Pedal board using Python and Digital Signal Processing

Anant Chary

May 2024

1 Introduction

This is a project that attempts to emulate the analog pedal board of a guitarist through Python and Digital Signal Processing. It allows the user to input a .wav file (not just of a guitar) and process it through a "pedal board." They can also adjust parameters on each pedal using a sliders, emulating the various knobs that are found on analog guitar pedals.

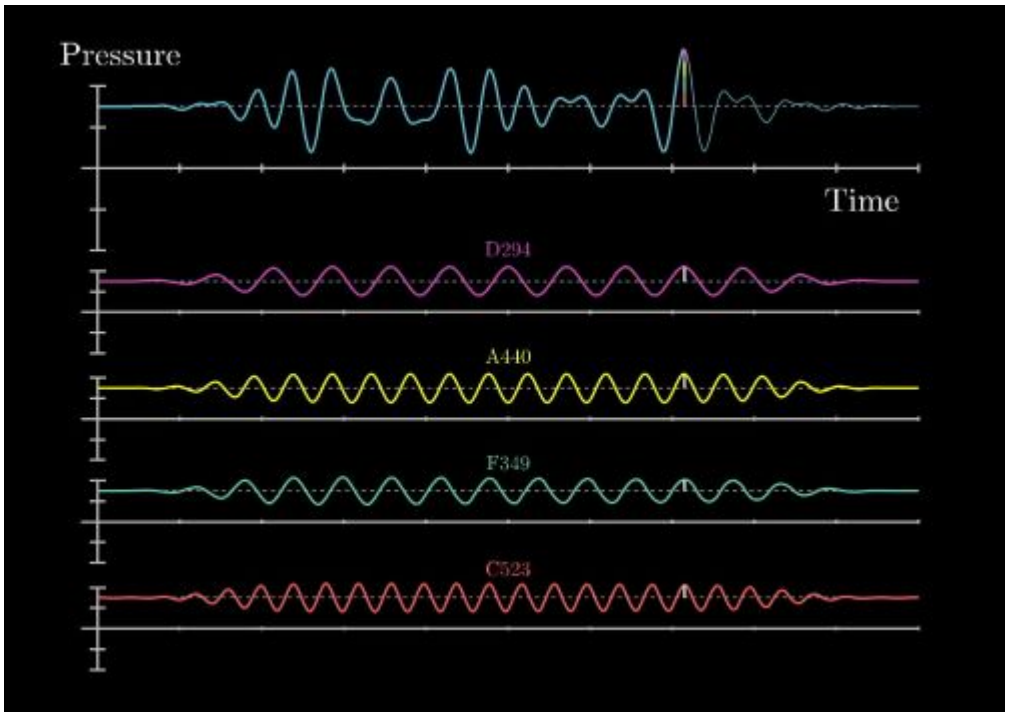
In this document we will go through each of the pedals and explain the workings and mechanisms behind them, along with the steps that we took to find them.

2 Boost

Our first pedal is the boost pedal, which is quite simple in result. It simply takes in a .wav file, and increases the volume by a specified multiplier. An analog boost pedal is also quite simple, often only having one knob which represents the voltage gain between the input signal and the output. To explain how we accomplished this, there is quite a bit of math we must get into. First, to simplify this operation, we will use a sin wave for an example rather than the more complicated wave we would get from a sound in a .wav file. The following equation is a general sin wave that can be used for such operations:

$$y(t) = Asin(2\pi ft)$$

where A is the amplitude of the wave, f is the frequency of the wave, and t is the elapsed time.



A representation of the Fourier Transform.

The simplest way to increase the volume of a signal wave is to increase the amplitude of the signal, that being the distance between the peaks of the waves and the x-axis. It is not so simple to increase the amplitude of a complex wave, so we will use a Fourier Transform to decompose the waves into sin or cosine waves. I will not be going into the specifics of the math behind this concept, but the final equation is the following:

$$\int_{-\infty}^{\infty} g(t)e^{-2\pi jft}\,dt$$

where g(t) is the function, j is an imaginary number, f is the frequency, and t is the time.

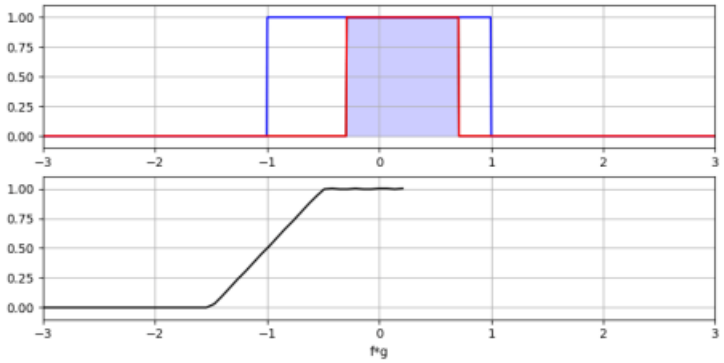
For our purposes, we will be using an algorithm called the Fast Fourier Transform, or FFT. This is closer to the discrete fourier transform, which is given by the following equation.

$$\sum_{n=0}^{N-1} g(n)e^{-2\pi jfn}$$

Using this operation, we increase the amplitude of each wave, then use the Inverse Fast Fourier Transform algorithm to create an output .wav file with the boosted audio.

