compile time.

In Matlab, scripts can be made that apply to the general workspace. These scripts do not need defined inputs and outputs, instead they can treat anything in the workspace as an input, and any variables saved during intermediate calculations will appear as outputs even if they are not specifically used outside that function. The codes from Dallas often used scripts instead of input and output defined functions. To better fit a block model for a transition to Simulink, I spent some time redefining *all* scripts other than the top level `MultiChanAidSim.m` to functions by finding all inputs needed and defining any variables used later in the program as outputs. The code generates the same output after applying these refactors.

All changes made to the Matlab codes will be shown. All function calls changed will be listed, as well as the new function headers.

Function Calls:

In order within MultiChanAidSim.m:

- `eval( sprintf('AidSettings%dChans', NChans) );` replaced with
  - `[deltaFSdB, chan_cfs, chan_crs, chan_thrs, t_atts, t_rels, ta_lim, tr_lim] ... %[aligndelaymsec]`
  - `= AidSettingsXChans(NChans);`
  - Also worth mentioning, this change allows the generalized configuration to be used instead of presets for 3, 5, or 22 channel predefined configurations. Also, the variable *aligndelaymsec* has been removed, as the realignment delay/advancement is calculated in the *update_channel_params* function
- `Loadwavfile` replaced with
  - `[VALID_WAV_FILE, sig, Fs, nbits, file_rms_dB, NStreams] ...`
  - `= loadwavfile(ipfiledigrms);`
- `update_channel_params` replaced with
  - `[ig_eq, calib_bpfs, dig_chan_lvl_0dBgain, dig_chan_dBthrs, Calib_recomb_dBpost] ...`
  - `= update_channel_params(Fs, NChans, chan_cfs, chan_thrs, insrt_frqs, insrt_gns, ...`
  - `ipfiledigrms, UPPER_FREQ_LIM, EQ_SPAN_MSEC, PRESCRIPT, DIAGNOSTIC, CALIB_DIAGNOSTIC) ;`
  - This function call is incredibly long (and the next…), however it could not be avoided. In the Simulink model, most of these inputs are constants defined in preprocessing, so they can be treated as constants after compile time. However, for Matlab to recognize them, all inputs, including constants, need to be defined.
- `recalculate` replaced with
  - `[proc_sig]...`
  - `= recalculate ...`
  - `(NChans, sig, Fs, re_level, calib_bpfs, t_atts, t_rels, ...`
  - `chan_crs, dig_chan_lvl_0dBgain, dig_chan_dBthrs,deltaFSdB, ... %[aligndelaymsec]`
  - `ta_lim, tr_lim, Calib_recomb_dBpost, ig_eq, DIAGNOSTIC);`
- `savewavfile` replaced with
  - `[opwav, stop_ok] = savewavfile(opfiledigrms, ipfiledigrms, proc_sig, Fs, nbits, NStreams);`

Function Headers:

Spacing in titles and ellipses are just for readability and consistency as some function headers are very large.
Note: all functions with new headers should include a line after all functionality:

```
end;
```

AidSettingsXChans.m :

```
function [deltaFSdB, chan_cfs, chan_crs, chan_thrs, t_atts, t_rels,
ta_lim, tr_lim] = ...
      AidSettingsXChans ...
            (NChans) %[aligndelaymsec]
```

loadwavfile.m :

```
function [VALID_WAV_FILE, sig, Fs, nbits, file_rms_dB, NStreams] = ...
      loadwavfile ...
            (ipfiledigrms)
```

update_channel_params.m :

```
function [ig_eq, calib_bpfs, dig_chan_lvl_0dBgain, dig_chan_dBthrs,
Calib_recomb_dBpost] = ...
      update_channel_params ...
            (Fs, NChans, chan_cfs, chan_thrs, insrt_frqs, insrt_gns,
            ipfiledigrms, UPPER_FREQ_LIM, EQ_SPAN_MSEC, PRESCRIPT,
            DIAGNOSTIC, CALIB_DIAGNOSTIC)
```

recalculate.m :

```
function [proc_sig] = ...
      recalculate ...
         (NChans, sig, Fs, re_level, calib_bpfs, t_atts, t_rels, ...
         chan_crs, dig_chan_lvl_0dBgain, dig_chan_dBthrs, deltaFSdB ...
         ta_lim, tr_lim, Calib_recomb_dBpost, ig_eq, DIAGNOSTIC)
```

savewavfile.m :

```
function [opwav, stop_ok] = ...
      savewavfile ...
            (opfiledigrms, ipfiledigrms, proc_sig, Fs, nbits, NStreams)
```

Functional Changes

Initialization

Configuration settings have been altered to apply to any number of bandpass channels.

Bandpass center frequencies are geometrically spaced, hard capped between 100 Hz and 10 kHz, using the following lines:

```
r = nthroot(100, NChans+3);
a = 100*(r^2);                          chan_cfs = [500 1000 2000 4000 8000]
chan_cfs = a*r.^(1:NChans);
```

Attack times are set by using slow compression, i.e. bounds below 0.2 seconds.
Release times are set decreasing from 2 seconds after the first band, down to 1000 by band 4, then setting all bands beyond the 4th to 1000. A switch is used to avoid multiple if/else statements.

```
t_atts(1) = 200; if NChans>1, t_atts(2:NChans)= 100; end
switch (NChans)
   case 1; t_rels = [2000];
   case 3; t_rels = [2000 1500 1200];
   otherwise; t_rels(1:3) = [2000 1500 1200]; t_rels(4:NChans) = 1000;
end
```

Compression threshold and ratio definitions are heavily redefined, removing any reference to knowledge of the full signal. A predefined 65 dB root mean squared (RMS) equivalent of the signal is replaced with a reference voltage and dBA value. Rather than defining a threshold and ratio relative to the 65dB RMS for each band, the floor volume is set to 0dBA (the least intense sound a typical human can reasonably hear), and the ratio is set using that threshold and the prescription's gain to set the highest possible output volume to 85 dBA (the least intense volume that can damage hearing). x_safe_dB is set to 85, x_low_dB is set to 0.

This maps typical hearing range to volumes a user can hear but are not dangerous, as a function of frequency and volume.

Code for applying these rules:

Max_Prescribed_Band_Gain defined as the maximum possible gain of a given bandpass and the prescription FIR. X_in_max defined as the maximum value for a point in the signal.

```
min_audible_thresh = x_ref * 10^(x_low_dB – ref_dB)/20
max_output = x_ref * 10^(x_safe_dB – ref_dB)/20
Comp_Threshold = min_audible_thresh * Max_Prescribed_Band_Gain
Comp_Ratio =
ln(X_in_max/min_audible_thresh) / ln(max_output/Comp_Threshold);
```

Remapping the compression thresholds fundamentally changes the philosophy behind the model.

The previous rules amplified all signals by the prescription gain, then attenuated only loud signals. This leads to disproportionate mapping of the signal in the hearing range of the user, and loads up the higher volume range more than the lower audible range because certain lower volumes are not compressed at all. With the new rules, rather than having bounds for compression thresholds and ratios based on **knowledge of the full signal RMS**, and with **no account for the prescription applied**, resulting in a signal that can **easily clip** bands with high prescription gains and requires full knowledge of the signal to process;

Instead, this design can create a signal that **form fits sounds coming in to match the needs of the user**, causing barely audible signals for the average person to play at barely audible ranges for their unique prescription, while still placing mid volume and high volume sounds into a range that is **never** dangerous to the wearer when the proper reference voltage and dBA value are used. This also skips the step of needing recombining gain to level power outputs of each band.

