Revista Vórtex | Vortex Music Journal | ISSN 2317–9937 | http://vortex.unespar.edu.br/
D.O.I.: https://doi.org/10.33871/23179937.2021.9.2.22

Analysis and resynthesis of real instruments using Pure Data and the MKMR library

Miguel Moreno

México

Abstract: In this experience report, I present my method for synthesizing realistic sounding instruments. Starting from the analysis of a marimba sample to how to recreate it using additive synthesis while reducing the number of objects, memory and CPU load. This synthesized model only uses pd vanilla objects, allowing an ease of implementation with projects using libpd. The instrument detailed in this article and many more are part of my open-source library: mkmr. This repository is a compilation of vanilla made abstractions with different categories such as: oscillators, filters, effects, instruments, waveshapers, sequencers, etc.

Keywords: Additive Synthesis, Pure Data, Spectrograph, Idiophones, Instrument Design.

Received on July 1st, 2021. Approved on: November 1st, 2021. Available online on: 30/11/2021. Guest Editor: Dr. Alexandre Torres Porres.

One of the most common ways of creating realistic sounding musical instruments is through sampling. It is quite advantageous having any musical timbre in a personal computer without having to buy it, tune it, store it or learn how to play it. Nevertheless, this requires plenty of storage since modern digital samplers are comprised of several recordings of every note, expression and dynamic in order to capture the full range of sounds an instrument can produce. For example, commercial virtual instruments such as Spectrasonics' Keyscape offers a wide variety of sounds produced by keyboard instruments, yet this comes at a cost with almost 80 GB for 36 instruments, that's around 2.22 GB per instrument[1]. However, available open source sampled instruments such as FluidSynth lack the expressivity of their commercial counterparts[2]. On the other hand, physically modelled instruments such as PianoTeq have been able to produce realistic sounds of a piano, an organ, a marimba, and much more with less than 50 MB per instrument[3]. Synthesis methods such as modal synthesis have also been able to mimic the sounds of real instruments such as xylophones, marimbas and bells with a low file size. In this experience report, I share my process on how to resynthesize idiophone instruments with additive synthesis using Pure Data.

The instrument detailed in this article and others are available on my open-source library: mkmr[4]. Currently, it features around 13 instruments such as a string ensemble using karplus-strong synthesis, an electric piano using modal synthesis, realistic sounding percussion and more. It is an ever-growing library of abstractions with a focus on synthesis and instrument sound design. Its use of pd vanilla objects allows an ease to implement them on any project using libpd such as Camomile[5], MobMuPlat[6], etc.

---

[1] Keyscape is a keyboard virtual instrument library released by Spectrasonics in 2016. Available at: <https://spectrasonics.net/products/keyscape/index.php>. Accessed on: 23 Nov 2021

[2] FluidSynth is a cross-platform, real-time software synthesizer based on the Soundfont 2 specification. Available at: <https://fluidsynth.org/>. Accessed on: 23 Nov 2021

[3] Pianoteq is a physically modelled virtual instrument by Modartt. Available at: <https://modartt.com/pianoteq>. Accessed on: 23 Nov 2021

[4] mkmr is a compilation of abstractions and instruments made with pd vanilla objects by Miguel Moreno. Available at: <https://github.com/MikeMorenoDSP/pd-mkmr>. Accessed on: 23 Nov 2021

[5] Camomile is an audio plugin with Pure Data embedded. Available at: <https://github.com/pierreguillot/Camomile> Accessed on: 23 Nov 2021

