Automatically Identifying and Fixing Single Channel Audio Defects in Stereo Audio

Koen Putman

January 23, 2018

Abstract

We attempted to automatically detect and filter audio defects in days worth of stereo recordings. These defects include noise, distortions, broken channels, channel imbalance, and clipping. We will describe how we detect these issues and how we attempt to filter them. We implemented software that could run in real time and attempts to deal with these issues with varying degrees of effectiveness. While this work can serve as a nice base to improve, it is far from production ready and we spend a lot of time describing possible improvements.

1 Introduction

Imagine having days worth of stereo recordings that all suffer from issues like noise, but mostly in one channel at a time. You can go through and manually try to get rid of it, but what if you could automate the detection and fixing of these problems. For this project we will try and implement some software that will automatically fix audio defects without supervision while not needlessly affecting audio without issues.
We realise this is not a common problem and do not expect anyone else to find much use for this, but we will take you along for the ride of trying to wrap our heads around this problem and a solution. Our test samples are based on days upon days of recordings from an internet livestream of a club, so they are one large collection of copyrighted content and can not be shared. Given that our approaches were mostly determined by experimenting on problematic samples from this dataset the presented methods are not guaranteed to be effective on anything else, let alone the dataset itself. As this is a very specific problem there is not much in terms of literature on this subject available, so we will mostly be describing our own findings.
We will start with a short introduction of the audio defects in Section 2, followed by our ways of detecting and filtering them in Section 3. We actually implemented these so in Section 4 we describe our implementation and the issues we faced while implementing. Our experimental results and shortcomings of the filters will de described in Section 5 and we end on a conclusion and a short list of possible improvements in Section 6.

2 Audio problems

This section shortly describes some of the issues we want to deal with. Given that describing audio in text is not our strong suit it will be somewhat vague. The project website will have some samples to make the issues clear.

Noise

While the recordings have some overall noise, there were several instances of very apparent white noise in a single channel while the other channel was fine. This was probably introduced by the mixer or something in the connection between the audio source and the streaming PC. While it is not very audible when the music is loud, it is definitely annoying when the amplitudes are lower. Even when it is virtually inaudible due to the relatively low amplitude, it is noticeable when you suddenly stop playing any music and take off your headset. Your brain will probably be generating similar noise in your other ear to compensate.

Broken channel

Broken audio in this case refers to one channel suddenly cutting out or becoming heavily distorted. This happens sometimes due to glitches in the streaming setup or the turntables. This is generally fixed by duplicating the channel without issues, but given that it only happens sometimes it is very annoying to locate and deal with.

Unbalanced audio

This is a more common problem and happens on virtually every recording, though usually not to a very noticeable degree. It is very annoying to listen to when the difference gets bigger though.

Clipping

When the channels are unbalanced issues like clipping can pop up. If the amplitude reaches the maximum some information is lost. This can give problems after normalisation where the channel with clipping will be missing something even if the general levels are comparable.

3 Detection and filtering

Here we will attempt to describe the way we dealt with the various issues we mentioned. These methods have changed a lot and are not even effective in many cases, but they should serve as a starting point for how to filter for these issues. Our approach uses Hann windows with 50% overlap and we obtain the frequency spectrum using a Fast Fourier Transform (FFT). We use a real optimised version of the FFT so we get $framesize/2 + 1$ bins from our transform.

3.1 General metrics

Given that we need some way of detecting these issues, we need some numbers. We generally keep a fixed size history of data for previous frames to base our filtering on. For the time domain we keep an average absolute amplitude and peak amplitude. For the frequency domain we keep a history of the absolute values for every bin, as well as peaks and average level over all frequency bins. These values are then used to determine if we want to filter and how much.

3.2 Noise

This was the first thing we attempted to deal with, but also turned out to be the hardest to deal with overall. The current approach is based on the average level for that frequency per channel over the entire history as well as a threshold for if the current bin needs to be altered.
Our way of detecting this is very flawed, but it is very difficult to effectively filter noise without some form of profile. Our method involves looking at the difference in level for this bin. This is, of course, not the best way to deal with this, but it is a fast way. The way we decided to tackle the noise problem is by applying some negative gain in the frequency spectrum to all channels below a certain level and smoothing these gains to reduce the side effects on neighbouring channels.
The main approach is to compute the average difference between the bins in the entire history in dB. If this is bigger than 1 dB we will reduce the level of the louder channel by this amount. We also use the difference to determine how much we reduce the level of this channel if we detect that it is below the threshold we set. Currently we choose -9dB as a reduction amount and make it vary between -0dB and -18dB based on the average energy difference between the channels. This is added to the gain reduction. Once all bins have been checked we smooth the gains. This is done by taking their natural logarithm and averaging the closest 3 in both directions and the current one. Once this is done we get the actual gain multiplier back by raising e to our smoothed power. The reason we use a logarithm here is because of the way audio works and how we perceive it. Both channels are then multiplied by these gains.
This method will suppress more than just the noise so it is certainly flawed. It will also let some noise through when it is heavy. These things are hard to avoid though when you are generating some form of noise profile on the fly without any knowledge.