3 Equalizer (Low Pass Filter)

The next pedal is a form of an equalizer, which is a filtering pedal. It is a low pass filter, a fitting name considering it lets the low frequencies pass through to the output while filtering out higher frequencies. This allows us to take our .wav files, and give them a darker tone quality, although we need to make the filter smooth out rather than hard cut at the cutoff frequency.



One representation of a convolution operation, with the red function being g(t) and the blue function being f(t)

One way to accomplish this is to use the convolution operation. This operation overlays one function, f(t) on top of another function, g(t) and creates a third function based on the amount of area of the second function that is within the area contained by the first function and the x axis. This operation is given by the following equation:

$$(f * g)(t) = \int f(\tau)g(t-\tau)d\tau$$

where the character $*$ gives the operation of convolution. In this expression, g(t) is usually the function that slides across f(t), although the functions could be swapped and the expression would be the same.

While this is one valid method for implementing the filter, the method that we chose is known as the moving average.

First, we separated the audio file into its channels, and applied the filter to only the first channel. Our parameter for this pedal that is set by the user is the cut-off frequency, which gives the frequency above which all frequencies are cut. We then calculate a ratio given by $\frac{cut-off\ frequency}{samplerate}$. We then use this ratio to find the moving average, given by the equation:

$$\frac{1}{k}\sum_{i=n-k+1}^{n} p_i$$

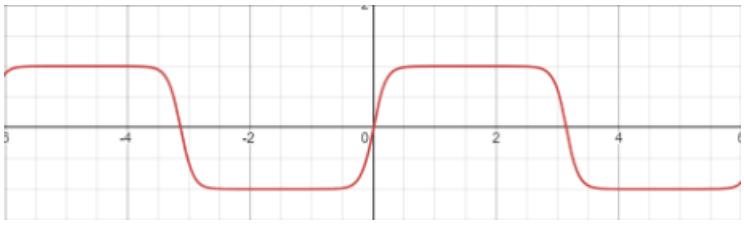
where k is the number of data points and p_i is the data point.

We used this to calculate the running mean and that gave us our final list that we compiled back into an audio file. This pedal would mainly be used to get rid of excess overtone or harmonic frequencies that are muddying the sound. It is not that lower frequencies become louder or higher instruments are cut out, but more so that the tone quality becomes darker and you hear less harmonics in the sound.

4 Distortion

For this pedal, we will need to perform clipping on our input wave. This makes it so that the wave will flatten out at a certain threshold, preventing values from being above it (or below if it is the lower threshold).

The function $y = tanh(Asin(x))$, where the sin wave represents our input wave and A represents the amplitude of the input wave, which is 1 in this case.



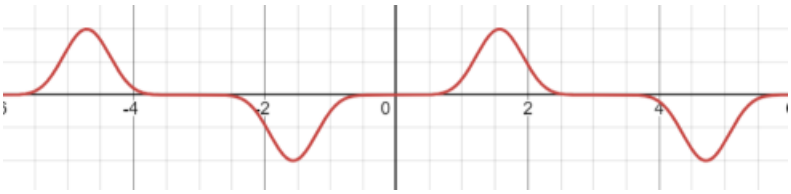
The function $y = tanh(Asin(x))$, where the sin wave represents our input wave and A represents the amplitude of the input wave, which is 5 in this case.

The way we decided to clip our wave was to pass our original wave into a hyperbolic tan function, which is represented by $tanh(x)$. This essentially causes the wave to clip as long as the amplitude is high enough, which is what we want. The greater the amplitude, the more clipping the function shows, which can be seen in the two figures above.

This begins to mimic an emulation of a square wave from the original wave, and while this is not as extreme as a true emulation of a square wave, since this is simple clipping, there is another pedal that is that extreme, and that is the fuzz.

5 Fuzz

For our fuzz pedal, we are going to read the .wav file as we usually do. Then, we will normalize the input audio to put it in a range of $[-1, 1]$. Then we raise the input to the power of a distortion factor that controls the amount of distortion, while preserving the sign of the original signal.



The function $y = sin(x)^D$, where the sin wave represents our input wave and D represents the our distortion factor, which is 8.2 in this case.

This allows us to create a non-linear fuzz effect with extreme clipping. While it may create a similar, albeit more extreme, result to distortion, the method used to achieve this result is quite distinct.

Works Cited

"2. Frequency Domain — PySDR: A Guide to SDR and DSP Using Python." Pysdr.org, pysdr.org/content/frequency_domain.html.

3Blue1Brown. "But What Is the Fourier Transform? A Visual Introduction." YouTube, 26 Jan. 2018, www.youtube.com/watch?v=spUNpyF58BY.

"Audio and Digital Signal Processing(DSP) in Python." Python for Engineers, 17 Aug. 2021, new.pythonforengineers.com/blog/audio-and-digital-signal-processingdsp-in-python/. Accessed 29 May 2024.

Baltic Lab. "Overdrive and Distortion Effect Digital Signal Processing (DSP) Algorithms on the Arduino GIGA R1." YouTube, 20 Aug. 2023, youtu.be/LCT-7UGgMzk?si=D7zPqIRK_EsiSJwR. Accessed 29 May 2024.

"Desmos — Beautiful, Free Math." Desmos.com, 2019, desmos.com.