ARPEGGIATORUM: AN AUDIO CONTROLLABLE MIDI ARPEGGIATOR

Axel BERNDT (axel.berndt@uni-paderborn.de) [1] and
Davide Andrea MAURO (davide.mauro@uni-paderborn.de) (0000-0001-8437-4517) [1]

[1] *KreativInstitut.OWL*, **Paderborn University**, Detmold, Germany

ABSTRACT

An arpeggiator is a system that takes in input notes (or chords) and breaks them up into sequences, so-called arpeggios. While typical MIDI arpeggiators receive and produce MIDI messages *Arpeggiatorum* allows for an audio signal to be the input of the system, performing mono- or poly-phonic pitch estimation. *Arpeggiatorum* is also specifically designed to work with pipe organs. In this contribution we present the architecture of *Arpeggiatorum*, highlighting the design choices that lead to the development of specific features, and its strengths and current limitations. We will focus our attention on the possibility to use an audio input, for example a voice, or multiple voices, to control the system, and the new musical possibilities that emerge. We discuss the impact that of some of the most crucial parameters, namely accuracy and delay of the audio input have on the performances and suitability of the system.

1. INTRODUCTION

Modern pipe organs no longer have exclusively mechanical actions, but are more and more often equipped with electromechanical actions. This allows organ consoles to be placed more freely and be equipped with a MIDI interface to control keys, stops, swells etc. However, these organs offer barely more new features than the possibility to record and replay performances, define mixtures, transpose and route keyboards to certain works. As part of a larger activity we began to evaluate the musical and technical possibilities of modern organs that lie untapped. These unfold their full potential when a computer (or more generally speaking, a programmable MIDI device) is connected between the console and the organ. Thereby, the computer can act as a mapping device for input from the console (effectively an ordinary MIDI input device), translate, enrich or react on it. Other input devices can also be utilized to play the organ. The computer can even become an input device in itself.

One of the mechanisms that we have specifically developed to expand the playing possibilities is an arpeggiator, called *Arpeggiatorum*. It came as a surprise that, although it would have been technically possible for decades and well-proven in the world of synthesizers (see for example [1]), arpeggiators so far never made it onto the organ. In contrast to many other software and hardware arpeggiators available *Arpeggiatorum* is conceived specifically for organs. This is reflected by some of its unique features and design decisions discussed here. For a review of sequencers, a class of systems that include arpeggiators, see [2]. For a discussion on the possibilities of an Arpeggiator as a compositional tool, see [3].

The paper is structured as follows: in Section 2 we provide an overview of the system. In Sections 3 and 4 we present the implementation details, and discuss the current limitations. Finally, in Section 5 we highlight the future steps for *Arpeggiatorum* and possible directions for this research.

2. AN OVERVIEW OF *ARPEGGIATORUM*

Arpeggiatorum is a Java application that implements a MIDI software arpeggiator, and it can also be controlled with audio as its input. It is released Open Source (with the GPL-3.0 license) utilizing the following libraries: JSyn [4], Tarsos DSP [5], and (optional) PortAudio [6].

The main GUI for *Arpeggiatorum* is presented in Figure 1. Three different tabs provide access to settings for MIDI (e.g. Input and Output Channel), Audio (e.g. Device, Channel, Pitch Tracking algorithm) and Arpeggiator (e.g. Enrichment Pattern, Tempo).

One of the most common scenarios for the use of *Arpeggiatorum* involves a musician performing on a MIDI equipped organ while controlling the software. For this reason a special performance mode GUI has been proposed. All relevant information should be readily available and easy to reach in a single screen and a touch-compatible interface (see Figure 2).

Settings are automatically saved and loaded from a text file upon opening and they include the Sample Rate (default 44100), parameters for pitch detection algorithms, and the state of the GUI.