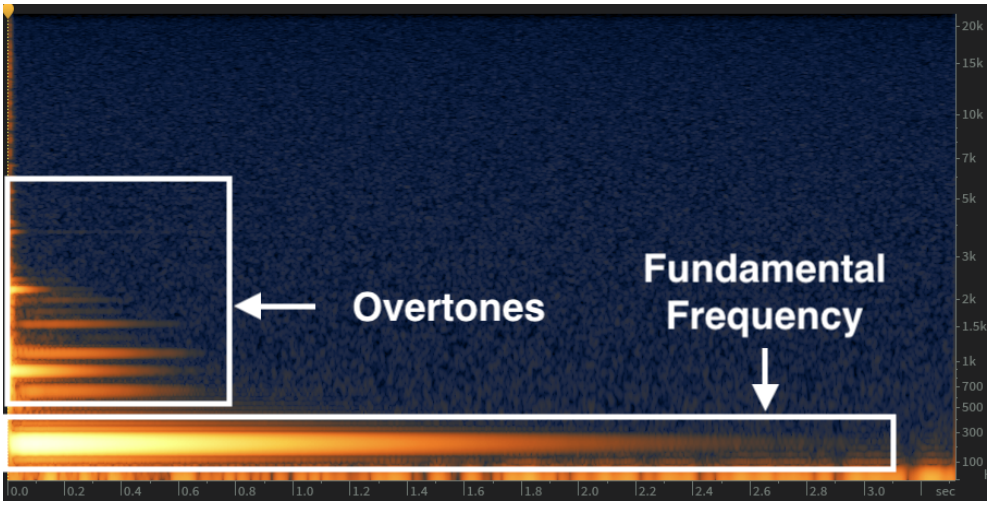
[6] MobMuPlat is an iOS and Android app for audio software made with Pure Data, with user-created interfaces Available at: <https://mobmuplat.com> Accessed on: 23 Nov 2021

I first tried this method with a recording of a struck plant pot. The sound was so tonal and rich I thought it would be a nice idea to create a synthesized version of this recording, then processing the sample into different pitches. So with my training in sound design and music production, I set out to analyze the sample with a spectrogram tool I've been using: iZotope RX[7]. With its noise reduction, filtering and deconstruction modules, I managed to isolate the core sound into 3 parameters: frequency, amplitude and decay.

1. Reference Sample Analysis

For example, let's analyze a marimba sample grabbed from University of Iowa's Electronic Music Studios sample library[8]. The sample is an A3 note in *fortissimo* with a cord mallet. By observing the spectrogram image, the difference between the fundamental frequency and the overtones is notorious. Especially by looking at the length and color of each line, which represents the harmonics decay and amplitude. By dragging the mouse from the start to end of each line, we can measure each harmonic's decay in milliseconds.

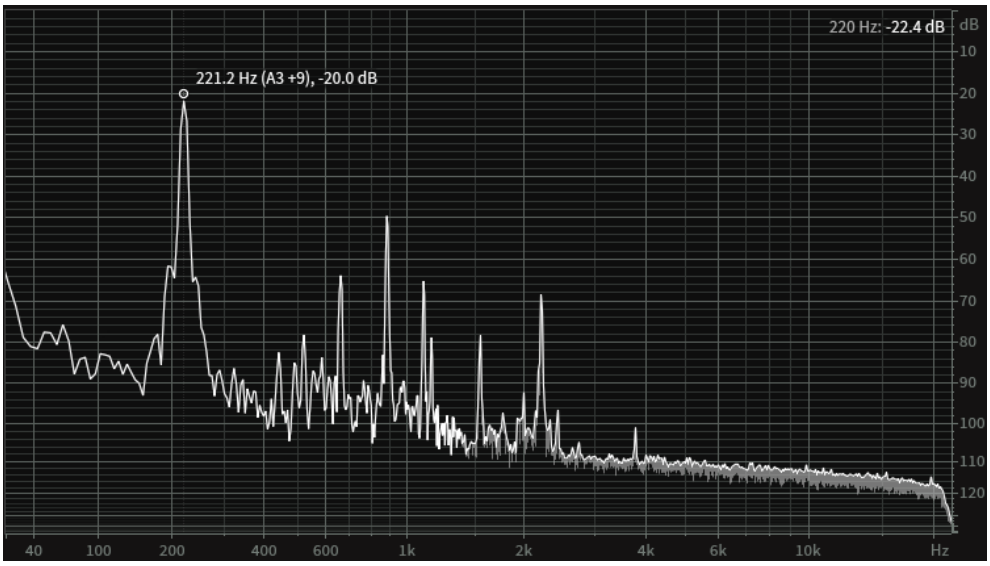
FIGURE 1 – Spectrogram. Useful tool for measuring the partial's frequency, amplitude and decay.



