Duke Univeristy

CS391

Digital Audio Plugin Design and Implementation

Final Report

Author:
Michael Seaberg

Supervisors:
Robert Duvall
Dr. Henry Pfister

May 4, 2017

Contents

1 Introduction

Over the past 20 years, the music production community has expanded in population and evolved technologically due to the development of the personal computer. Tasks which used to require a plethora of expensive, outboard audio gear can now be done by moderately skilled users with an interest and a laptop. The vast majority of recording and mixing techniques that were developed for use on analog gear can now be utilized in software programs called Digital Audio Workstations, or DAW's. Used in these DAW's are sub-applications known as "plug-in's", which are used to obtain the same type of audio processing that analog gear provides, at a fraction of the cost. The goal of this project was to develop a plug-in that can be used in a DAW to implement a specific audio effect. There are many different types of DAW's, each of which use a proprietary plug-in format, but the goal for this project was to create an Audio Unit (AU) plug-in which functions inside Apple's DAW, Logic Pro X.

There are many different types of plug-ins. Some plug-ins alter the frequency content of a signal, or use signal processing techniques like convolution to add reverb to a signal. Some plug-ins are modeled after hardware devices, such as Universal Audio's 1176 Limiting Compressor (a legendary compressor which has numerous different software emulations, each with slightly different characteristics), while others are developed from scratch, such as Antares' Auto-Tune pitch correction plug-in. For this project, I developed a compression plug-in. A compressor is a device that uses a dynamic range compression algorithm to control the difference in amplitude between "loud" and "soft" portions of an audio signal. Due to the resources required to be able to model a hardware device, an algorithm was designed for this plug-in by examining the ideal characteristics of a compressor and implementing them digitally.

1.1 Project Structure

This project was divided into three main tasks. The first task was to perform the necessary background research for the execution of this project. The second task was to use the knowledge gained from the research to design a compression algorithm. Finally, the algorithm was used to create a plug-in with a UI so that producers and engineers can use the developed algorithm to edit their audio. To design the algorithm for this plug-in, MATLAB's Simulink and audio system toolbox were used. After successfully designing and testing the algorithm using the tools provided, the JUCE application framework was used to develop a simple but robust plug-in that provides a user interface for access to the designed algorithm. While the original goal was to use the Audio Unit API to strictly create an AU plug-in, the project instead used the JUCE framework due its ability to easily produce multiple format plug-in's.

At the inception of the project, extra goals were set to improve the plug-in in the case that the project was finished before the end of the semester. These goals included, but were not limited to:

- Design a GUI for the compressor plug-in
- Modify the compression algorithm to emulate an existing hardware compressor's circuitry
- Using the base structure of the compressor plug-in, create an equalizer plug-in that is used to alter the magnitude of certain frequencies in a piece of audio

As discussed later in this report, the original algorithm was modified to introduce "soft" distortion of the signal as an extension of the project. Additionally, a rudimentary GUI was developed to enhance the user experience.

As a note to the reader, all documents and code related to this project reside in a GitHub repository, the address of which may be found in the bibliography at the end of the report at[8].

2 Background

A compressor is a form of a dynamic range processor. These processors most often utilize a non-linear function which applies a time-varying gain to a form input signal, in order achieve attenuation of a desired section of audio [1]. Compressors map the dynamic range of an input signal to a smaller range, effectively reducing the loud parts of a signal while simultaneously leaving the quiet parts untouched. On the opposite side of the spectrum, we have a dynamic range processor called an expander, whose function is the exact opposite. An expander attenuates the quiet parts of the signal, while leaving the louder parts untouched. While both are useful in their own ways, compressors are regarded by the music industry to be one of the most important tools for creating a well-rounded mix.

2.1 Compressor Controls

Every compressor is constructed with the intent of user manipulation due to the wide variety of audio signals that need to be processed. While the actual amount of user control varies depending on the exact implementation of the compressor, all compressors rely on certain fixed or variable parameters as input to the various stages in their signal path.