3. IMPLEMENTATION

An arpeggiator's basic functionality involves an input device providing pitches (in this case *noteOn* and *noteOff* messages from a MIDI device) and the arpeggiator logic (see Figure 3). Since MIDI input devices are not necessarily limited to classic keyboards and organ consoles we were also striving for alternative input modalities. Singing is particularly relevant for church music, thus we have also added an audio input to *Arpeggiatorum* that determines

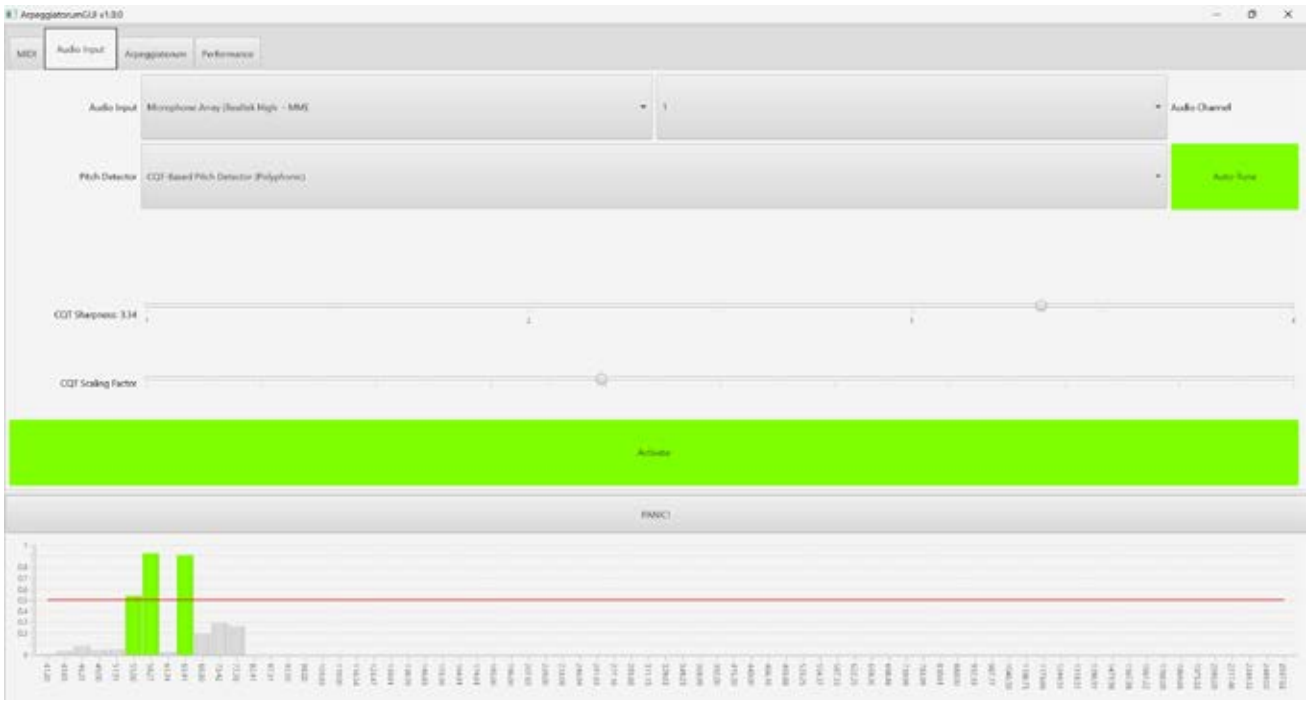

Figure 1. The GUI for *Arpeggiatorum*.