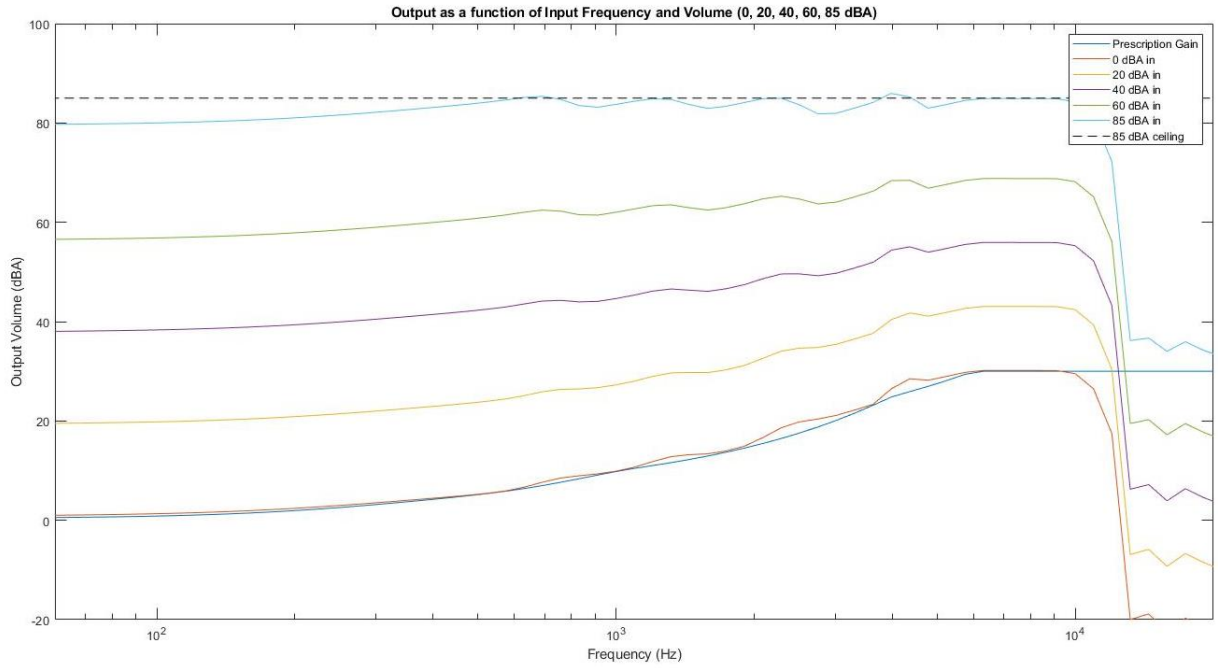

Figure 6 – Example Expected Updated Frequency/Volume Map

Notes on Real-Time Application in Simulink Model

One of the functions in the Matlab codes realigns the input signal after passing through bandpass filters. Since each filter has its own group delay (all linear), the signals need to be time-shifted to preserve the proper output. The Matlab codes time shift slower filters forward to line up together

A Simulink model however cannot use future inputs to determine an output. Instead, I set up a series of delay blocks by identifying which bandpass filters have the longest group delay (in sample time), then setting the delays of all blocks to the difference between that longest delay and that channel's own delay. This way, all channels effectively have the same group delay. Figure 7 shows the block model for this section. Also shown are the bandpass filters separating the input into 5 channels, the compression blocks, and recombining gain blocks to help compensate for non-unity gain when considering the prescription FIR.

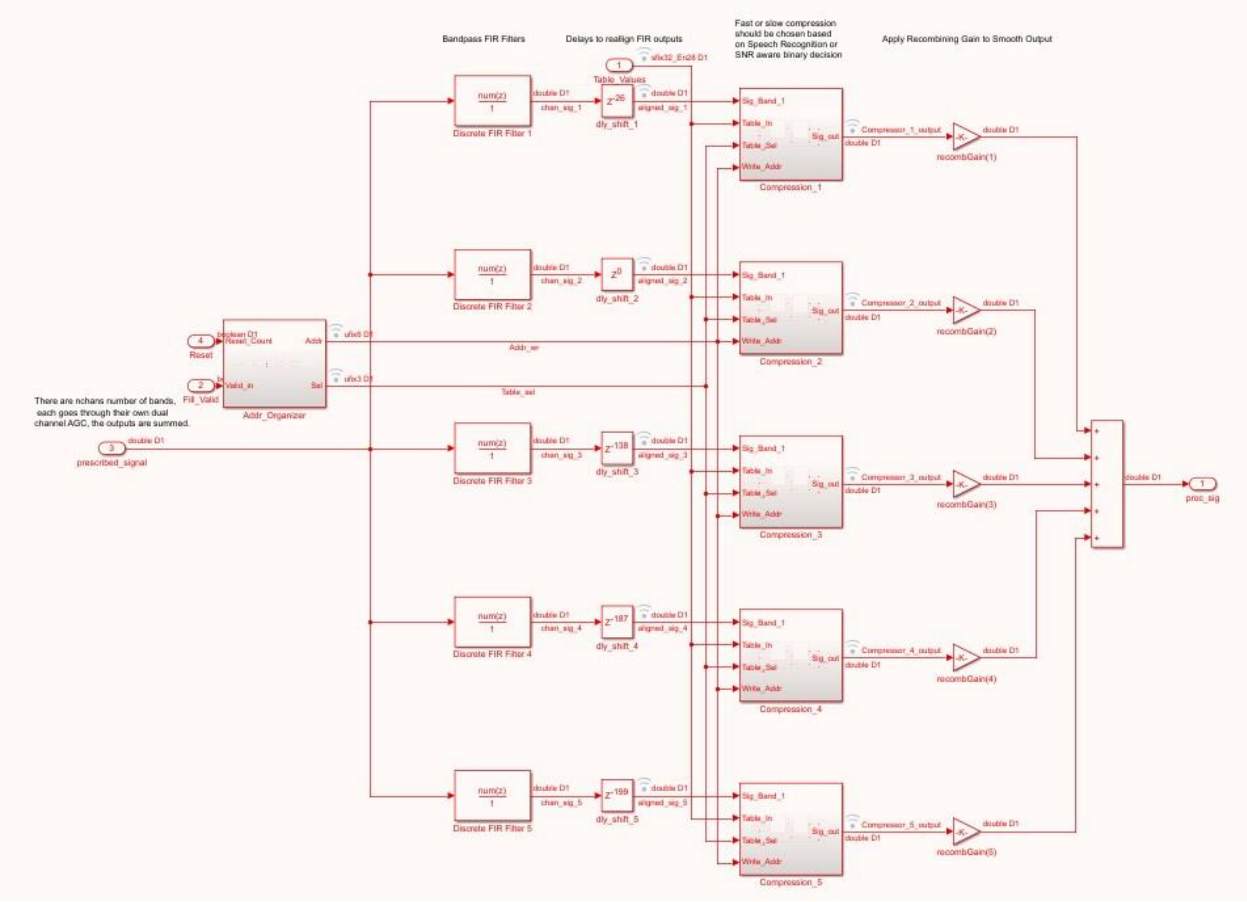

Figure 7 – Block Model For Band Compression and Reconstruction

One problem with real-time compression is that using decibels requires a non-trivial calculation. Translating into dB requires a $\log_{10}$ function, which with VHDL fixed point numbers is not innately possible and translation back into output voltage values requires an exponential function, also not innately possible with fixed point numbers in VHDL. As a work-around, all values and gains will stay in non-decibel form. To emulate a log function, a lookup table is generated for each channel that contains 2^6 (64) points for the compression gain based on the input multiplied by the largest possible gain within that band from the bandpass filter and prescription FIR (to help avoid clipping).

At an input of the threshold multiplied by the max band gain, the compression gain is set to unity. At the maximum possible input multiplied by the max band gain, compression gain is set to the reciprocal of max band gain. The lookup tables pre-calculated points are linearly spaced for ease of creating a hash function to identify the proper gain for any given input. Shown in Figure 8 is both a linear axis and log axis version of the lookup table with identical values. The linear axis version shows how linear interpolation can be applied to estimate output values for any input.