The first, and probably most important control is the *threshold*. The *threshold* defines the level (in dB) above which the compressor is active. In other words, the level

will be reduced when the signal goes above this *threshold*. Figure 1 shows an example of a characteristic curve. As seen in the figure, the output is a non-linear function of the input, which gets reduced at levels above the *threshold* by a factor determined by the *ratio*.

The *ratio* control determines the amount of compression that is to be applied when the input signal crosses the *threshold*. The *ratio* is represented in terms of input-to-output magnitude. A $1:1$ *ratio* signifies that there is no compression being applied, and the input is simply mapped to the output. A $3:1$ *ratio* signifies that for every 3 dB increase in the input signal, the output signal will only increase by 1 db. Logically, as the ratio approaches infinity, the output signal above the threshold will be limited to the level at which the threshold is set at. This is resonably known as "limiting", and is an extreme form of compression. The $10:1$ curve in Figure 1 shows an example of this.

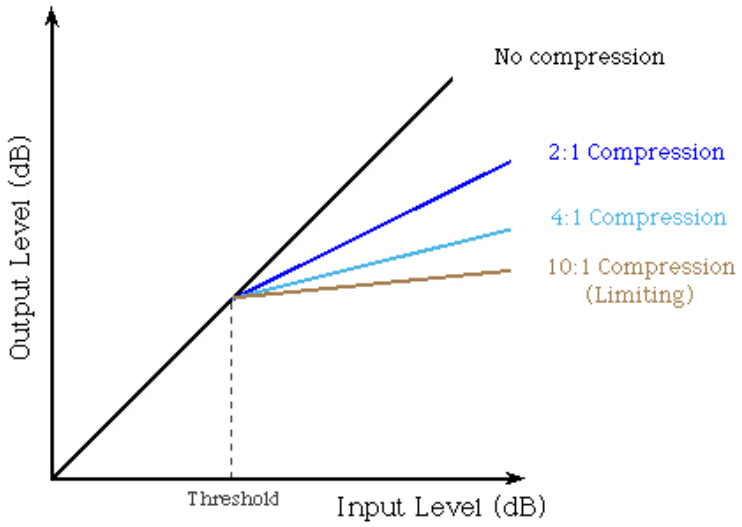


Figure 1: Compressor Characteristic Curve [2]

Because instantaneous gain reduction has consequences in both the analog and digital domains, compressors express a need for time control. The *attack time* of a compressor is the amount of time which the device takes to reduce the gain to the specified level *after* the input signal has crossed the *threshold*. The *release time* is the amount of time that the device takes to restore the gain to unity after the signal has fallen back below the *threshold*. Both these parameters are typically represented in milliseconds, with the latter having a wider range of control.

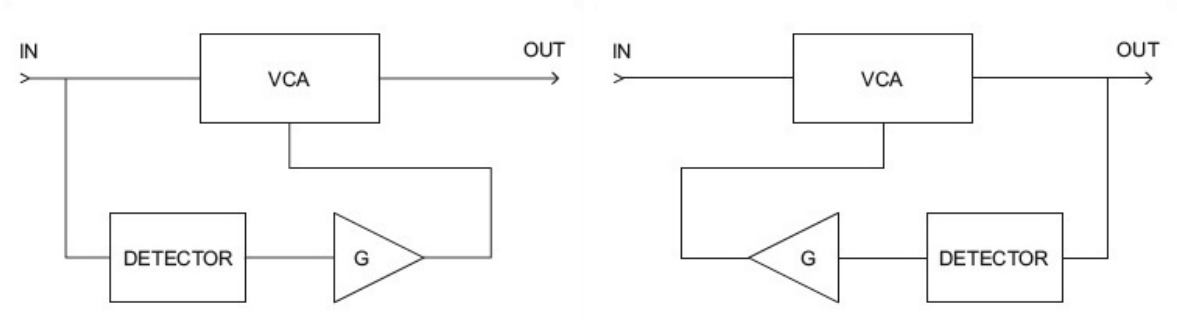
Another control that allows for smoother application of gain reduction is the *knee width* control. As shown in Figure 1, the transition from the uncompressed signal to the compressed signal is "sharp", resembling a piecewise function. The *knee width* control can be used to turn this, what is known as a "hard-knee", into a more rounded transition, known as a "soft-knee". Controlling the *knee width* essentially is controlling the bend that exists in the response curve; to achieve a more natural and transparent compression sound, a greater bend, or "soft-knee" is desired.