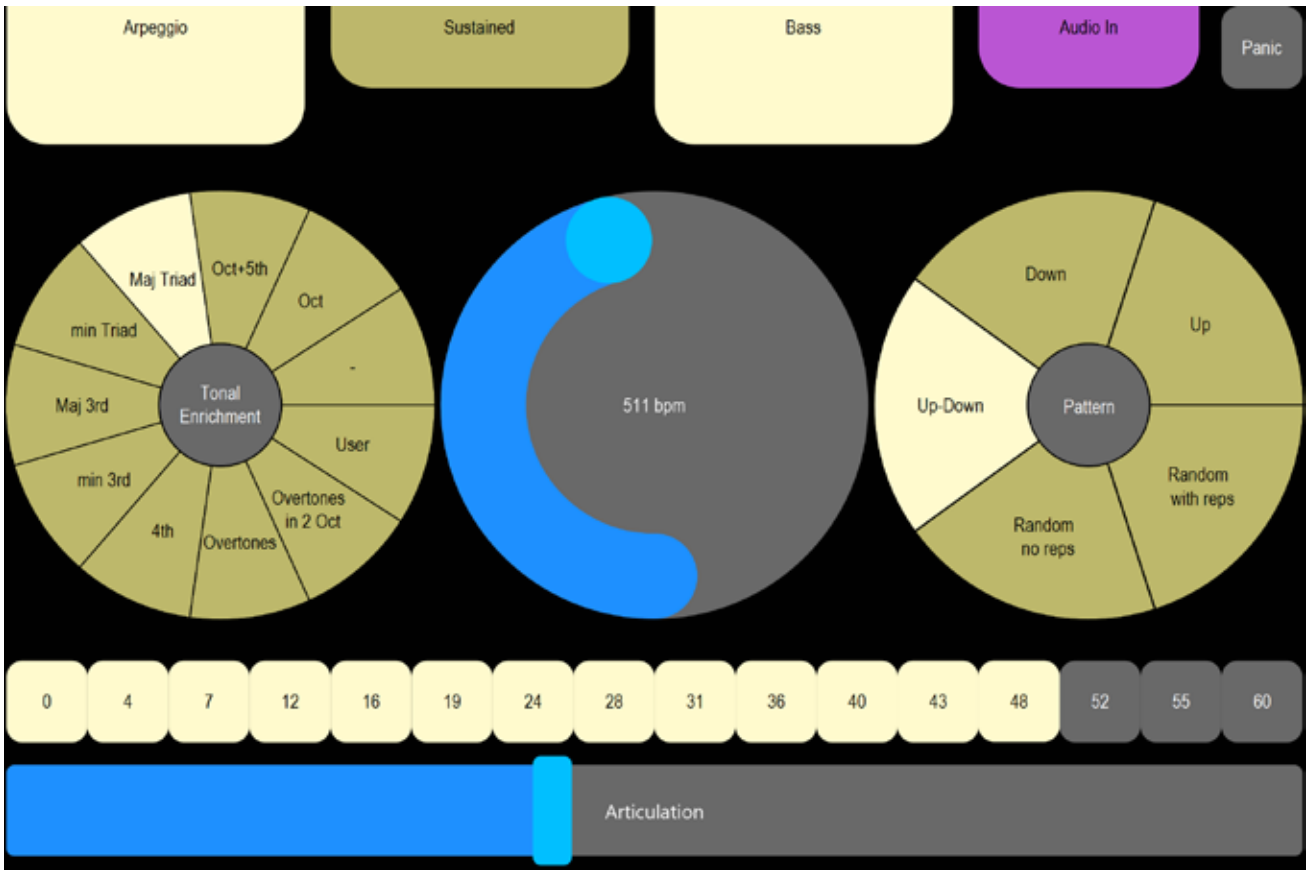

Figure 2. The GUI for Performance Mode.

sung pitches described in Section 3.1. The arpeggiator logic then enriches the pitch pool and plays them in sequences according to certain patterns, as detailed in Section 3.2.

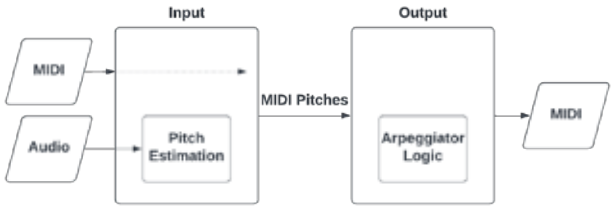

Figure 3. Functional diagram for *Arpeggiatorum*.

