API 2017/2018
Towards real time audio mosaicing

Christiaan Lamers [s0315435]

January 22, 2018

Introduction

This project is inspired by the Audio Mosaicing done by Driedger et al. [1]. Audio Mosaicing is the process of approximating a target sound sample using slices of a source sound. Driedger et al. demonstrated this by picking "Let it be" by the Beatles as a target sound and the sound of buzzing bees as the source sound. The result is bees buzzing the song "Let is be". The result can be heard on their website along with more examples [1].
The method of Driedger et al. is designed to work on prerecorded audio. For this project we wondered if it would be possible to alter the method in order to make it functional in a real time setting. This way the method could potentially be used as a live musical effect.
The goals of this project are to recreate the method of Driedger et al., to alter the method to make it suitable for a real time environment and to try to run the method in real time.

Let it Bee

Audio mosaicing is done by Driedger et al. by using the method depicted in figure 1. In this figure we can see a target sound, in this case "Let is be" by the Beatles and a source sound, in this case the buzzing bees. The source sound and target sound are transformed using a Fast Fourier Transform in order to produce matrices containing spectrograms. The goal is to learn an activation matrix for which the dot product of the source sound and the activation matrix is similar to the target sound. This dot product is the resulting audio mosaic. It is important to note that the source sound matrix and the target sound matrix contain only real values derived by taking the magnitude of their complex counterparts. The activation matrix consists of positive real values. The audio mosaic is made by using the complex values of the source sound together with the real values of the activation matrix. The resulting matrix is then converted back to audio using the inverse Fast Fourier Transform.

Classic method

The basic method to learn an activation matrix is by trying to minimize the Kullback-Leibler Divergence of the target sound and the mosaic. The formula for calculating the Kullback-Leibler Divergence is the following:

$$(V||WH) = \sum_{nm} \log \frac{V_{nm}}{(WH_{nm})} - V_{nm} + (WH)_{nm}$$

In this formula, $(V||WH)$ is the Leibler Divergence, V is the target sound, W is the source sound, H is the learned activation matrix and WH is the mosaic.
According to Lee and Seung [2] the Kullback-Leibler divergence is non-increasing under the following update rule:

$$H_{km}^{(l+1)} = H_{km}^{(l)} \frac{\sum_n W_{nk} V_{nm} / (WH^{(l)})_{nm}}{\sum_n W_{nk}}$$
$l \in [1 : L - 1]$

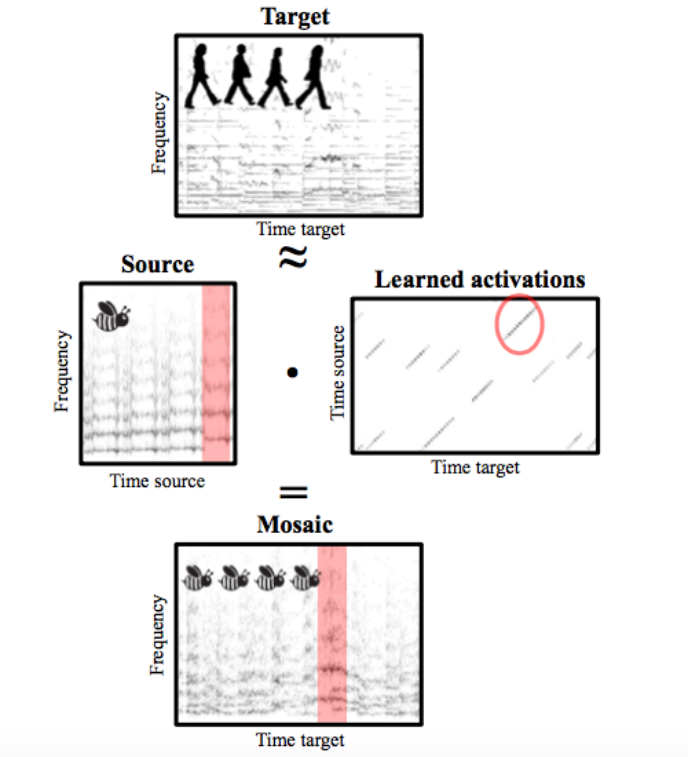

Figure 1: Let it Bee

Here H, W, V and WH are the same as in the previous formula. The variable L represents the number of total iterations and l holds the current iteration number.
Because the Kullback-Leibler divergence is non-increasing under this update rule, it can be used to minimize the Kullback-Leibler divergence, thereby learning an activation matrix H for which the mosaic WH is similar to the target sound V.