After all is said and done, the entire level of the input signal may be significantly reduced depending on the amount of compression applied. To remedy this, a *makeup gain* control is included to bring the overall level of the output signal back up so that it can more closely match the level of the input signal.

2.2 Compressor Anatomy

A compressor uses the previously described parameters as inputs to the functions in its system. As discussed in the design section, there are many different choices to be made regarding the organization and functionality of a compressors several stages, but all stages serve the same general purpose regardless of the exact compressor implementation.

As an input signal enters the compressor, it is split into two copies. One copy goes to a variable gain amplifier while the other goes to what is called the sidechain. The sidechain is responsible for controlling the variable gain amplifier in order to reduce the level of the input signal when desired. To accomplish this, it uses the input signal to create a control signal by using a gain computer along with other necessary devices. Figure 2 shows how the sidechain can be implemented into the compressor, either as a feedforward (Figure 2a) or feedback (Figure 2b) device. This is another design choice that must be made, as this basic setup heavily influences the overall sound of the compressor.



(a) Feedforward Design (b) Feedback Design

Figure 2: Basic Block Compressor Implementations [3]

2.2.1 Gain Stage

In an analog device, the gain stage operates using a voltage-controlled amplifier. There are many different ways to build a VCA with analog components, but the functionality remains the same across all implementations. The VCA takes in a control signal and an input signal, and outputs the signal that results from applying the control signal to the input signal. As compressors are refined to exhibit "ideal" characteristics, analog VCAs can become very complex, largely due to the exact specifications of the many discrete components which they are made of. In the digital domain however, we can model an ideal VCA much more easily. In the log domain, we can specify the gain stage using this equation below, where y represents the output signal, x represents the input signal, c represents the control signal, and M represents the applied makeup gain. In the linear domain, this equation equates simply to the product of each of the terms instead of the sum.

$$y_{dB}[n] = x_{dB}[n] + c_{dB}[n] + M_{dB}$$

2.2.2 Gain Computer

The gain computer is in charge of determining the amount of gain reduction that is to be applied to the input signal by formulating the control signal to be used in the gain stage. To determine the amount of reduction applied, the gain computer uses the threshold, ratio, and knee width parameters. The control signal operates using the equation below as a "guideline".

$$R = \frac{x_G - T}{y_G - T} \text{ for } x_G > T$$

The variable R stands for the compression ratio, T for the threshold, and x_G and y_G for the inputs and outputs of the gain computer, respectively. This equation can rearranged to be put in terms of the output as shown in the piecewise equation below.

$$y_G = \begin{cases} x_G & \text{for } x_G \leq T \\ T + \frac{(x_G - T)}{R} & \text{if } x_G > T \end{cases}$$

To create a smoother transition, we can use the knee-width parameter. Introducing it into the equation above will create an additional, quadratic function for the values centered around the threshold, with range specified by the knee-width.