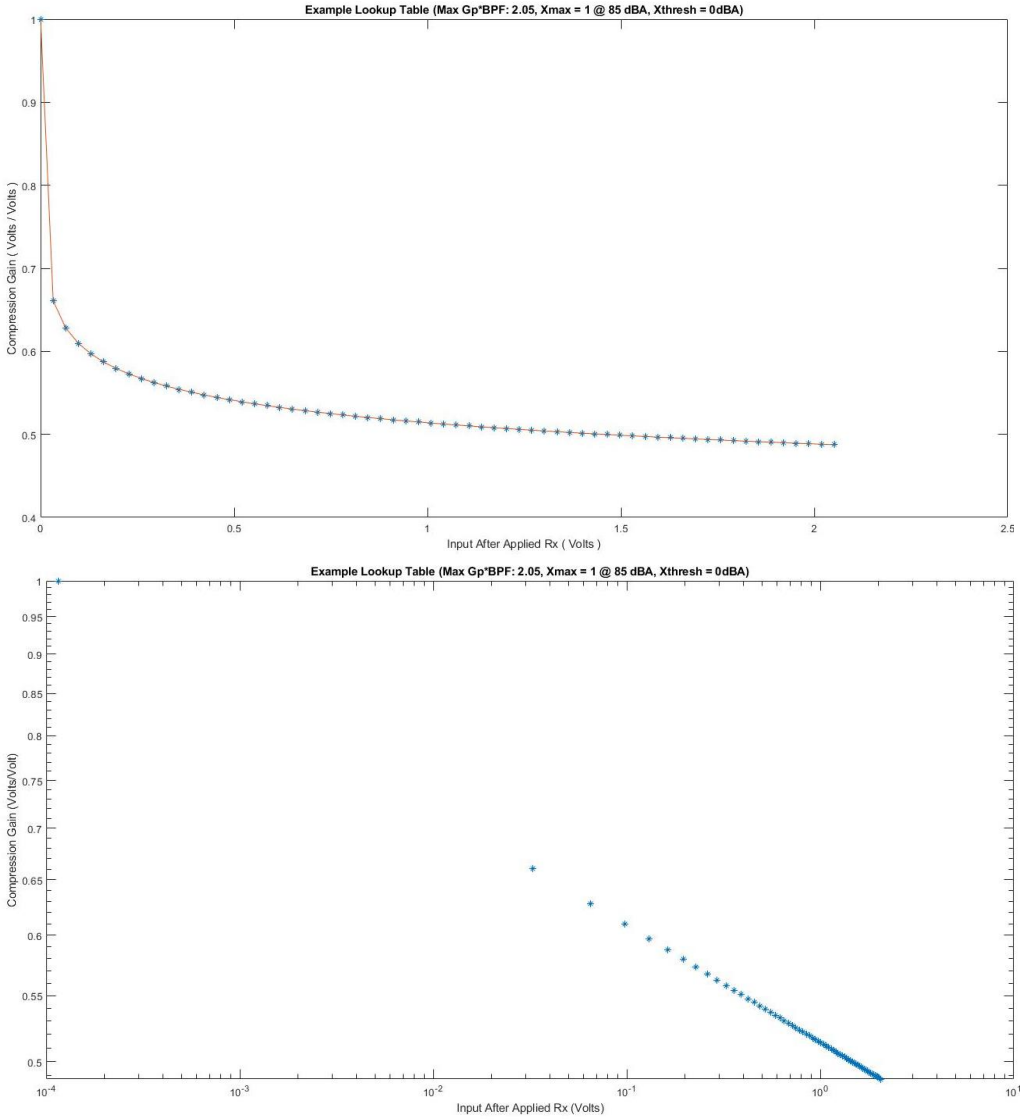

Figure 8 – Example Linear Lookup Table

It is important to identify the issues with this lookup table. First, linear interpolation is least accurate here when considering low-volume inputs. This is most obvious when looking at the log axis plot, where the open space between the first two points is the comparable as the distance every other point up to the maximum input is held in. This is a natural weakness of forcing a log scale onto linearly spaced points. This lack of data points affects sounds levels up to average talking volume, with an error of up to 40%.

To work around this, the addressing was changed to use **logical shifts left** to determine the power of two closest to the input, then the next M bits are sliced to make the N+M full address. **This type of hash function allows a $\log_2$ spacing of the precalculated points in the lookup table** and avoids the use of log functions by taking advantage of the properties of binary numbers. Powers of two are bound between 0 and N_bits, M_bits provide additional precalculated points between powers of two. Since the threshold value will most likely be set to 0dBA, or 5.6234 e-5, no more than 15 powers of two are needed. Conveniently, this allows us to use 4 bits for N, the first part of the address in the lookup table. M_bits is then a variable defining the number of bits sliced from the input after the first 1 to determine the size of the lookup table.

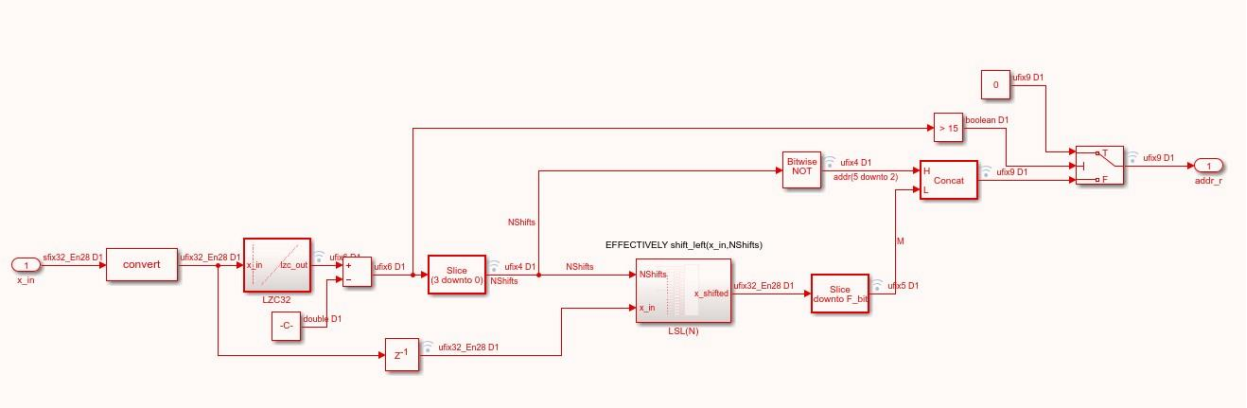

Figure 9 – Log Addressing Block

Two main blocks are used to determine the address. The first (LZC32) counts the number of bits in the fixed-point number before the first 1. In this example there are 32 word bits and 28 fractional bits, this count is subtracted by 3 so that a value greater than 1 yields an N of 0, and a value of 2^{-N} results in a value of N. The second block then shifts the result left N times, and the output of that block has bits 28 through 28-(M_bits-1) sliced to make up the second part of the address. In Figure 9, M_bits = 5, for a total of a 9-bit address in a table of 512 points. If the result of N is sent through a bitwise NOT block, the addresses in the table will be organized from least to greatest, between 2^{-15} and a number bound by 1 and $2\text{-}2^{-N\_bits-M\_bits}$. This range accepts both X_thresh (5.623 e-5) and the predicted maximum input of 1.

If this method is used to determine the address of any given input |x_in|, the error is drastically reduced in lower amplitude inputs. Figure 10 shows the results compared to the desired log function, with a table size of 64, both on a linear scale x-axis and a log scale x-axis.