3.3 Broken channel

The interesting part of dealing with this is the detection, as the filter basically just copies either the time or frequency data for the normal channel. We use detection in both the time domain and frequency domain. The time domain detection is based on the amount of difference between the average amplitude of the channels and the amount of times the signal is zero. For the frequency

domain it is based on the average frequency.
The currently defined ratios are one channel's average being three times as high as the other on average in either frequency or time, or the amount of times the signal is zero being twice as big as the other channel. These tend to detect a lot of the broken audio, but are not as effective on smaller distortions. Not every consecutive frame is going to pass these thresholds when during the periods with broken audio and not every frame that passes them is actually broken. We employ some counters for our approach with a maximum level. and a set increment and decrement. The current maximum is 512 and every broken frame increases the count by 16, while a normal frame reduces it by 1. If it reaches over a certain threshold, 32 in this case, it will start replacing a broken channel. This channel is not necessarily the right one to replace though, because broken audio shows up in several forms. So we keep a threshold for changing the channel we replace as well. For every bad frame that is the same as a previous one the counter is decreased to a low point of -20. If the broken frame is different than the last we increase the counter. If the counter reaches 3 the channel switches and it is reset to 0. This way larger stretches of broken audio will still replace the correct channel.
As we mentioned before this method is not always effective at detecting distortions, and even if it does they have to happen very often in short succession to reach the threshold. The switch from stereo to mono and back is also a bit obvious, but it is not as bad as the broken audio is.

3.4 Normalisation

We thought of multiple approaches, but the first one we tried seemed to work rather well, so we stuck with it. We normalise in the time domain based on the difference in average amplitude, but only when it is within reasonable bounds. This seems to take care of most of the issues. The other approach would be to do it based on the energy in the frequency spectrum, which would actually help with some of our spectrum based filters, but not with others. We have two moments at which we can normalise, one is before the FFT and one is after the iFFT and just before writing to a file. These are generally good enough to take care of a normal channel imbalance.

3.5 Clipping

Detecting clipping is pretty easy, but trying to regain peaks is not a trivial task and something that I could not figure out within the scope of this project. Detecting clipping is basically just testing if the signal reaches full amplitude. My ideas for filtering were mainly based on the levels in the frequency spectrum and multiplying some of the higher levels of the normalised signal to try and regain some of the lost peaks in frequencies. This approach is flawed and does not work well in its current state, but there is probably a decent way to improve it.

4 Implementation

In order to get this implemented efficiently so that it could be run in a real-time setting we implemented it in C. The main reasoning for this is that it could also be used on streams as they are running so the problems do not have to be fixed afterwards. This is also the reason why the proposed filters are all based on the current frame and do not modify older frames based on new information. This does result in some issues getting through, but it will run with a very minimal delay of around 1 frame length. Given that we chose 1024 samples/frame and run on 44.1 kHz signals we are talking about under 25ms delay here. It's possible to reduce the frame size even further, but that reduces the frequency bin count so it comes with some trade-offs. We currently keep 512 frames of historical data, so all the filtering is based on the past approximately 10 seconds.
For audio file reading we used libsndfile, which will not work for our final goal of running in real-time on running streams since it has some issues with the way audio is read/written from pipes, but we did not have time to implement our own replacement. So for now we can only run on existing audio files.
For the FFT we use Kiss FFT a simple drop-in FFT with a real-optimised variation that is reasonably fast and easy to integrate. This could of course be improved, but it is not the limiting factor.
The implementation is fast enough to run faster that real-time even before real optimisations, so that will not be an issue. The implementation is kind of library-like, where the main function just calls several library functions in succession, but the structure is not entirely done yet.
There is a set order for filters in every iteration, which goes through all the steps for a frame made up of old an new samples in a 1:1 ratio. It starts by reading the samples from the input file and places them in the second half of the frame while the first half is the samples from the previous frame. It detects clipping and applies the window. After this we analyse the time domain and generate some statistics. If the option to fix broken channels is enabled this is where the time domain analysis would happen, so the channel will be replaced right off the bat. After this the signal will be normalised if we so desire. This takes care of the first time domain step, so we can FFT and obtain the spectrum. Right after the transform we generate statistics about the current spectrum. Then we detect and replace broken channels again, but based on the frequency spectrum. If there was clipping and the option was enabled this is when the clipping fix would be attempted. Finally before transforming back the de-noising happens. After the transform back we generate more statistics and apply the post normalisation if required. Then the final step is adding the old samples from this frame to the ones from the previous frame and write them to the output file. The newer samples are stored in a buffer for the next loop iteration so they can be added together to deal with the Hann window.
This order is not perfect. De-noising becomes a lot harder after you normalised in the time domain, so especially for that case we could probably do some frequency based normalisation instead, or transform multiple times to avoid influencing the metrics, but that comes with a performance penalty and impli-