2.2.3 Level Detector

The level detector is also included in the sidechain, with its exact position dependent on the implementation of the compressor. It is used to provide a smooth representation of the signal level in order to prevent the output signal from being generated with the unpleasant artifacts can come from instantaneous changes in level. To generate this representation, it uses the specified attack and release times as inputs to a smoothing detector filter. While this filter is represented by an RC network in an analog circuit, the behavior of this filter in the digital domain can be simulated by using the one-pole filter specified in the equation below.

$$y[n] = \alpha y[n-1] + (1-\alpha)x[n]$$

The α parameter, defined below, is dependent on a time constant τ which is specified by the attack/release time set for the system. This parameter represents the exact amount of time that it takes for the system to reach $1 - \frac{1}{e}$ of its final value.

$$\alpha = e^{-1/\tau f_s}$$

Using this equation it is easy to see that the attack and release controls have an influence on the change in gain. If the compressor is set to have an attack time of 25ms relative to a -15db threshold, it will take the compressor 25ms to reduce the gain of a 0dB input signal to -15dB. Several different versions of level detectors include the RMS detector, the peak detector, and the level corrected peak detector, discussed in more detail in [1].

3 Prior and Related Work

3.1 Compressors on the Market

The digital and analog compressors that exist today use many different combinations of the components previously discussed to achieve the same end result. In the analog domain, devices use different types of discrete components to achieve different characteristic sounds. For example, Universal Audio's 1176 compressor uses a FET to control voltage in the gain stage while Teletronix's LA-2A limiting compressor uses a light dependent resistor for the same function. More modern analog designs use specialized integrated VCA circuits. Digitally, compressors vary heavily depending on the approach that was taken to make them. Some software companies chose to design an algorithm from scratch, while other software

companies devote their resources to emulating analog devices as best as possible. Two possible ways that this is achieved is through part by part circuit analysis or using a "black box" approach and trying to characterize the whole system with a set of formulas. This is discussing in more detail in [4].

3.2 MATLAB : Audio System's Toolbox

MATLAB's Audio system Toolbox was used for this project to assist with the prototyping stage of the compression algorithm. The Audio System Toolbox provides tools for the design and prototyping of audio processing systems, emphasizing low-latency streaming in real-time [5]. It also includes automatic VST generation from MATLAB code, which is useful for DSP engineers who need to test end-product functionality but don't want to worry about having to implement a user interface while designing and debugging their algorithms.

3.3 JUCE Application Framework

The JUCE Application Framework is an open-source cross-platform C++ application framework, used for cross-platform development of applications, with heavy emphasis on GUI development for multimedia programs [6]. The Projucer is an application that can be downloaded that generates all the (cross-platform) files and code needed to begin creating a functional application right away. The JUCE framework was selected for use in this project due to its ease of use and its ability to generate plug-ins in multiple formats from a single set of code.

JUCE plug-in projects are arranged into two main components, the `PluginProcessor` and the `PluginEditor`. The `PluginProcessor` handles all of the audio/MIDI processing while the `PluginEditor` deals with the creation of the user interface for interaction with the processor. Only one processor can be present in a plug-in, while a plug-in can have multiple editors. Furthermore, it makes sense that it is the editors job to get information from the processor and not vice-versa. The JUCE API is very extensive and the code for this project uses many of it's classes. The API reference can be found at [7].

4 Implementation

After gathering enough background information regarding the specifics of compressor function, a compression algorithm was ready to be designed. Using [1] as a reference, the decision was made to implement a transparent compression algorithm with the least amount of artifacts as possible. This section of the report details the exact design of the compressor, and leads into the Simulink prototyping and JUCE final development of the plug-in.

4.1 Compressor Design

The complexity of the dynamic range compressor due to its non-linearity leads to many choices that must be made regarding its design. While the design choices made will be listed and discussed in detail, the analysis of other design choices and their benefits/consequences will not, and can be read about in detail in [1].

As mentioned before, there are two topologies for the side-chain, feedforward and feedback. Due to the facts that the feedback design is unable to provide lookahead functionality and that perfect limiting can't be achieved (due to the infinite negative amplification needed), the compressor in this project is designed using the feedforward topology. Additionally, because our design is digital, we don't have to worry about the high dynamic range problems that may occur when using a feedforward topology in an analog design. The block diagram for the design of this compressor is shown in Figure 3.