3.1 Audio Pitch Tracking

A novel feature of *Arpeggiatorum* is the possibility to use an audio input signal to control the arpeggiator. Various different algorithms are currently implemented, with the possibility to easily add other ones in the future thanks to the modular architecture of JSyn. In particular two classes of algorithms are currently implemented: monophonic and polyphonic pitch trackers. Monophonic pitch algorithms currently include: 1. Autocorrelation Function 2. Harmonic Product Spectrum and Cepstrum 3. FFT-YIN. Polyphonic pitch algorithms currently include: 1. Constant Q Transform.

Table 1 summarizes the lowest tracked frequency and delay introduced by each proposed algorithm.

Algorithm	Min Freq (Hz)	Delay (ms)	FFT Size
JSyn ACF	40	P + 8 samples	-
HPS/Cepstrum	86.13	12	512
FFT-YIN (Tarsos)	86.13	12	512
CQT (Tarsos)	41.21	743	32768
	82.41	372	16384
	164.82	186	8192
	329.64	93	4096
	659.28	46	2048
	1318.56	23	1024

Table 1. Lowest frequency, delay, and FFT size (where relevant) for various algorithms. P is the period of the target frequency. 8 samples is the buffer size of the internal scheduler of JSyn.

3.1.1 JSyn ACF (Monophonic)

JSyn includes a pitch detector that estimates the fundamental frequency of a monophonic signal. It utilizes the autocorrelation function and outputs the frequency as well as a confidence value. A new output is generated at every update of the internal engine (8 samples) so this value is further processed as follows:

3.1.2 Harmonic Product Spectrum (HPS) and Cepstrum (Monophonic)

JSyn also includes an implementation of the FFT. Two simple monophonic algorithms have been implemented based on the FFT. These methods work best for an input signal that is monophonic and has an harmonic pitched content.

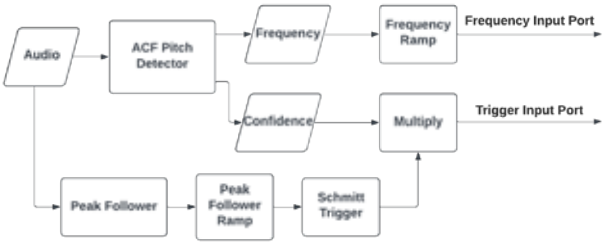

Figure 4. The workflow of the ACF pitch detector.

The Harmonic Product Spectrum method was presented by Noll in [7]. The main idea is that the spectrum consists of a series of peaks corresponding to the fundamental frequency and harmonic components positioned at integer multiples of the fundamental. So by downsampling and multiplying the spectrum several times the strongest harmonic peak should emerge. E.g. by downsampling the spectrum by a factor of two, the second peak in the original spectrum will line up with the fundamental.

The use of "cepstrums" was introduce by Noll in [8] in the context of vocal pitch detection. The FFT of the input signal is computed and then its spectrum, in log scale, is treated as a periodic signal itself. The periodicity of this new signal is linked to the spacing between the different peaks. In our implementation we compute the Cepstrum as:

$$C_p = |F\{log(|F\{f(t)\}|^2)\}|^2 \quad (1)$$

The same post-processing used for JSyn ACF is applied here.

Parameters include the chosen method, the FFT Size (default 1024 samples), which in turns determines the Lowest Frequency, and the Highest Frequency (default 1567.98 Hz MIDI Pitch 91 - G6).

3.1.3 FFT-YIN (Monophonic)

The implementation comes from the Tarsos DSP library. The library includes 6 pitch detection algorithms. It is a faster implementation of the YIN algorithm described in [9] and implemented by Matthias Mauch (Queen Mary University, London). The YIN algorithm is a modified version of the autocorrelation method.

A pitch is triggered when it exceeds a confidence threshold.

Parameters include the FFT Size (default 1024 samples) and the Confidence Threshold (default 0.98).