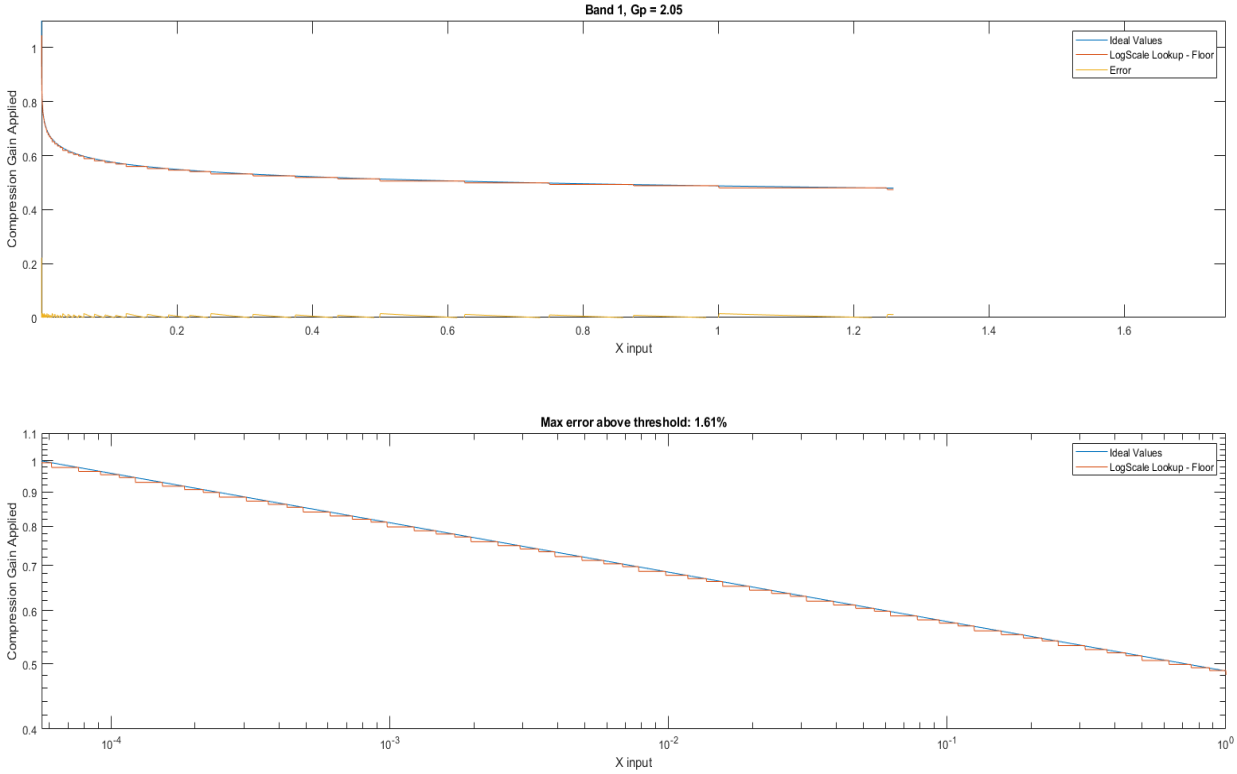

Figure 10 – Showing Error of Log-scale Floor Lookup Table

Linear Interpolation

The maximum error of the lookup table estimate above and desired log value is 1.61%, or 5 fractional bits of precision. To reduce this error even further, linear interpolation can be applied between two points in the lookup table. The blocks shown below show how this is implemented by copying a second lookup table, and increasing the address by 1 to read from two tables at once, providing a gain below and above the ideal output. The value for x_low is then calculated through the sum of 2^{-N} and the M value shifted to the correct power of 2. delta_x' = 1/(x_high – x_low) is found through logical shift lefts of the number 2^{M_bits}. This is identical to the reciprocal of the difference x_high-x_low, verified to be bit true through a full testing of all possible lookup entries.

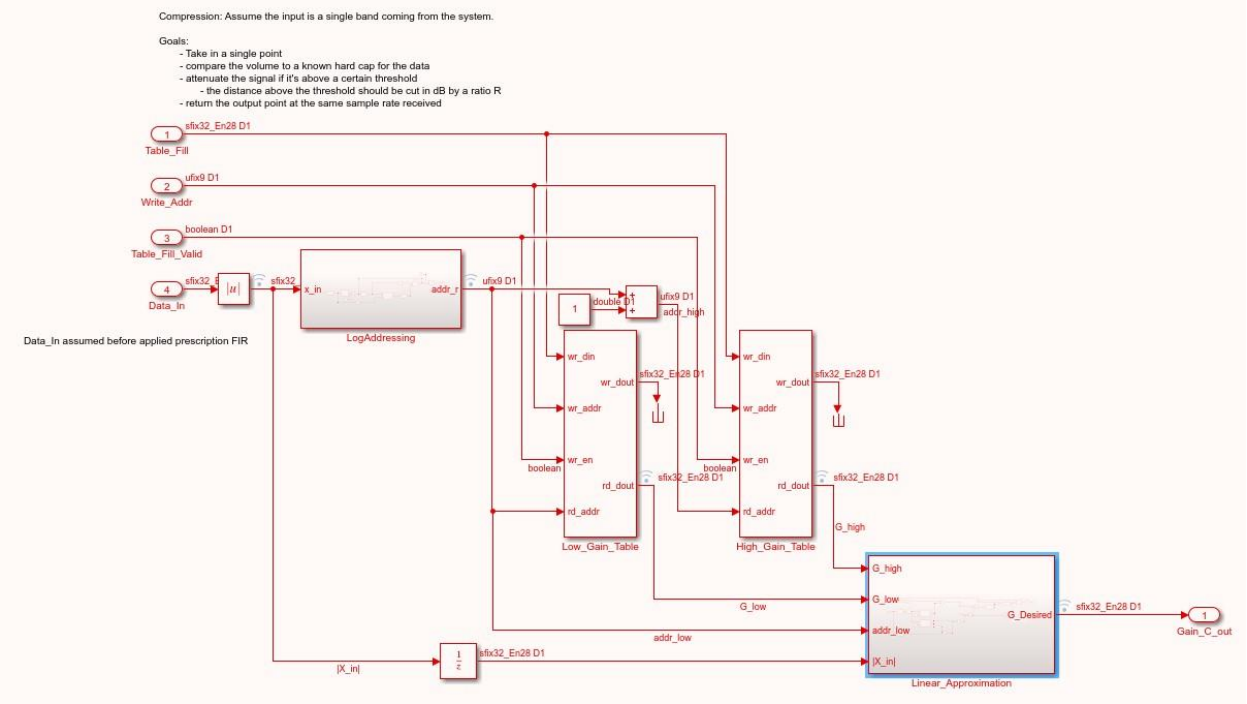

Figure 11 – Use of 2 Tables for Linear Interpolation

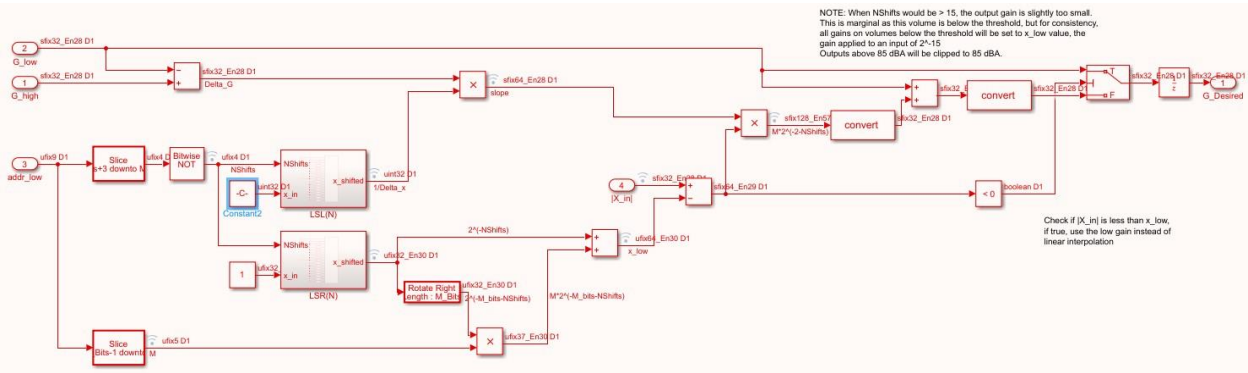

Figure 12 – The Linear Interpolation Block

To show the effect of table size on error, a sweep of volumes was run with varying M_bits between 0 and 12, or table sizes between 2^4 (16) and 2^{16} (65536) shown in Figure 13. These tests used linear interpolation, as this will be the model used moving forward.