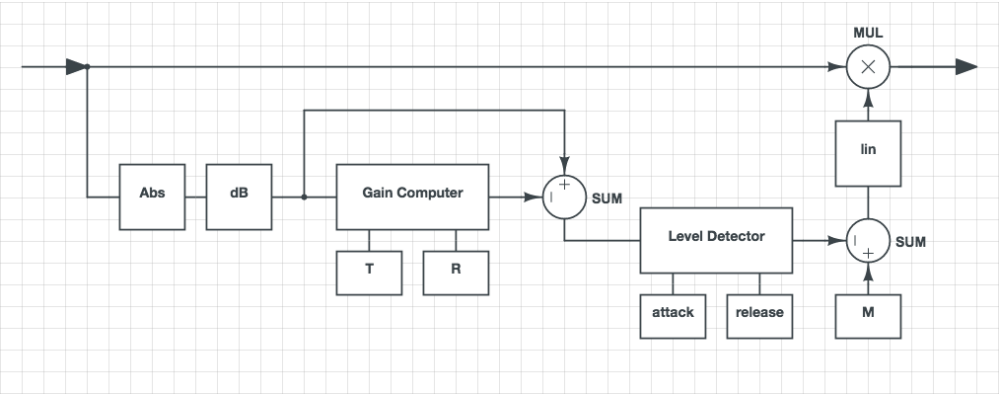


Figure 3: Compressor Design Block Diagram

As shown in Figure 3, the side-chain initially forms the control signal by taking the absolute value of the input. This is done to ensure that the peak detector is able to detect all of the peaks in the signal, since it can only measure unipolar signals.

The largest design decision is where to place the level detector within the circuit. To achieve smooth and and artifact free compression, the best location to place the level

detector in the circuit is within the log domain, yet after the gain computer. This design essentially puts the least amount of stress on the level detector as possible. Placing it after the gain computer ensures that the detector smooths the control signal instead of the input signal. A smooth release envelope is guarunteed due to the fact that the control signal returns to zero when it the input isn't being attenuated. By placing the detector in the log domain, it operates on "log values" instead of "linear values". This results in an exponential behavior that seems smoother to the human ear.

The exact implementation of the level detector was important to consider. Simulating an ideal peak detector proved to be problematic due to the incorrect peak estimation that happens because of attack and release times that are close in range to each other. However, because the design is digital, the attack and release activity of the compressor can be decoupled, which provides smooth, level-corrected performance on a wide variety of signals. The exact equation used to implement the detector is shown below.

$$y_L[n] = \begin{cases} \alpha_A y_L[n-1] + (1-\alpha_A)x_L[n] & \text{for } x_L[n] > y_L[n-1] \\ \alpha_R y_L[n-1] + (1-\alpha_R)x_L[n] & \text{for } x_L[n] \leq y_L[n-1] \end{cases}$$

4.2 MATLAB Simulink Prototype

In order to test the functionality of the previously described algorithm, the Audio Systems Toolbox was used to construct the compressor in MATLABs Simulink. The exact implementation varied slightly from the block diagram shown in Figure 3, and can be viewed in Appendix A. The code used to generate the gain computer and level detector blocks can be viewed in Appendix B. Debugging was accomplished by manually using Scope and Audio out blocks to view/listen to different parts of the side-chain.

4.3 JUCE Final Plug-in

After the algorithm 's function was verified in Simulink, it was able to be implemented in XCode using the JUCE application framework. For this plug-in, it made sense to use the project files that were generated by the Projucer, as the setup proved to be easy to work with. The `PluginProcessor.cpp` and `PluginEditor.cpp` files hold the bulk of the code for the application.

The first step was to correctly write the code for the algorithm. This was done in the `processBlock()` method of `PluginProcessor.cpp`. This method takes in an `AudioSampleBuffer` and uses a pointer to write to that buffer of data. Each sample in the buffer is processed sample by sample using a for loop that iterates through the buffer.

Buffer size is determined by the host DAW and is passed down through the processor to the `processBlock()` method. The code is organized so that for every block in the Simulink model (shown in Appendix A), there is a separate helper method which carries out its function. Problems that were solved through debugging included incorrect integer to float casting and pointer referencing.