3.1.4 CQT (Mono- and Poly-phonic)

Utilizes the Tarsos DSP implementation of the Constant Q Transform, based on the work of Brown and Puckette [10] and Blankertz [11]. This transform is particularly well suited for musical applications given its frequency-varying resolution that more closely resembles our auditory system. By selecting its starting frequency to match one of the MIDI pitches, and 12 as the number of bins per octave each bin will correspond to a MIDI pitch. This results in

a significant reduction of the number of output bins when compared to FFT.

A notable drawback is the amount of delay introduced. Let us assume that for a Sampling Rate of 44100 we are interested in tracking the pitches starting from E1 (41.21 Hz). If we want to be able to discriminate E1 from F1 (43.65 Hz) a resolution of 2.44 Hz is required. This results in an FFT of at least 18074 samples. The current implementation search for the next power of 2 in order to optimize the computation. So a resulting FFT of length 32768 needs to be computed. Such an FFT introduces a delay of 743 ms, which might not be suitable for real-time audio applications.

An effective solution to mitigate this problem, at the expenses of a larger memory consumption, and setup time, is to compute multiple CQTs with different starting frequencies. By doing so each CQT will have a different delay, decreasing as the frequency increases, as the resolution needed is also decreasing. This means that higher frequencies can be detected faster than lower frequencies. Even for a single note this might mean that the first pitch triggered would be one octave higher than the actual note. In our application this does not cause particular problems as notes are added to the pool, possibly contributing to a new lowest note (sent to the bass channel), or enriching the arpeggio with new lower octaves.

Two other parameters are available for the CQT: Threshold and Spread. Threshold governs how the pre-computed kernels are truncated. A greater sparsity of the kernel matrices results in fewer operations to be computed. Finally, Spread effectively alters the resolution of the transform. A smaller Spread might result in a bigger FFT to be computed. For the previous explanation a value of 1.0 was used.

The current implementation utilizes 1 CQT per octave. Parameters include Lowest Frequency (default 41.205 Hz MIDI Pitch 28 - E1), Highest frequency (default 2637.02 Hz MIDI Pitch 100 - E7), Threshold (default 0.01), Spread (default 0.55, which is the smallest value that does not increase the FFT size compared to a Spread of 1.0).

The lower portion of the GUI depicts in real-time the CQT bins and their amplitude. Bins are colored green when they exceed the threshold (red line) set by the Threshold Slider, resulting in output pitches to be triggered.

A flag parameter called Auto-Tune engages a mode where if up to N contiguous bins (configuration parameter) are active only the one with the highest magnitude triggers an output. This can be useful if the audio input is not in tune with the CQT bins, or if the Spread is increased to reduce the delay resulting in a "blur" between consecutive bins. If more than N consecutive bins are active we consider it a "cluster" and all the pitches are activated.

The magnitude of a bin in relationship with the threshold is used to determine the Velocity of the *noteOn* MIDI message. Two parameters for minimum and maximum velocity are mapped with a linear interpolation to the threshold and a value corresponding to $2 * threshold$. The use of a Velocity based on the strength of the audio input can be further expanded by using the Aftertouch MIDI message (as opposed to a Channel Pressure message as it does not carry the Note information) to alter the volume of a note that is already playing, without triggering the notes in the arpeggio again.

3.2 Arpeggiator Logic

Arpeggiatorum's arpeggiation logic has been designed with flexibility in mind and the possibility to define and play rich arpeggios. Hence, it offers a number of parameters, all editable in the GUI (see Figure 1) and via MIDI control change messages.

A typical arpeggiator takes a number of pitches (MIDI *noteOn* messages, respectively) into a pool and plays them in repeating sequence as long as they are not removed from the pool (via corresponding *noteOff*). So does *Arpeggiatorum*. It collects the input pitches in a list sorted in ascending order, the *note pool*. A real-time thread (using the JSyn scheduler) traverses the note pool in a specified *pattern* (upward, downward, up and down, randomly with and without tone repetitions) and in a specified *tempo*, thus creating the arpeggiation sequence. The note pool is more than a sorted list of notes. The traversal pattern is part of its functionality, thus offering a handy `getNext()` method. Notes can be added to and removed from the pool while the sequence is generated. The traversal functionality ensures that the pattern stays intact.