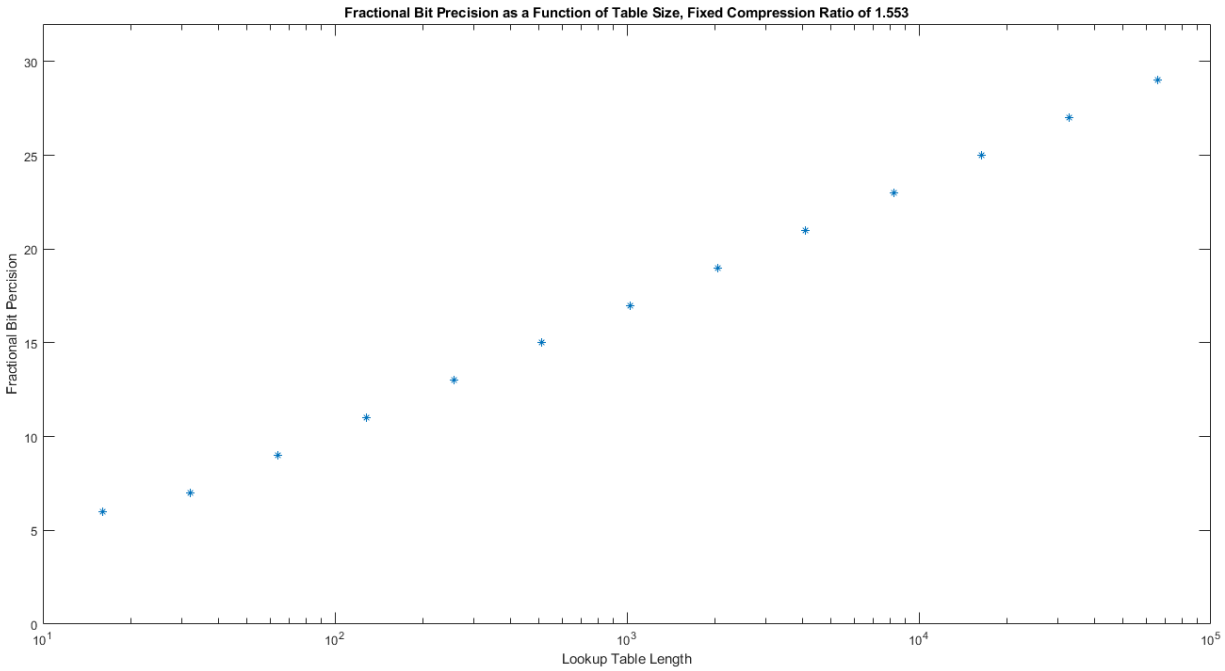

Figure 13 – Fractional Bit Precision as a Function of Table Size

To show an example case, below is a plot of the gain as a function of input of 5 channels with a table of 64 points, and their respective ideal gains. Dots show location/value of precalculated points.

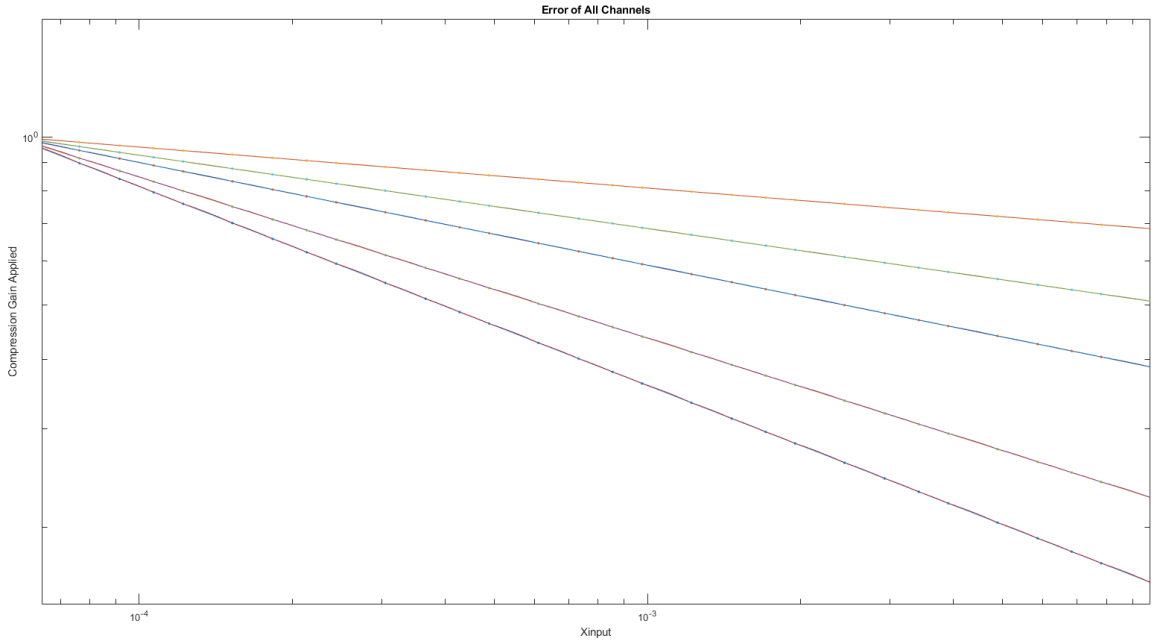

Figure 14 – Example of Linear Interpolation Accuracy

Error increases slightly as the compression ratio grows steeper. To show the effect of this behavior, Figure 15 describes the fractional bit precision as a function of compression ratio, which is based on the max prescription gain applied to a band of frequencies.