The next step was to implement an interface that gave users control over all of the parameters implemented in the design (everything except knee width). The `PluginEditor.cpp` class was modified so that it would automatically generate sliders for any parameter that is represented by a float. This code was refactored into the `createControl()` method which creates the ratio setting buttons for the `AudioParameterChoice` object if the input `id` of the current parameter is `"r"`, and sliders for all `AudioParameterFloat`objects. Listener methods for each of these parameter objects were also written for `PluginEditor.cpp`, which extends the Listener classes for both the Slider and Button objects.

5 Extensions

Several of the planned extensions for this project were able to be explored. The UI was enhanced to allow the user to view the amount of compression on the audio in realtime. The algorithm was also altered to introduce "color" into the signal by using soft distortion.

5.1 UI

A rough version of the modified UI is shown in Figure 4 below. In addition to the sliders and buttons described in the implementation section, the plug-in now has a portion of its interface dedicated to displaying the audio which it is processing. As the audio "moves" through the compressor, it appears on the right hand side of the screen and moves to the left as time progresses. The blue waveform represents the output signal, while the gray unipolar waveform on the top of the screen displays the level of the audio that has been reduced from the signal.

Special classes were made to implement this feature. The `CompressorDisplay` class was constructed to hold any component that may have been included in the scrolling waveform section of the screen. The `WaveformVisualizer` class extends the `AudioVisualiserComponent` class, which allows for easy generation and manipulation of scrolling waveforms. The `CompressorDisplay` class holds two `WaveformVisualizer` objects, one for the blue waveform and one for the gray waveform. These classes proved to

be problematic due to the fact that they needed to be updated in real time. A timer callback function is set to repaint the component thirty times every second, but in order to get the correct version of the audio on the screen, the `CompressorDisplay` *and* the `PluginEditor` class needed to be repainted as well.

Figure 4: Final Plug-in UI

5.2 Distortion

Another extension of this project was to modify the compression algorithm to emulate an existing hardware compressor's circuitry. However, due to the complexity of this task and the resources required, this was unable to be accomplished. To add some "color" to the output signal, which is present in analog audio devices due to their imperfections, a gain stage was added to the compressor to boost the level of the input signal before it was compressed.

Distortion is another form of nonlinear effect, but it is much easier to implement due to the fact that it is memoryless. Distortion effects usually have fixed characteristic curves that operate on the same principles of compression characteristic curves. The block diagram for this effect is shown in Figure 5.

Unfortunately, distortion in the digital domain does not come without consequences. The unbounded series of harmonics present aliasing problems. Fortunately, there

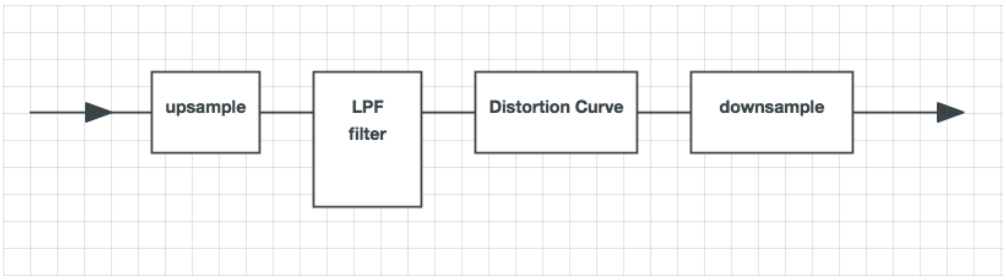


Figure 5: Distortion Design Block Diagram

are solutions to this. First the input signal is oversampled to extend the range of frequencies present in the signal. This signal can then be filtered through a low-pass filter to remove any of the high frequencies that would have been previously aliased. After processing the signal through the distortion, we can then downsample again to restore our signal to its original sample rate.