cations for the metrics when a signal does not have noise but did have to be normalised. Working these out is still an ongoing problem and will require more experimentation and consideration. Our implementation also made it more difficult to do mono tricks when replacing the channel, like delaying one channel for a while. Given our very fixed amount of data that needed to be written it was kind of difficult to balance everything. We kept all the statistics private to the filters, so you can not access them normally outside of that. The main reason for this was to statically allocate everything and keep the constants in one place. Just like there is only a single call to malloc in the entire code because all the buffers are placed in a single contiguous memory space. In the current version of the code it is not even in an active codepath because we also allocate the large buffer statically. One of the initial ideas was that it would be nice if it could run on some small computer like a Raspberry Pi, but getting it fast enough to run all filters in real-time on that might require some extra optimisations.

5 Results and shortcomings

Results are subjective and far from perfect, but for the most part normalising is actually really effective. On all the heavily imbalanced samples it manages to get a decent balance back, but louder channel is still a bit lower after normalisation. This can probably be dealt with using some better balance of normalisation or normalising based on the frequency spectrum. Clipping fix is not really helping at this point it time, so we will not discuss that further. Fixing broken channels by replacing them is fairly effective, but suffers from the fact that it is always a few frames late and has some issues with falling below the threshold when the audio sounds decent for a while. It does work really well when one channel is properly broken though. One of the main problems is that the metrics used to detect it are very imperfect and sometimes classify normal audio as broken leading to channel replacements when not appropriate, like when one channel is intentionally left silent in some music as an effect. De-noising has become our worst enemy. It can be really effective, but also has an effect on all of the other audio. Sure the final results are decent in some cases and it can work pretty all right, but it is far from the easy to fix problem we expected it to be. The main reason for this is probably how we generate a makeshift noise profile from channel differences and then apply it to both channels to varying degrees in order to keep both of them noise free.
The biggest issue is that, while some filters like normalisation and clipping fix are fine to run on properly balanced audio without causing many distortions, the other filters tend to mess with the audio a lot more. So running these without noticeable side effects is probably never going to be possible. It should be possible to dynamically allow changing the filter configuration when running in real-time and suggest filters to the user though. Some examples of decently filtered audio will be on the project site.

6 Conclusion

We set out to create automatic detection and filtering software and we managed to create a decent base implementation, but getting actual good results will require more time spent on experimentation and research. Some of the results are very promising though, so they make it seem like a working solution for these problems is more than a pipe-dream.
As improvements to the implementation itself we would like to implement some form of lookahead to be able to catch some issues and correct them before writing to the output instead of only basing everything on the history. Other than the history there is also the issue with not being able to run with piped input/output with this implementation and the audio library it uses. A simple PCM in wav read/write replacement should not be too much work and will make it possible to chain this between a livestream and output application/encoder. Currently basically all configuration in terms of parameters is compile time because it is efficient at runtime, but we would like to expose some of these to runtime configuration so it can be easier to experiment.
In terms of the filters themselves we would want to experiment more with the order and maybe add more steps. There are also a lot of potential optimisations possible with SIMD for instance. In terms of filters we would like to improve all of them of course. It is a work in progress so there is always room for improvement. The main short term goals per filter would be interesting to list though.
De-noising will need some major logic changes to stop ruining proper audio. There have been previous instances during this project that were less destructive, but also worse at removing the noise. Getting average levels from several neighbouring bands instead of just the one would probably also get a bigger improvement, though it would incur an even bigger performance penalty.
Fixing broken channels would need some additional detection methods and parameter tweaking. It is not good enough at detecting slight distortions that still ruin the listening experience. It could also stand to be more aggressive in replacing consecutive frames since it does tend to switch back to stereo after short periods of not very broken audio. There is also the possibility of faking stereo effect on the audio by delaying it a bit. We already experimented with inverting the amplitudes for the other channel which had some effect, but that was not more pleasant to listen to. This filter in general is able to run on most clean audio without causing problems, but the occasional misstep where it replaces proper stereo with mono for very little reason can be annoying.
Normalisation will require some tweaking and maybe some work in the frequency domain so that it can be used in combination with other filters more easily and perhaps get some better results on some samples. Clipping fix will require a lot more work. One of the ideas to fix clipping is to try and recreate the shape of the non-clipping channel in the one that did, but that is probably going to be hard to balance.