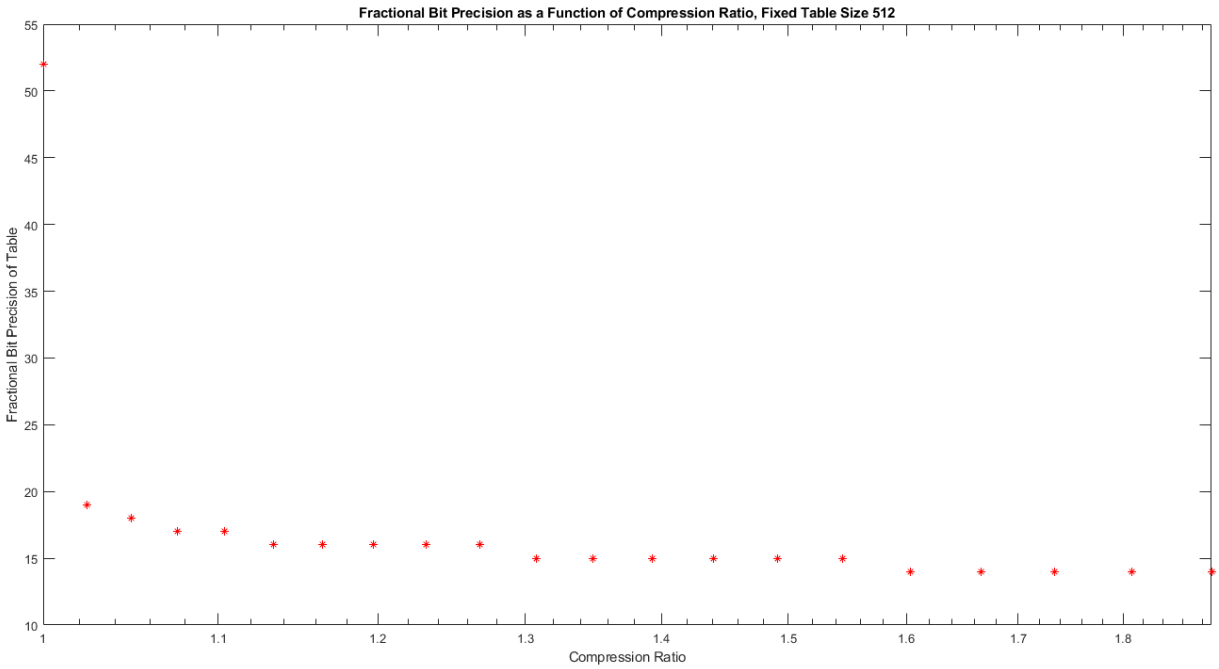

Figure 15 – Fractional Bit Precision as Compression Ratio Increases

This behavior can be explained by how as compression ratio steepens, the slope of the log function applied changes more rapidly, thus linear interpolation becomes less accurate. Due to the setup of the model mapping incoming sounds to a desired range of volumes as a function of frequency, it is unreasonable for compression ratio to be 2 or above, as this would mean the prescription applied is greater than 40 dB, or 100x in a given band of frequencies. While possible, this is a fringe case, and the effect on fractional bit precision is minor enough to consider a table size of 512 accurate to at least 14 fractional bits **(-84.3 dB).**

Improving and Applying the Frequency/Volume Map

Both a prescription and dynamic audio compression are applied in this model, and to keep calculations as simple and consistent as possible, everything must be considered.

First, applying the prescription before compression makes sense to avoid clipping, and to properly attenuate signals amplified by the prescription. However, the compression ratios are already calculated with awareness of the frequency response of the bandpass filters and prescription FIR. With some minor changes to the compression ratios and thresholds, it is possible to apply the prescription after dynamic compression, massively simplifying the lookup table and addressing schemes of log-scale functions. With an input bounded by -1 and 1, lookup tables only need values ranging between 2^{-15} and 1, 2^0. This fits perfectly with a 4 bit address. If prescription gains were applied before compression, the lookup tables would need to have |x_in| account for this, and either divide by the maximum possible gain between the frequency responses of the prescription and bandpass, or have a more complicated function for determining the x_in value associated with each precalculated point in the lookup table. This would also increase the difficulty of linear interpolation, as the function for determining delta_x' wouldn't be able to use only properties of binary numbers and powers of 2 to find the slope *without the need for division*. The logical shifts left as a substitute for a reciprocal function only works if the input is exactly a power of 2.

By changing the order of compression and prescription application, the frequency volume map remains identical if the ratios and thresholds for compression are changed accordingly.

```
Comp_Threshold = min_audible_thresh;

Comp_Ratio =
      ln(X_in_max/min_audible_thresh) /
          ln(max_output/(min_audible_thresh*max_band_gain));
```

NOTE: These tests showed that audioread/audiowrite have some minor discrepancies when writing very low amplitude signals. To avoid this, the signals used were saved to a .mat file and read directly for testing.

The frequency response of multiple volumes yields the following graphs in Matlab:

