Brian M. Roach
Adam Eisenstein

Boston University
CS591: Computational Audio
Professor Wayne Snyder
Spring 2016

Final Project
Audio Effects Suite

Introduction

To all whose curiosity is provoked by our work and digital audio effects simulation:

We set out to design a suite of audio effects simulators in Python 3.5 which would read in a .wav file, apply the desired effect, and write out the newly created .wav file. For most of the effects, we designed the functions with variable parameters, but set the default variables to values we found most effective through testing on an array of both polyphonic and monophonic signals, such as vocals, string instruments, percussion instruments, simulated sounds, and symphonic pieces. While we did a great deal of research on how these effects have been implemented in the past, we ultimately took a more creative route, and derived intuitive implementations which deployed many of the concepts we learned in CS591: Computational Audio. We had a great time designing these effects and testing code, and hope you enjoy reviewing our code and listening to our created files.

Best,
Brian Roach & Adam Eisenstein

Implementations

Reverb

To implement a simple reverb simulator, we first decided to allow for two parameters: predelay and delay, where predelay is how many seconds would pass before one heard the initial sample repeat, and delay was for how many seconds one would hear the sample repeating for after the predelay period had elapsed. Rather than having each repeated sample in the delay period be of the same amplitude, we implemented a generic exponential decay function to make each sample decay exponentially throughout the course of its decay period. We then lowered the amplitude of the output signal to reach the input signal's average amplitude, converted our output signal back to ints for .wav writing compatibility, and wrote out the file.

Convolution Reverb

Convolution reverb simulates the acoustic nature of an environment, then applies that reverb pattern to a clean, secondary signal to make an output signal sound though as if the secondary signal had been recorded in the environment which the model signal had been recorded in. To get a clear model of the environment, we implemented a function that generated the convolution kernel of the environment, and it did this by analyzing a signal with a loud, quick sound (we used a clap). First, we designed the kernel generator to mark the max amplitude, then mark the sample index after the max where the amplitude dipped below a threshold(variable driven), and then slice the signal to include just those samples in the max to post threshold range. Then, we normalized the samples in that range to start at 1.0 and decay onwards. With this resulting array of floats, we were then prepared to apply reverb similarly to our standard reverb, but rather than using generic exponential decay, we used our model decay to shape the reverb for each sample. The remainder of the function was similar to that of our reverb simulator, where we corrected for any potential out of bound values and ensured correct pattern matching.

Distortion

For distortion, we took an approach often referred to as "soft clipping". In this approach, we multiplied an entire signal by a constant > 1.0 (variable driven), until a certain percent of the signal (variable driven), was clipping (where the amplitude exceeds the fixed range, in this case [-32768, 32767], marked the starting and ending indices of

each clipping event, and then rounded those (the then flat at the top (positive) or bottom(negative)) signals. We did this by setting the middle index in the event to the maximum amplitude allowed - 1, and then set each cascading sample to be the previous * a variable < 1.0 (previous meaning closer to the middle, hence working outwards in both directions from the peak). Then, we applied the standard range and type checks before exporting to a .wav file.

Phaser

For the phaser simulator, due to mathematical complexity, we relied more on previously done research and libraries rather than designing our own novel solution. To start, we made two half-amplitude clones of the input signal. The first of which will remain unaltered. The second of which we apply the Hilbert transformation to, and then we recombine the two initially split signals, do the usual checks, adjust for timing, and export to a .wav file.

Flangers (Constant & Sinusoidal)

For the flanger (constant implementation), similarly to phaser, we split the input signal into two half-amplitude clones. From here, we recombined the signals with one signal at a delay (variable driven, default 20ms). Afterwards, standard timing, type, and range checks were implemented. For the sinusoidal approach (non-conventional method, for academic curiosity) we did something similar, but whereas in constant implementation, outputSignal[i] = halfSignal1[i] + halfSignal2[i-20ms], in the sinusoidal implementation, outputSignal[i] = halfSignal1[i] + halfSignal2[math.sin(i)]. In theory, we would have one half running in original time, and one half speeding up and slowing down. However, our results did not yield this, and we'll discuss that in the conclusion section.

Utilities

The majority of the utilities we used were borrowed from Wayne Snyder's teaching code directory - including functions to read and write data from .wav files. We implemented some auxiliary functions, however, to assist with code condensing/repetition removal and simplification.

Conclusions

Reverb

Although we made every attempt we could think of to optimize the code, the runtime was still very long. Possibly it's simply the nature of directly interacting with every sample thousands of times, multiple times, that made the process take a substantially long period of time, and maybe we could have implemented more advanced solutions, but in reflection, our intuitive approach gave us a more concrete understanding of exactly what our code was doing, rather than making a plethora of advanced library calls where we were none the wiser as to what was going on 'under-the-hood'.

Convolution Reverb

We initially took a much more complicated approach, attempting to utilize a variety of libraries and advanced formula found on Wikipedia and StackOverflow entries, but we reverted to our intuitive design approach, where we attempted to implement the algorithm exactly as we understood the phenomenon to occur. Similarly to standard reverb, we felt this approach was easier to debug, understand, and convey to others.

Distortion

Distortion was fairly straightforward (as straightforward as it tends to go with standard amounts of debugging, testing, and waiting for files to be written). We had a solid idea going in with some notes, wrote the code, and were relatively pleased with the outputs. We did, however, find that signals with more background noise were less affected by the function, whereas cleaner simples became practically incompressible under certain testing circumstances, possibly because our averaging was thrown off by background noise.

Phaser

Due to the greater complexity of the phaser, we utilized scipy.signal.Hilbert to perform the Hilbert transformation on one half amplitude clone, but other than that step, the implementation was fairly straightforward. At points we had some issues working between numpy types and Python native types, but some quick research simplified conversions and our workflow

Flangers (Constant & Sinusoidal)

Going in, we knew the (constant) flanger would sound similar to the phaser, but a bit to our dismay, it sounded nearly identical (all of that work for the same (nearly) product -

damn!) but we still learned from the experience about how flangers are implemented in general. For the sine flanger, rather than just hearing a combination of two half amplitude clones, where one was running in constant time and one was running at variable time, we ended up producing a signal where there was a buzz at a certain frequency, and then the original half amplitude clone running in original time. Given more time we probably could have found a different approach, but given that it was solely based on curiosity, we figured we'd throw it in just to give anyone else who was curious an idea on the approach we took.

Overall

For several of the functions, we could not locate the a bug that made the output signal twice as long as it was supposed to be - we implemented assert statements to ensure the data going in and out of functions was of the proper length and type, yet could not figure out why the timing had been changed. To remedy this problem, we used a vocoder to change the timing before the data was to be written, however, we could not find a perfect combination of timing and pitch alteration to make all of the output signals as authentic sounding as we would have liked. Despite this, we still gained a deep understanding of the workings of the effects simulators we built, and enjoyed the process.