[7] RX is a software application for audio repair by Izotope. Available at: <https://izotope.com/en/products/rx.html> Accessed on: 23 Nov 2021

[8] University of Iowa Musical Instrument Samples is a library of various instrument recordings free of use. Available at: <http://theremin.music.uiowa.edu/Mismarimba.html>. Accessed on: 23 Nov 2021

FIGURE 2 –Spectrum. Useful tool for measuring the partial's frequency, amplitude and decay.



With the spectrum, we can also analyze the frequency ratio between the fundamental frequency and its overtones. By hovering the mouse on top of each peak, it displays its frequency and volume in decibels. Then by writing this information on a spreadsheet, we can start measuring and drafting our synthesized model.

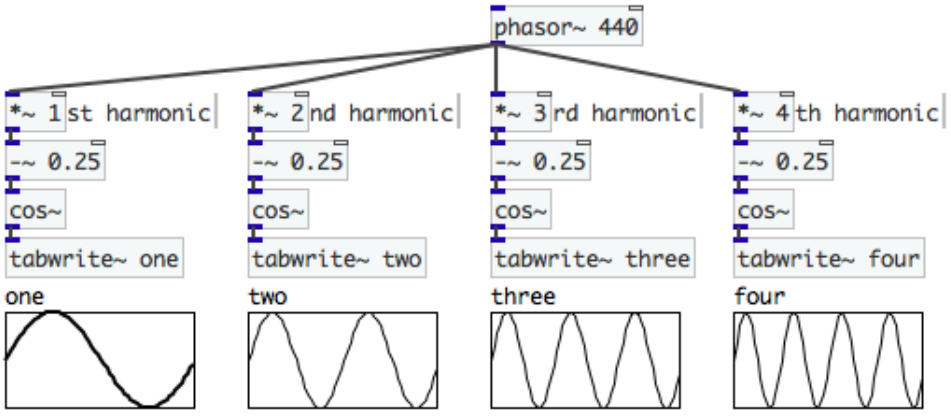
TABLE 1 – Measured data from spectrum analysis.

Harmonic Frequency	**Frequency Ratio (fn / f1)**	**Amplitude (dB)**	**Decay**
f1= 220 Hz	1.00	-9.5 dB	2967~ ms
f2 = 660 Hz	3.00	-49.9 dB	760~ ms
f3 = 880 Hz	4.00	-36.1 dB	760~ ms
f4 = 1100 Hz	5.00	-51.8 dB	760~ ms
f5 = 1150.6 Hz	5.23	-65 dB	300~ ms
f6 = 1537.8 Hz	6.99	-64.5 dB	620~ ms
f7 = 2195.6 Hz	9.98	-53.5 dB	150~ ms
f8 = 3748.8 Hz	17.04	-83.4 dB	70~ ms

2. Frequency Component

Of course, we could make individual sine wave oscillator objects for each harmonic (osc~), but there is a way to create this set of harmonics with a single [phasor~] object and keep all oscillators in phase with each other. This can be achieved by multiplying the output of [phasor~] by the harmonic number (*~ n) and that signal through a sine waveshaper (-~ 0.25) (cos~). With this, we have full control over the amplitude of each harmonic and set individual envelopes for each one.

FIGURE 3 – Additive synthesis with a single-phase generator and multiple sine table lookups.



Almost all frequency ratios are integers, except for f5 with a value of 5.23. But we can approximate this frequency with a value of 5.25 by dividing our oscillator's frequency by 4, letting us create non-integer harmonics. If we consider our fourth harmonic the fundamental frequency, then everything below four will be a sub-harmonic and everything in between a non-harmonic partial. In this example, let's make 220 Hz our fundamental frequency, so our oscillator's frequency would need to be 55 Hz, and the 21st harmonic of 55 Hz would have a frequency ratio of 5.25 and would be equal to 1155 Hz, just two hertz below f5. With these simple operations, we can determine the rounded frequency ratio given our oscillator's frequency.