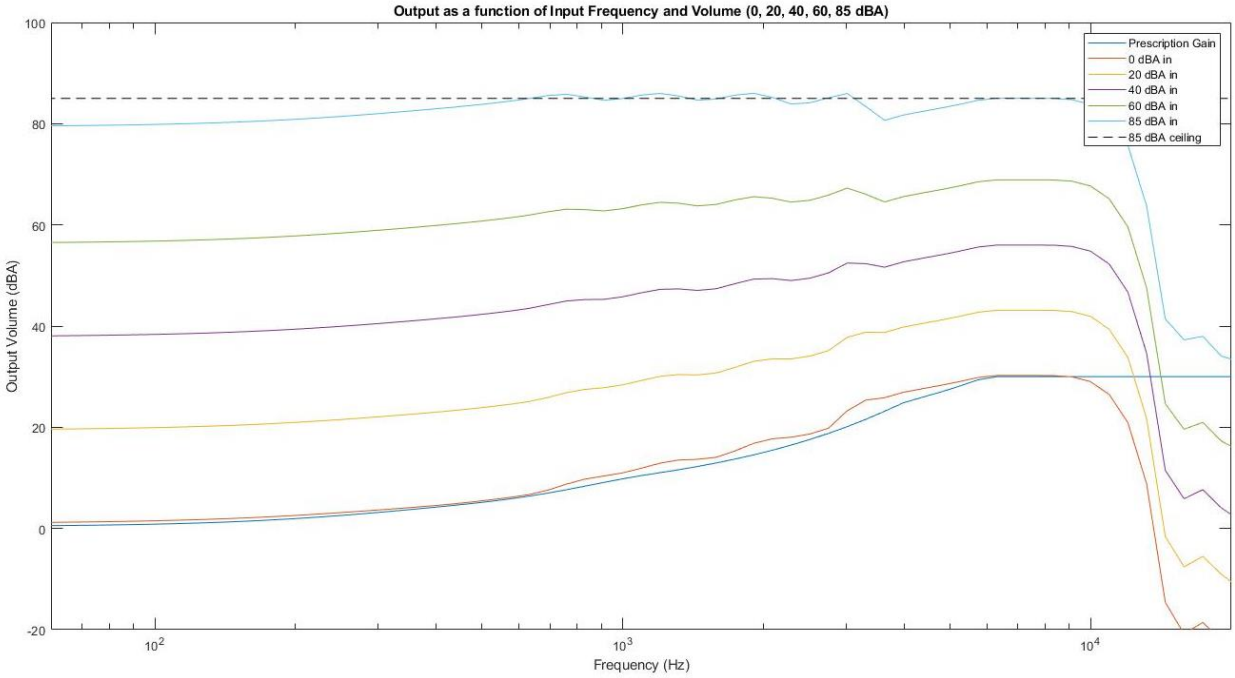

Figure 16 – Frequency Volume Map, Prescription First

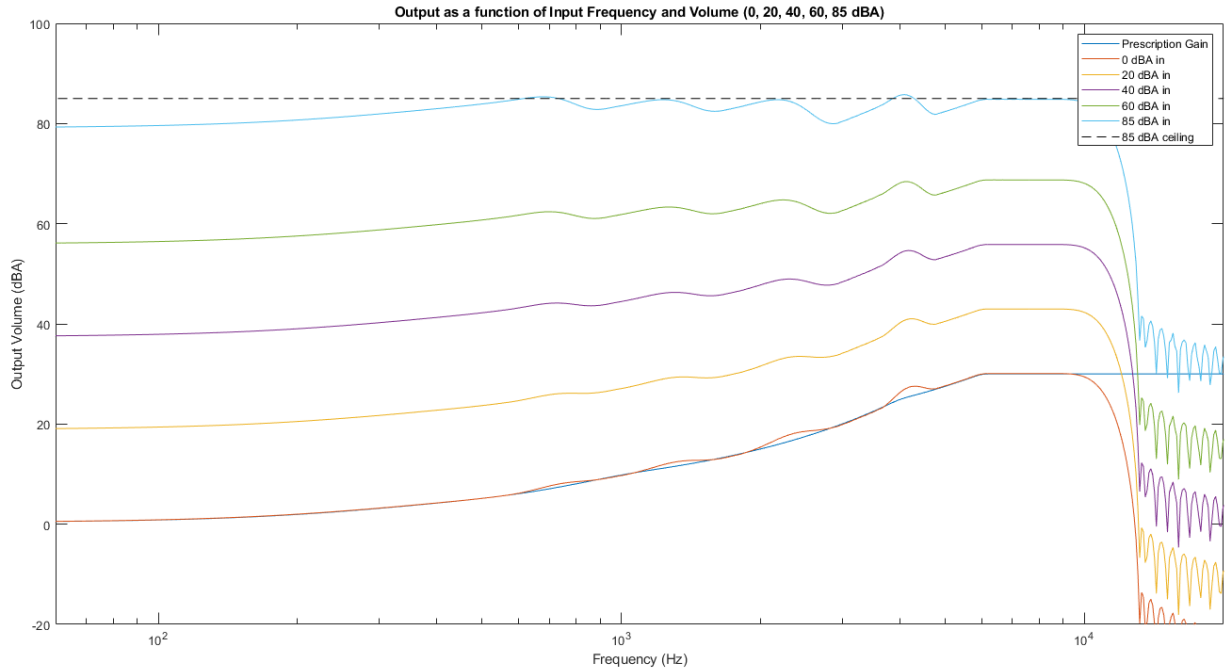

Figure 17 – Frequency Volume Map, Compression First

The Simulink model's frequency volume map was found by sending a set of frequency sweeps through the system at amplitudes equal to the tested Matlab amplitudes. Frequencies ranged from 100 Hz to 10 kHz, in a time of 4 seconds at a 48000 Hz sampling rate. The same FIR filters were then implemented in Matlab with fixed point values, using exactly the same input file. The amplitudes of the response were found and graphed.

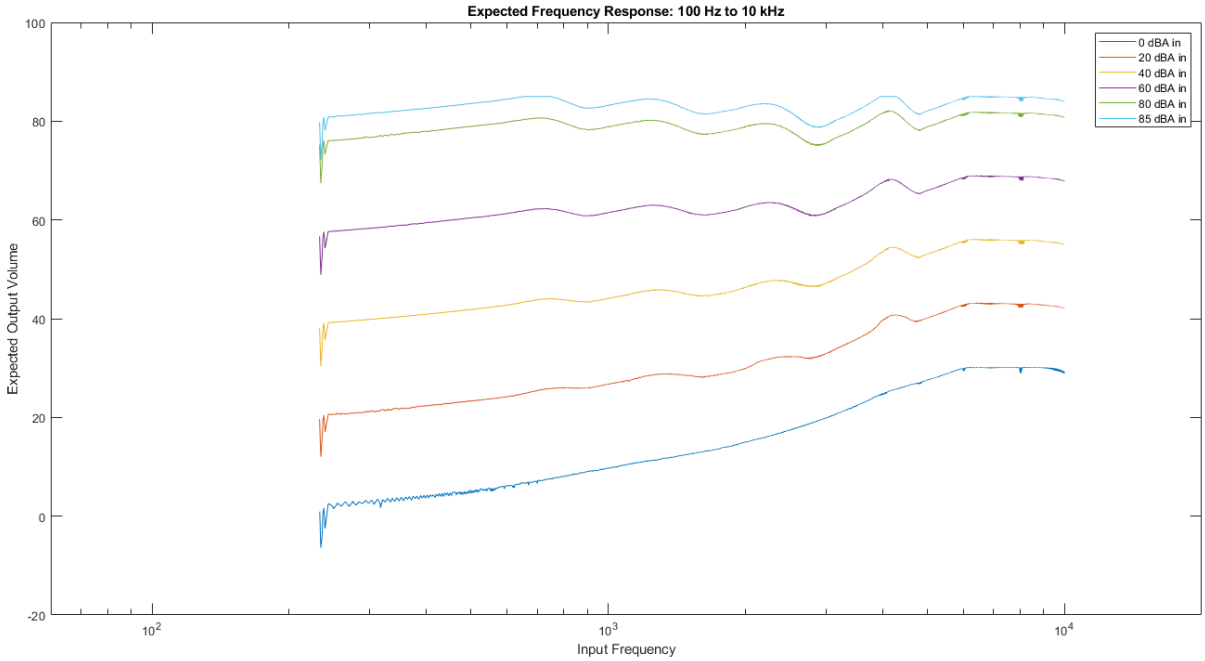

Figure 18 – Matlab Fixed Point Frequency Sweep Test

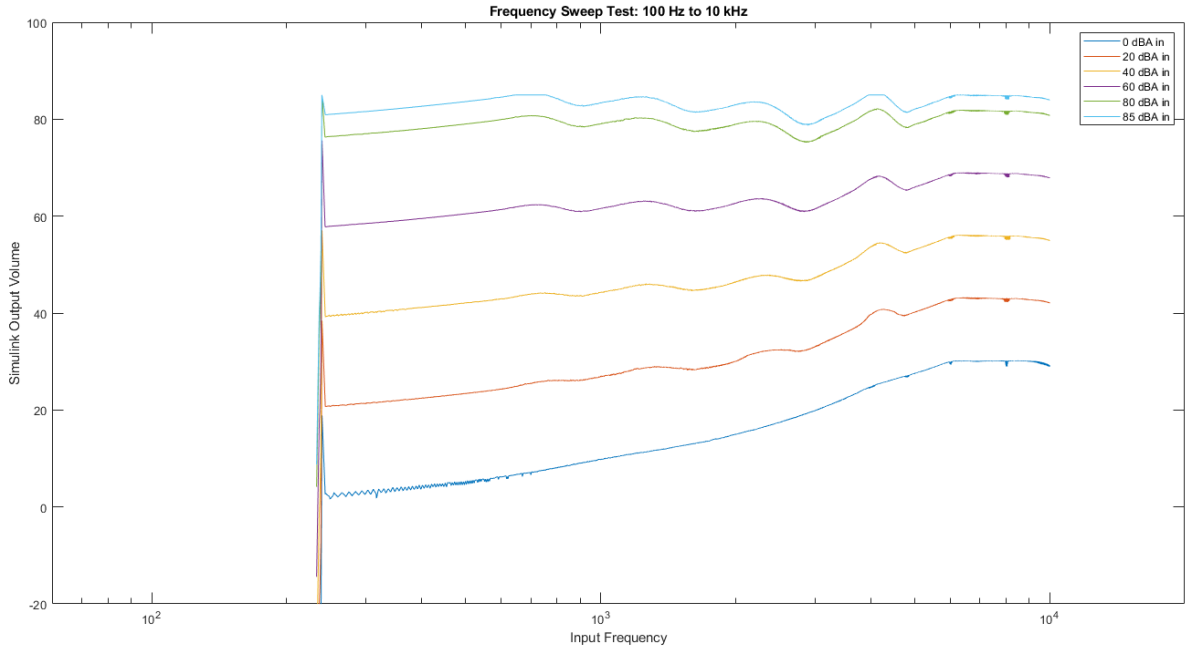

Figure 19 – Simulink Fixed Point Frequency Sweep Test, Lookup Table Size 512

The Simulink response has some peaking on startup, which has been mitigated through clipping gains and limiting compression gain between 0 and 1. What's left is the transient response of the FIR filters kicking in once the lookup tables are filled. The Matlab code does not have to spend runtime filling the lookup tables, which is why the transient response there is different.

Error between the two graphs is plotted below, found by dividing the Matlab response by the Simulink response.