The characteristic curve used for this effect exhibits a "soft clipping" characteristic in order to achieve a tube-like saturation effect. An input gain control was added to the plug-in to add gain to the input signal on its way into the gain stage, which is included in the UI. The algorithm for this distortion effect was built right into the `PluginProcessor`'s `processBlock()` method, and utilized the `IIRFilter` class to implement the anti-aliasing filter. Separate buffers had to be made to store the oversampled signal, from which the compressor takes its input from.

6 Conclusions and Future Work

There are many future extensions for this project. Due to the lack of time and resources during the semester, I was unable to model the characteristics of my plug-in off of an actual analog compressor but with the rough framework of the compressor plug-in laid out, it could be possible to do so without diversion. Having the rough framework of a plug-in laid out also lends itself to designing other types of plug-ins, such as delay effects or equalizers. Additionally, it would be interesting to create a multi-threaded plug-in in order to handle designs that required a large amount of data manipulation. There are also specific refinements that I would like to make to the existing material in the plug-in. For example, a wet-dry knob would be useful for users to be able to blend the compressed signal with the original signal, which can achieve a more natural sounding compression. The UI is not as complete as I would have liked to make it, as there are other specific items that can be added to it. This includes but is not limited to a VU gain reduction meter or a frequency spectrum graph displaying the harmonics added by the distortion.

Turning the design from a MATLAB block-system into a piece of functional software proved to be challenging, but rewarding. The JUCE library proved itself time and time again to be a valuable asset for real-time signal processing by providing robust interfaces that could easily be adapted for use in any plug-in. Working with audio is a fun way to learn basic (or advanced) signal processing concepts, and designing a plug-in allows for a perfect blend with software design.

References

[1] Reiss, Joshua D, and Andrew P McPherson. *Audio Effects: Theory, Implementation, And Application*. 1st ed. Boca Raton, Fla. [u.a.]: CRC Press, Taylor & Francis Group, 2015. Print.

[2] *Compressor Input / Output Characteristic*. 2017. http://www.digital-redux.com/images/tech_4_2.gif.

[3] *Compressor Block Diagram*. 2017. https://www.gearslutz.com/board/attachments/so-much-gear-so-little-time/120800d1241911741-compression-feedback-vs-feedforward-picture-1-1.jpg.

[4] M. Lambert, "Plug-in Modelling : Emulating Hardware In Software", *Sound on Sound*, 2017.