The *articulation* of the notes can be set from staccato over tenuto up to an overlapping legato play. This is particularly useful when the actions of an organ are rather sluggish and would prevent very fast arpeggios from ever creating a sounding tone. Typical organ keyboards and pedals have a further limitation: They do not offer the full *range* of 128 MIDI pitches. This applies to most other instruments as well. Hence, we added a two-pole range slider to delimit the pitch values in the note pool that are included in the arpeggiation sequence.

So far, arpeggio sequences have only been created from manually triggered notes. A further functionality of the note pool is *tonal enrichment*, i.e. for each manually triggered note it can add up to 15 further notes. These are defined by interval relations with the triggered note. While the GUI of *Arpeggiatorum* allows users to specify their own series of intervals, it is also possible to choose from a list of presets such as "Octaves and Fifths", "Major Triad", "Minor Triad", "Fourths", and "Series of Overtones". A slider also allows to modulate on-the-fly how many of the enrichment notes should be added to the note pool, thus offering a further dimension for creative interaction during performances.

A final element of *Arpeggiatorum*'s logic is aimed directly at the organ with its multiple manuals and pedals. These are typically accessed via different MIDI channels. The arpeggios are sent to one of these. *Arpeggiatorum*, however, also allows sending the triggered notes (i.e. without arpeggiation) to another channel. And the lowest of the triggered notes is automatically transposed down (as bass note) and sent to a third channel, typically the pedals. Each

of the three channels (arpeggio channel, held note channel, and bass channel) can be muted. An organist who has used *Arpeggiatorum* in concerts emphasized this as a particularly useful feature. It allowed him to perform on a short 2-octaves MIDI keyboard in the middle of the nave (WiFi connection to the organ console) but effectively playing on two manuals and pedals.

4. AUDIO INPUT EVALUATION

For testing purposes the application is set to receive audio input from a virtual interface (VB-Cable [1]) connected to a DAW (Reaper [2]). The generated MIDI messages are sent to a software synthesizer (Java's Gervill), and the output is then sent to another virtual interface that is recorded in the DAW. All test input files have the same structure: 3 s of silence as the head and tail of the samples. All samples are normalized at -6dB FS. This allows us to conduct a preliminary evaluation measuring the accuracy of the CQT algorithm. Regarding delay the current setup (with virtual interfaces) introduces 40 msec to the measurements. For each test signal 5 measurements have been recorded.

The use of CQT for pitch-tracking is not in itself new, but in our implementation, aimed at real-time tracking, we use multiple CQTs for different octaves, thus mitigating the delay problem. For this reason we focus our attention to this method. In Table 2 we see the results for a sine wave of 440 Hz. We can observe that in Run 3 the correct bin has

Run	Pitch	Velocity
1	[69] 440Hz	127
	[69] 440Hz	34
2	[69] 440Hz	127
	[69] 440Hz	35
3	[69] 440Hz	121
	[69] 440Hz	127
	[68] 415Hz, [70] 466Hz	41, 39
4	[69] 440Hz	127
	[69] 440Hz	34
5	[69] 440Hz	127
	[69] 440Hz	34

Table 2. 5 Runs of the CQT Algorithm without Auto-Tune on a -6 dB FS sine wave. Spurious pitches have been detected in neighbouring bins.

been detected first with the strongest velocity, followed by the 2 neighbouring bins. For this reason the introduction of the Auto-Tune mode seems a promising approach to mitigate these occurrences. In Table 3 we show the results for the same input, with Auto-Tune activated.

In both examples, the second pitch, with a lower velocity, is triggered at the end of the sine wave.

In Figure 5 we show the view from the DAW of 5 takes for a polyphonic input. We can see that different runs have different delays. Delay depends on the actual frequency content, as a result of running multiple CQTs, and the position within the internal buffer when the input is captured.

