Dattorro's Reverb: A MATLAB Implementation

By Fred Kelly
MUMT-618, Fall 2024

Objectives and Motivation

The objective of this project was to write a MATLAB implementation of Jon Dattorro's plate reverb algorithm, as presented in part 1 of his 1997 AES paper "Effect Design" (1). Implementations for this particular algorithm have already been published in numerous programming languages (including MATLAB) (2) (3). Nonetheless, I felt this was an appropriate choice for my final project because I enjoyed our previous reverb assignments, and having never used MATLAB before this course I wanted to strengthen my skills on a more ambitious algorithm.

One technique that I found intriguing from James Moorer's "About this Reverberation Business" (4) was to compute early reflections based on an actual space's dimensions and listener position, and then using a less computationally intensive algorithm to generate the longer reverb tail. Since plate reverbs by their nature don't create early reflections, my original plan was to extend Dattorro's algorithm by calculating early reflections using the phantom source technique described by Moorer. As the project went on it became apparent that implementing Dattorro's reverb alone would be challenging enough, making additional features outside the scope of this project.

Project Design

My MATLAB script consists of the following stages: user parameter definition, file loading, object initialization, pre-processing, sample-by-sample tank calculation, and output taps. I've included a number of interesting sound files to process, as well as a folder of impulse response graphs showing the effect of adjusting different parameters compared to the default settings defined by Dattorro.

The script begins with a number of reverb parameter definitions, set to the defaults as per Dattorro's paper. The desired file to add reverb to is then read from the project folder, reading sample data into a buffer as well as the sample rate of the file.

One component of the algorithm that required some extra thought was the reverb's sample rate. Dattorro defines a sample rate of 29,761 Hz, which doesn't correspond to the most common sample rates for audio files (44.1 kHz and 48 kHz). One approach would be to convert the input file's sample rate to 29,761 Hz, and then proceed with the algorithm as written. The biggest impact of this conversion would be a loss of the highest frequencies since Nyquist would also be lowered, as well as a higher noise floor due to interpolation. High frequency loss isn't a major issue for the wet reverb signal since the reverb tail itself won't usually contain a lot of important information close to Nyquist. Furthermore, the reverb signal is usually mixed fairly low compared

to the dry signal, making a reverb signal with a higher noise floor more acceptable. That said, I opted against this approach because it meant that the dry signal would also have to be downsampled, so even if the reverb was very low, the dry signal would be noisier and lose high frequency information. Degrading the dry signal did not feel acceptable.

Instead, I chose to compute the reverb at the native sample rate of the input file. At this new higher sample rate, implementing the delay line lengths exactly as defined by the paper would preserve the exact relationship between delay lines. Although the underlying delay lines would be the correct length in terms of samples, the reverb would decay faster than intended at the same settings since each sample corresponds to a shorter unit of time. Therefore, I scaled all the delay line lengths by the ratio of the files sample rate to the original sample rate and used the ceiling of that value. This will increase the memory requirements of the algorithm but the overall decay time of the reverb will be the same as if the reverb were running at the original sample rate of 29,761 Hz. One downside of this ceiling operation is that it slightly alters the ratio of delay times between two delay lines, impacting the reverb's exact echo pattern and the comb filtering that occurs between summed output taps.

With all the delay line lengths calculated, class objects representing modulated and unmodulated allpass filters as well as simple delay lines are initialized. Then, the input signal is processed as a vector to convert to mono, apply any pre-delay, low-pass the signal, and decorrelate the input with 4 unmodulated allpass filters. This preprocessing stage can use vector computation and act on the whole input signal since all of these stages occur in series. This should run faster than computing one sample at a time in the main loop. This processing stage uses MATLAB's built-in filter() function for the pre-delay, low-pass, and all-passes.

To implement what Dattorro calls the "tank," I defined classes for modulated allpass filters, unmodulated allpass filters, and standard delay lines. This was primarily to abstract away the bookkeeping of an unwieldy number of buffers, read and write pointers, and temporary variables. This provided the additional benefit of allowing for arbitrary tap outputs that read from a class's internal buffer, since the final output of the reverb is a sum of taps from various locations in the tank.