Extra update rules

The basic method of Driedger et al. produced undesirable results in the form of stuttering, phase cancellation and loss of temporal characteristics. In order to address this, they proposed extra update rules restricting repeated activation of the same audio frame, countering the stuttering. They also restricted the amount of frames being activated at the same time. They call this polyphony restriction and is counteracts the phase cancellation. The loss of temporal characteristics is countered by promoting diagonal activation patterns. This way continuous strips of audio of the source sound tend to be activated instead of single frames.
To avoid repeated activations, Driedger et al. present the following extra update rule:

$$R_{km}^{(l)} = \begin{cases} H_{km}^{(l)} & \text{if} H_{km}^{(l)} = \mu_{km}^{r,(l)} \\ H_{km}^{(l)}(1 - \frac{l+1}{L}) & \text{otherwise} \end{cases}$$

$l \in [1 : L - 1]$

$$\mu_{km}^{r,(l)} = \max(H_{k(m-r)}^{(l)}, ..., H_{k(m+r)}^{(l)})$$

In this formula $\mu_{km}^{r,(l)}$ is the largest entry in among entries in the horizontal neighborhood of the currently updated entry. The variable r determines the size of this neighborhood. The effect of this formula is that subsequent activation of the same frame multiple times will be suppressed.
The key to this formula is the $(1 - \frac{l+1}{L})$ factor. This factor gets closer to 0 as the number of performed iterations increases. In the last iteration this factor becomes 0, completely removing the updated entry if it is not the maximum in the neighborhood. The effect of this is that at the first iteration, the update rule is roughly the same as in the classic method, but during the application of iterations, the effect of the μ function will increase. In the final iteration, only one entry will remain in any neighborhood.

In order to restrict the number of simultaneously activated frames, a polyphony restricting update rule is defined:

$$P_{km}^{(l)} = \begin{cases} R_{km}^{(l)} & \text{if} k \in \Omega_m^{p,(l)} \\ R_{km}^{(l)}(1 - \frac{l+1}{L}) & \text{otherwise} \end{cases}$$

$l \in [1 : L - 1]$

$\Omega_m^{p,(l)}$ contains p highest values of m^{th} column of $R^{(l)}$

This formula works roughly the same as the previous formula, but it operates on a different axis. The $\Omega_m^{p,(l)}$ function searches for the p largest activations. Only these p activations are allowed in the final iterations, restricting the number of simultaneous activations to p.

In order to promote temporal characteristics, activations are spread out in a diagonal direction using the following formula:

$C_{km}^{(l)} = \sum_{i=-c}^{c} P_{(k+i)(m+i)}^{(l)}$

$l \in [1 : L - 1]$

The effect of this formula is that every entry will become the sum of the entries lying in it's diagonal that are less than or equal to c entries away.

Finally the whole update cycle is wrapped together by the update rule:

$$H_{km}^{(l+1)} = C_{km}^{(l)} \frac{\sum_n W_{nk} V_{nm} / (WC^{(l)})_{nm}}{\sum_n W_{nk}}$$

$l \in [1 : L - 1]$

This update rule is the same as the classic update rule with the only difference that this new update rule takes the output of the previous three new update rules.

In order for this method to work in a real time setting we recreated the method with the extra update rules from scratch in Python and modified it. In order for the method to except streams of audio, we decided to update the activation matrix using a sliding window. Also we changed the $(1 - \frac{l+1}{L})$ factor to $(1 - \frac{Hx-m}{Hx})$ for all update rules, where Hx is the horizontal coördinate in the sliding window.
On $t = 0$ we position the sliding window at the start of the audio stream. We then do one update cycle using all the extra update rules with $(1 - \frac{l+1}{L})$ replaced by $(1 - \frac{Hx-m}{Hx})$. The Hx has the effect that entries in the activation matrix near the output side of the sliding window will be most effected by the new update rule restrictions. After every update cycle, the sliding window is advanced one frame. Every entry, except the first few entries will receive as much iterations as the sliding window is wide. In between update cycles one column of the

Window size	Run time (seconds)	Cycles	Desired speedup	Delay (seconds)
65000	15.413	28	1.10	5.505
6500	293.111	370	20.94	7.943
650	36572.119	3802	2612.294	96.192

Table 1: fft window size vs. desired speedup on "sheep" source and "back in black clean" target. Time of mosaic: 14.0 seconds, sliding window width: 10

activation matrix will have finished it's training process. This way this column can be used together with the complex valued source sound matrix to make one new frame of the mosaic. This way every update cycle one output frame is created. If one update cycle can be performed in a time shorter that the time of one frame, this method can be ran in real time, with only the width of the sliding window determining the delay between input and output.