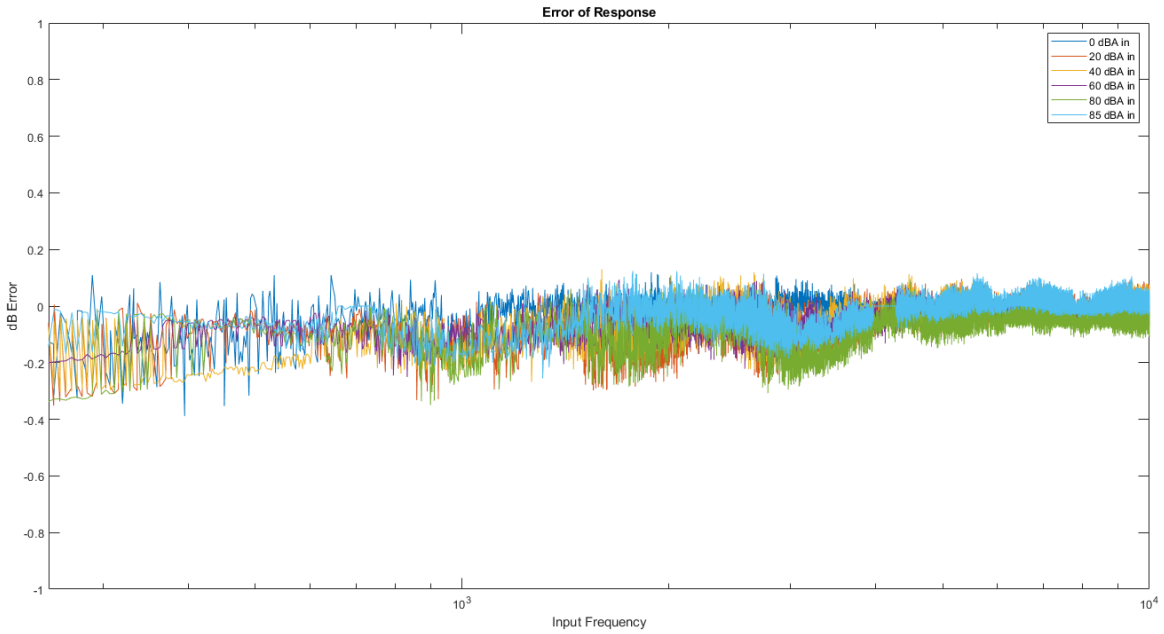

Figure 20 – Relative Error of Simulink Frequency Response

Since a table size of 512 has a maximum error of 2^{-14} in calculating gain, the error from the model is likely due to rounding errors of the system, and constant truncation of any add or multiply block used. The error here effectively decreases the sound to noise ratio (SNR) by 0.2 dB.

Attack and Release Times

The final test to discuss is the envelop function to apply attack and release times to compression gain. Attack and release times are defined as the time needed to change from 90% to 10% of the difference between starting and target gain.
Attack and release times are set in the startup of the system, and are constant for each band. With the current configuration, given in seconds:

```
t_att = [0.2   0.1  0.1  0.1  0.1];
t_rel = [2     1.5  1.2  1    1];
```

To implement this in Simulink, a block takes in the last applied gain, and desired target gain of the current sample. The difference is found and identified as positive or negative to choose envelope mode, and the magnitude is multiplied by a ratio r_att/r_rel . These ratios are based on the sampling rate of data (Fs) and the desired real time attack/release time of that band. The multiplied difference is then added to the target gain for attacking envelopes, or subtracted from the target gain for releasing envelopes. If the target never changes, the result is an exponential decay of the difference between the target and current gain.

Figures 21 and 22 show this behavior using a stepping volume input of constant frequency. Note there is some transient response of the FIR, but this will be ignored for the purposes of the test.

The first attack changes gain from 1.000 to 0.518. In this band, attack time is set to 0.2 seconds. The 90% value (0.9510) is reached at time 0.514 sec. The 10% value (0.5590) is reached at time 0.755 sec. The experimental attack time is therefore *0.241 seconds*, 20.5% slower than expected.

The first release changes gain from 0.491 to 0.518. In this band, release time is set to 2.0 seconds. The 10% value (0.4937) is reached at time 3.050 sec. The 90% value (0.5153) is reached at time 3.750 sec. The experimental release time is therefore *0.700 seconds*, 65% faster than expected.

The nature of sine/cosine waves forces the target to be constantly changing, as any input sample with amplitude below the threshold X_thresh (5.623 e-5) will be given a gain of 1. If in attack mode, this will likely change the envelope mode to release, slowing the attack times. If in release mode, the target will now be farther away from the current gain, causing the gain's envelope to be accelerated, decreasing release times.

Fortunately, the effect of these envelopes is largely for qualitative purposes, and the exact number for attack and release times is not as important as the behavior itself, and if necessary, the model for applying envelope functions can be modified to increase consistency of attack and release times.

Figure 21 – Single Band Gain Example Envelope

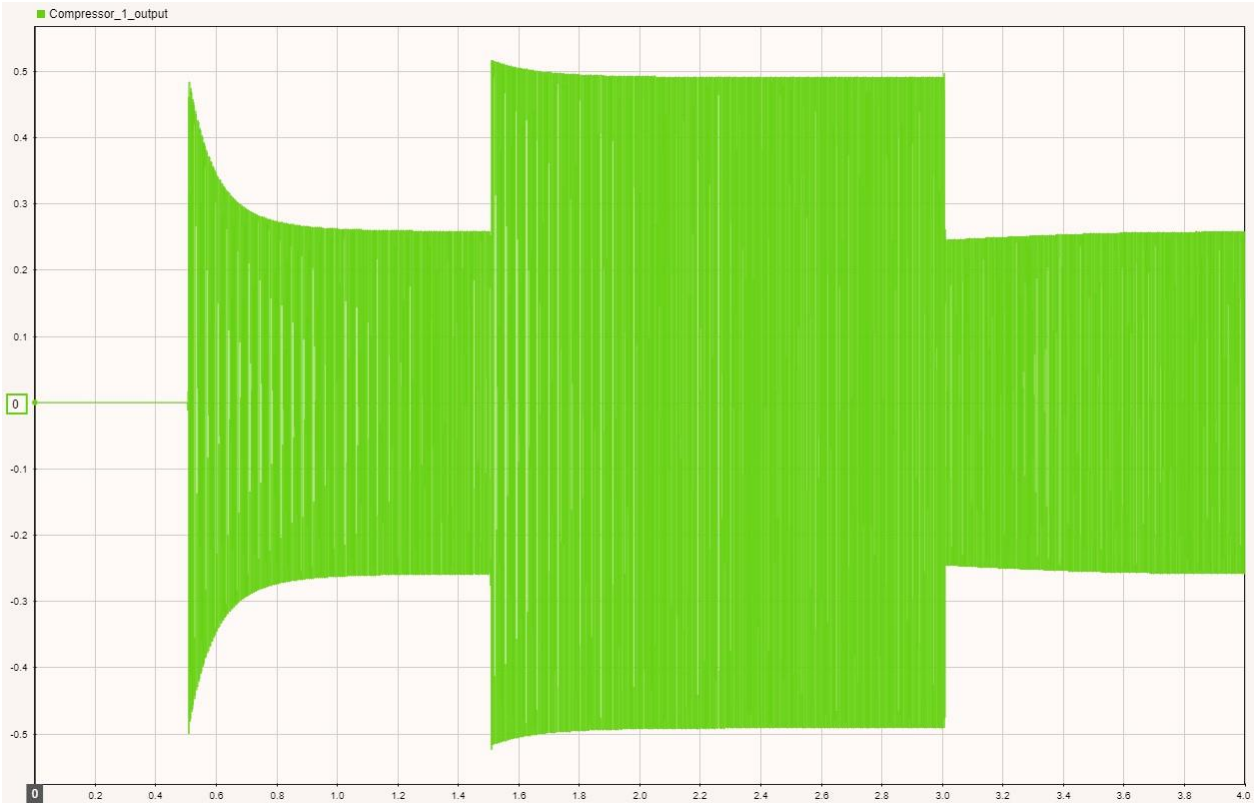

Figure 22 – Envelope Example Output (before Rx FIR)