[1] https://vb-audio.com/Cable/

[2] https://reaper.fm

Run	Pitch	Velocity
1	[69] 440Hz	123
	[69] 440Hz	30
2	[69] 440Hz	127
	[69] 440Hz	37
3	[69] 440Hz	127
	[69] 440Hz	34
4	[69] 440Hz	127
	[69] 440Hz	34
5	[69] 440Hz	127
	[69] 440Hz	34

Table 3. 5 Runs of the CQT Algorithm with Auto-Tune on a -6 dB FS sine wave.

Figure 5. Example of 5 takes for a test sound, showing different delays.

Synthesized sounds, as well as recordings, mono- and polyphonic, have been employed for testing. Test materials, as well as the settings used for the application, are available on GitHub.

5. CONCLUSIONS AND FUTURE WORKS

With the development of *Arpeggiatorum* we wanted to provide a tool that can be readily employed by organ players, expand playing techniques on organs (ultimately also for other instruments), and foster cooperation and future collaborations. *Arpeggiatorum* is scheduled to be used in concerts and other performances. The valuable feedback provided by musicians will determine how the tool will evolve and what features will be introduced.

The next phase for *Arpeggiatorum* includes both an extension of its core features to address requests received

during the preliminary evaluation, and a more thorough examination of the affordances that are granted by this application. In terms of functionalities we plan to introduce an audio feature extractor that would allow to map input parameters (e.g. roughness) to arpeggiator parameters (e.g. pattern).

The source code for the project is available on GitHub: (https://github.com/axelberndt/Arpeggiatorum).

Acknowledgments

This research is supported by KreativInstitut.OWL [3]: a consortium consisting of OWL University of Applied Sciences and Arts, Detmold University of Music, and Paderborn University, funded by the Ministry of Economic Affairs, Industry, Climate Action and Energy of the State of North Rhine-Westphalia, Germany.

6. REFERENCES

[1] J. W. Robinson and S. L. Howell, "Automatic arpeggiator," *The Journal of the Acoustical Society of America*, vol. 74, no. 5, pp. 1666–1666, 1983. [Online]. Available: https://doi.org/10.1121/1.390092

[2] R. Arar and A. Kapur, "A history of sequencers: Interfaces for organizing pattern-based music," in *Sound and Music Computing Conference, Stockholm, Sweden*, 2013.

[3] M. Lozej, "The Arpeggiator: A Compositional tool for Performance and Production," Master's thesis, York University, Canada, 2016. [Online]. Available: http://hdl.handle.net/10315/32712

[4] P. Burk, "JSyn-a real-time synthesis API for Java," in *Proceedings of the 24th International Computer Music Conference (ICMC)*, 1998.

[5] J. Six, O. Cornelis, and M. Leman, "TarsosDSP, a Real-Time Audio Processing Framework in Java," in *Proceedings of the 53rd AES Conference (AES 53rd)*, 2014. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=17089

[6] R. Bencina and P. Burk, "PortAudio - an open source cross platform audio API," in *Proceedings of the 27th International Computer Music Conference (ICMC)*, 2001.

[7] A. M. Noll, "Pitch determination of human speech by the harmonic product spectrum, the harmonic surn spectrum, and a maximum likelihood estimate," in *Symposium on Computer Processing in Communication, ed.*, vol. 19. University of Broodlyn Press, New York, 1970, pp. 779–797.

[8] ——, "Short-time spectrum and "cepstrum" techniques for vocal-pitch detection," *The Journal of the Acoustical Society of America*, vol. 36, no. 2, pp. 296–302, 1964.

[9] A. De Cheveigné and H. Kawahara, "YIN, a fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 2002. [Online]. Available: https://doi.org/10.1121/1.1458024

[10] J. C. Brown and M. S. Puckette, "An efficient algorithm for the calculation of a constant Q transform," *The Journal of the Acoustical Society of America*, vol. 92, no. 5, pp. 2698–2701, 1992.

[11] B. Blankertz, "The Constant Q Transform," 1999, TU-Berlin, unpublished notes. [Online]. Available: https://www.user.tu-berlin.de/blanker/drafts.html

[3] https://kreativ.institute