After summing the necessary taps (Table 2 of Dattorro's paper), the reverb signal is mixed with the dry signal based on the "mix" parameter, and finally written to a new audio file.

Challenges

I first attempted to implement this reverb algorithm in one single script file, similar to how I completed the homework assignments. But with an algorithm containing so many interconnected processing blocks, the main loop quickly became a mess of repeated, similar-looking calculations, making it hard to decipher the overall structure of the program and how it related to the block diagram. Keeping track of all the required buffers, pointers, and temporary values introduced many potential points of failure, where a mis-numbered variable name could cause the reverb to blow up. As I found various mistakes in my implementation for

the allpass filter, I had to edit many equivalent lines at once, and it became clear I needed to abstract these details. After implementing each processing block as a MATLAB class, I stopped having to manually update each instance of a filter since they all referred to the same functions, making it faster to fix mistakes. After taking the time to write these classes, the main loop shows a much more obvious resemblance to the block diagram in the paper.

Most importantly, it also became simpler to check my work for correctness by comparing the output of my generalized allpass filter class to an instance of the built in filter() functions in MATLAB. The file “FK_Ap_Verification.m” shows this process. To do so, I passed the same input through both implementations, using equivalent delay lengths and gain scalars. Then I wrote each result to a column in a table, and a computed third column where I subtract one from the other. I was expecting this column to consist only of zeros, assuming the two vectors would be identical. However in reality, many elements of this column are zero as expected, but many are nonzero. At first I thought this meant my class was incorrect, but upon inspection I concluded the differences were negligible. For example, index 63860 of the “compare” table shows the filter() object computed a value of 3.852287879614781e-24, my class computed a value of 3.852287879614739e-24, with the difference amounting to 4.261167021730792e-38. I presume this small discrepancy is due to floating point or rounding error, or perhaps an internal optimization of the built-in filter() function. Although I didn’t directly compare my overall reverb algorithm with another implementation of Dattorro’s reverb sample by sample, I checked my algorithm by ear against an online implementation in Web Audio(5). I downloaded the dry audio files from the website’s source code and processed them through my MATLAB script with the same parameters. I couldn’t distinguish the two processed sounds by ear.

Another challenging, and sometimes frustrating, part of this project was interpreting the notation from the paper. I haven’t meaningfully encountered DSP literature until taking this course, so I didn’t yet have the intuition to recognize the diagrams for common operations like allpass filters. Working purely visually from the diagram, I felt that the signal flow presented the feedback components as occurring simultaneously. This made translating the diagram into sequential time steps ambiguous - should the multiplied argument of the sum be leftover from the previous iteration? When should I update the delay line? Suppose I compute the first sum using the output from the delay line from the previous iteration. I would write to the delay line, which would push a new sample through the end of the delay line. But then, wouldn’t the sample passed to the second sum be a different sample than the one passed to the first sum, even though they’re the same node? My thinking would go around in circles. At first, I often had the multiplied argument of the sums being one time-step off, or mistakenly computed more than one time step per iteration of the main loop. Any of these errors would cause the reverb to become unstable/blow up. Things finally started to make sense after reading Beltran, Holzem, and Gogu’s “Matlab Implementation of Reverberation Algorithms” (6), which presented many aspects of Dattorro’s algorithm in MATLAB-specific syntax. Most helpful was the modified diagram in Figure 2, which shows an equivalent allpass circuit to the one in Dattorro’s paper, but with the elements re-ordered. This diagram clarified the correct order of each part of the computation. Checking this for correctness became even easier after writing classes for my allpass filters since I could compare directly to the built-in filter() function.