Results

We implemented our method and it produces audio mosaics. Unfortunately one update cycle takes up too much time to run it in real time. Our implementations also lacks good pitch replication due to the lack of a pitch shifting algorithm operating on the source sound. Driedger et al. pitch shifted the source sound multiple times in order for it to have enough frequency variation to allow for recognizable melodies and harmonies.
At first we recreated the classic method by Driedger et al. we then added the extra update rules and compared the results of both update rule sets. We found that the classic method was the best at matching the target sound, but that the texture of the source sound got lost. The method with the extra update rules was better at maintaining the texture of the source sound, but the output contained some harsh glitchy spikes. The results can be heard on our website [3].
In order to evaluate the speed of the algorithm, we measured the runtime of the algorithm in seconds and divided this by the length of the audio mosaic in seconds. We consider this to be the desired speedup the method needs to have in order for it to run in real time.
Table 1 shows the desired speedup for different FFT window sizes. Reducing the FFT window size will result in having more windows, thus increasing the runtime. So a smaller FFT window size will need a larger speedup in order for it to be ran in real time. Smaller FFT window sizes will yield better defined mosaics. The Delay in this table is the time between one target frame entering the procedure and the matching mosaic frame exiting the procedure. This is calculated by dividing the run time in seconds by the number of update cycles performed. Then this is multiplied by the width of the sliding window, in this case it is 10 frames wide. The audio of these experiments can be heard on our website [3].

Discussion

We showed that it is possible to adapt the method of Driedger et al. to a real time setting. The question remains if it can be truly ran in real time after optimizing the method. For now we can only speculate.
In case of a FFT window size of 6500, we have a delay of 7.943 seconds. For a real time feel, we need to have a delay of under 0.01 seconds. Therefore in this case we need a speedup of $7.943/0.01 = 794.3$.
In the optimistic case that we can get a speedup of 100 when implementing this method in C++ instead of Python and then multiplying this by a speedup of 30 when using a GPU, we get a combined speedup of 3000. This is enough speedup for the FFT window size of 6500 and more than enough to get the delay under the 0.01 seconds mark. It's also enough speedup for the FFT window size of 650, but in order to get the delay under 0.01 seconds, we would need a speedup of $96.192/0.01 = 9619.2$. But with a speedup of 3000 we can bring the delay down to $96.192/3000 = 0.03$, which makes it still useful.
In a pessimistic case, we get a speedup of only 10 when switching from Python to C++. A pessimistic speedup of 2.5 by using a GPU would yield a speedup of only 25. This is not enough to get the delay under 0.01 seconds, even for the 65000 FFT window size.
It might be possible to run this audio mosaicing method in real time, but we did not incorporated the extra computation time associated with larger pitch shifted source samples. We don't know for sure if it's feasible until we tried.

Future Work

Future Work includes porting the method to a compiled language like C++. We expect to gain a lot of speedup by utilizing the power of GPU's, so this is an interesting thing to try.
Another interesting thing to track down is what causes the glitches in our implementation fo the method of Driedger et al. with the extra update rules. It might be as simple as properly windowing the target and source sounds, but it might be a more fundamental problem, where the method concludes that these glitching sounds are the best match to the target sound.

Conclusion

First steps where made towards a real time implementation. We managed to recreate part of the method of Driedger et al., but our implementation produced undesired glitches in the output. We did not implement the pitch shifting of the source sound. We expect this should improve the output, but it will cause longer run times, since the source sample becomes longer.
The sliding window implementation produces desirable results. The method successfully matches parts or the source sound to the target sound. However

melodies and harmonies are not recognizable in the output, only rhythmic patterns are recognizable. This can be attributed to the lack of pitch variation in the source sound, because it is not preprocessed by a pitch shifting procedure. The sliding window implementation is too slow in order for it to work in a real time environment. It is expected that implementing the method in a compiled language (C++ for example) instead of an interpreted language (Python) should yield a better performance. In order for this method to run real time we expect it is necessary to utilize the power of a GPU. This way it might be possible to run the method in real time.

References

[1] Jonathan Driedger et al., Let It Bee - Towards NMF-Inspired Audio Mosaicing, 10-26-2015. URL: `https://www.audiolabs-erlangen.de/resources/MIR/2015-ISMIR-LetItBee`

[2] D. D. Lee and H. S. Seung. Algorithms for non- negative matrix factorization. In Proc. of the Neural In- formation Processing Systems (NIPS), pages 556-562, Denver, USA, 2000. URL: `https://papers.nips.cc/paper/1861-algorithms-for-non-negative-matrix-factorization.pdf`

[3] Christiaan Lamers, Towards real time audio mosaicing (API), Leiden University, Netherlands, 2018. URL: `http://chris.sorkel.nl/`