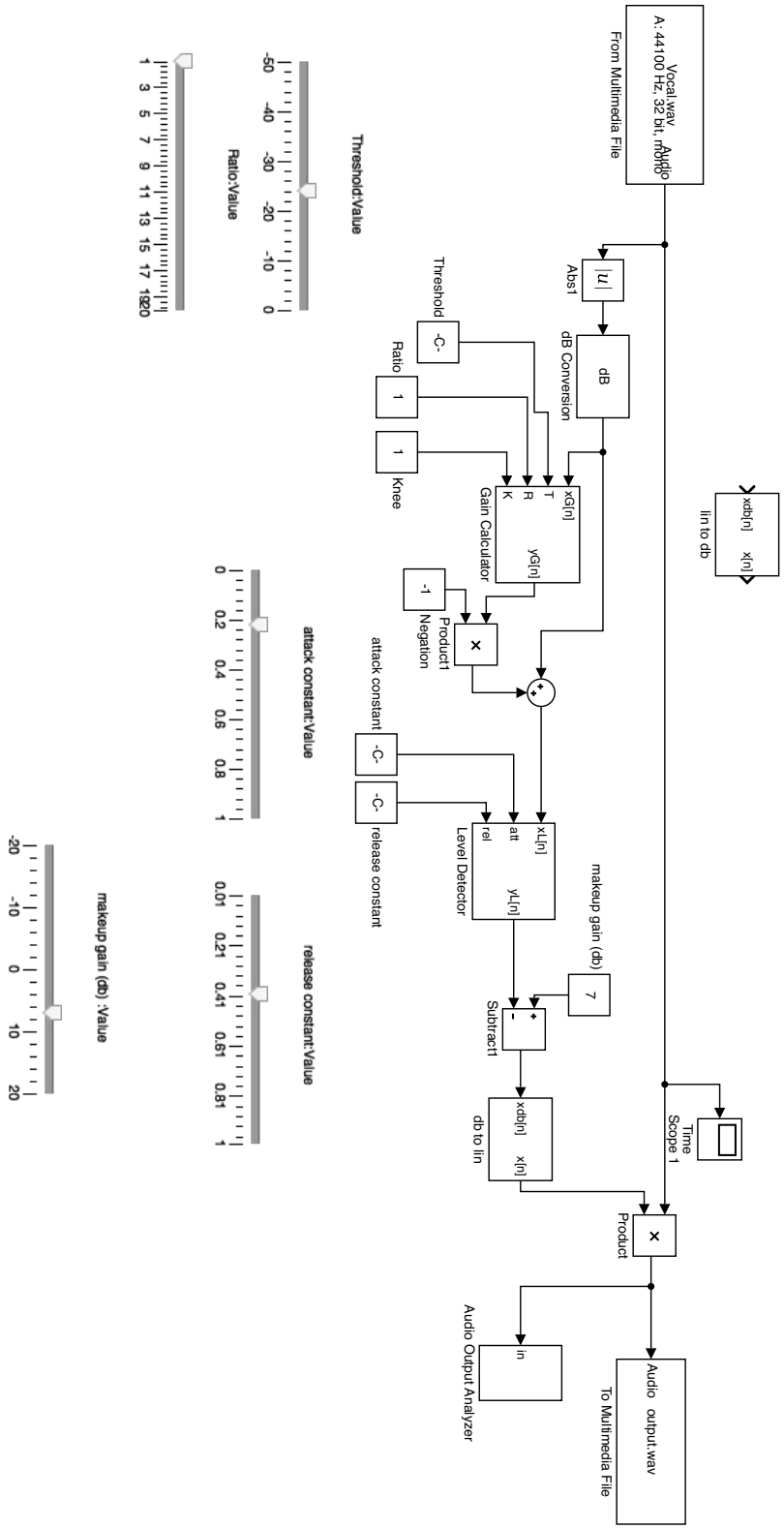
[5] "Audio System Toolbox - MATLAB & Simulink", *Mathworks.com*, 2017. [Online]. Available: https://www.mathworks.com/products/audio-system.html. [Accessed: 04- May- 2017].

[6] "JUCE", *En.wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/JUCE. [Accessed: 04- May- 2017].

[7] "Class Index — JUCE", *JUCE*, 2017. [Online]. Available: https://www.juce.com/doc/classes. [Accessed: 04- May- 2017].

[8] M. Seaberg, "michaelseaberg/cs391audioplugindesign", *GitHub*, 2017. [Online]. Available: https://github.com/michaelseaberg/cs391audioplugindesign. [Accessed: 04- May- 2017].

A Simulink Prototype



B Simulink Code

B.1 gainCalculator Code

```
function y = gainCalculator(x,T,R,~)
y = zeros(size(x));

for n=1:length(x)

   if x(n)<=T
      y(n,1) = x(n,1);
   else
      y(n,1) = T+(x(n,1)-T)./R;
   end
end
```

B.2 levelDetector Code

```
function [y,lastSample] = levelDetector(xL,attack,release,previousState
   )
Fs = 44100;
attackCoeff = exp(-1./(attack*Fs));
releaseCoeff = exp(-1./(release*Fs));
output = zeros(length(xL),1);
for n=1:length(xL)
   if xL(n) > previousState
      output(n,1) = attackCoeff.*previousState+(1-attackCoeff).*xL(n,1)
         ;
   else
      output(n,1) = releaseCoeff.*previousState+(1-releaseCoeff).*xL(n
         ,1);
   end
   previousState = output(n,1);
end

lastSample = output(length(xL),1);
y = output;
```