Computing the final output from intermediate delay line taps was also tricky to interpret from Dattorro's notation. For example, one tap is written as "node48_54[266]". I understood this to mean that this output sample should be taken from some desired node and delayed 266 samples. In the block diagram, the output from the internal delay line of an allpass filter is labeled "48", but the signal passes through a sum and another delay line before reaching the point labeled "54." At first glance, it seemed like there were 3 different locations where this tap could reasonably be taken: after the internal delay line of the allpass filter, after the sum, or after the second delay line. After thinking about it, it seemed most logical to tap after the sum, because I could achieve the desired 266 sample delay by reading from the appropriate place in the second delay line. I'm not completely certain this is the correct implementation - if the desired point is actually the output of the first delay line before the sum, then I should define an entirely new delay line for these values rather than reading from the second delay line. In the event that I'm wrong, my implementation is more efficient since it requires less memory by eliminating the need for an extra delay line per tap. Regardless, my implementation is perceptually the same as the WebAudio implementation I compared with.

Sound Examples

Labeled sound examples are included in the project files. The drum loop is a file I originally composed for a sample pack, while the Brian Wilson and Michael Jackson isolated vocal stems were sourced from YouTube (7) (8).

Biggest Takeaways

This project often required me to make educated guesses on implementation details even after referring to other resources and examples. I still don't feel certain that I'm perfectly implementing the exact algorithm from Dattorro's paper. In most software engineering contexts, this kind of imprecision is a bad thing, and throughout this process I felt a real discomfort around not knowing the "right answer." When my project reached the point where it was stable and finally making reverb sounds, this discomfort eased dramatically. I imagined what the original design process of this algorithm might have looked like to Dattorro. Although it sounds like a plate reverb, I don't classify this algorithm as a physical model or simulation - rather an extremely clever, convincing, and efficient approximation. Dattorro's reverb doesn't directly account for the size and shape of the plate, the material properties of the metal, the location of the driver and pickups, or other physical characteristics. In other words, it's not a precise recreation of a real plate. Instead, he must have experimented with different signal flows, tap points, delay line lengths, and filtering stages, evaluating by ear until the result sounded good. As my project approached completion, I began to see my own imprecision as secondary to the perceptual characteristics of my implementation, because the educated guesses I made were also informed and evaluated by listening. When the reverb sounded bad, I searched for areas where I may have misinterpreted the paper's notation. After lots of trial and error, soon the reverb sounded good, and eventually indistinguishable from another implementation I found online.

Even more broadly, completing this project exposed me to DSP techniques that I aim to use in developing other audio effects, with the goal of making real-time plugins for use in recording software. I'm better at digesting unfamiliar notation and interpreting signal flow diagrams, which will make it easier to implement ideas from other papers and eventually build the intuition needed to design original effects. I feel that this project was certainly worth the effort, and has inspired me to dive deeper into physical modeling and DSP.

Works Cited

(1) Dattorro, J. (1997). Effect design. Part 1: Reverberator and other filters. Journal of the Audio Engineering Society, 45(9), 660–684. https://www.aes.org/e-lib/browse.cfm?elib=10160

(2) Adsp-21369-Reverb. (n.d.). GitHub - adsp-21369-reverb/MATLAB-Reverb: Jon Dattorro Plate Reverb implemented in MATLAB. GitHub. https://github.com/adsp-21369-reverb/MATLAB-Reverb

(3) Mjarmy. (n.d.). dsp-lib/PlateReverb.hpp at main · mjarmy/dsp-lib. GitHub. https://github.com/mjarmy/dsp-lib/blob/main/PlateReverb.hpp

(4) Moorer, J. A. (1979). About this reverberation business. Computer Music Journal, 3(2), 13. https://doi.org/10.2307/3680280

(5) WebAudio Dattorro. (n.d.). https://khoin.github.io/DattorroReverbNode/

(6) Beltran, J. R., & Beltran, F. A. (2002). Matlab implementation of reverberation Algorithms. Journal of New Music Research, 31(2), 153–161. https://doi.org/10.1076/jnmr.31.2.153.8096

(7) Brian Wilson. (2020, October 13). 1966: Brian Wilson vocal insert overdubs for "Don't Talk (Put Your Head On My Shoulder)." [Video]. YouTube. https://www.youtube.com/watch?v=iOFBd3l2UKE

(8) Sefton Productions. (2020, May 20). Michael Jackson | Beat It | Lead Acapella | AUDIO ONLY [Video]. YouTube. https://www.youtube.com/watch?v=8Q20CkstUo4