FIGURE 4 – Method for more "Resolution" per harmonic.

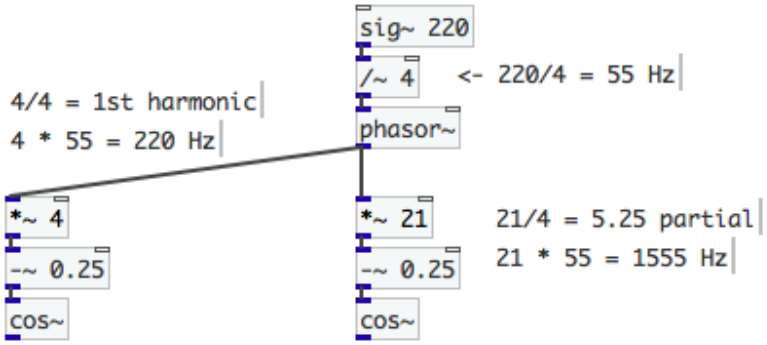


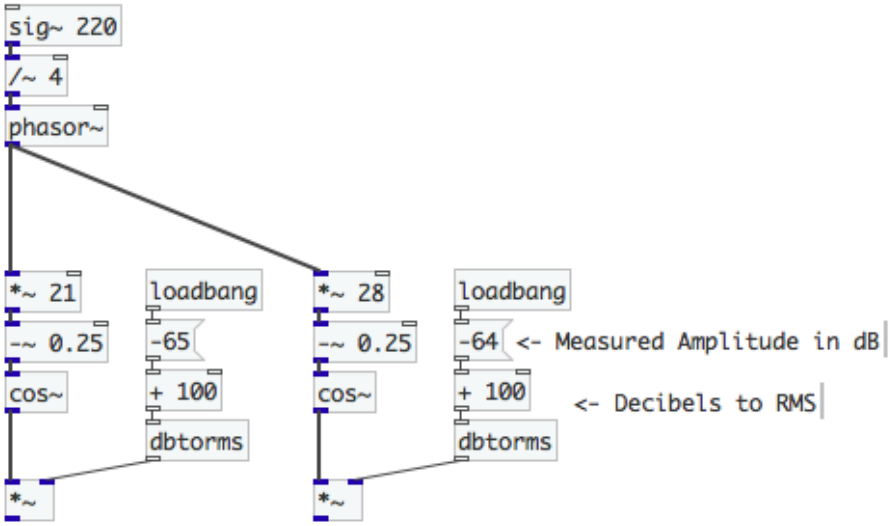
TABLE 2 – Rounded Frequency Ratio.

Harmonic Frequency	Frequency Ratio (fn / f1)	Amplitude (dB)	Decay	Rounded Frequency Ratio (fn / 55 Hz)
f1= 220 Hz	1.00	-9.5 dB	2967~ ms	4
f2 = 660 Hz	3.00	-49.9 dB	760~ ms	12
f3 = 880 Hz	4.00	-36.1 dB	760~ ms	16
f4 = 1100 Hz	5.00	-51.8 dB	760~ ms	20
f5 = 1150.6 Hz	5.23	-65 dB	300~ ms	21
f6 = 1537.8 Hz	6.99	-64.5 dB	620~ ms	28
f7 = 2195.6 Hz	9.98	-53.5 dB	150~ ms	40
f8 = 3748.8 Hz	17.04	-83.4 dB	70~ ms	68

3. Amplitude and Decay

After calculating the frequency ratios for our partials, implementing the amplitude seems like an easy task, but we have to consider this may vary depending on the position in which the amplitude values were measured for each partial. It is suggested to grab the amplitude value of the partial on its peak amplitude. Sometimes it helps to isolate a single partial using FFT filtering, but if this still doesn't provide the desired result, it can be tweaked manually until it does. In Pure Data, implementing this is relatively simple, considering only one message and two objects need to be created: a message with the measured value in dB, summing 100 to the value and connecting it to a [dbtorms] object, effectively converting our decibel value into RMS.

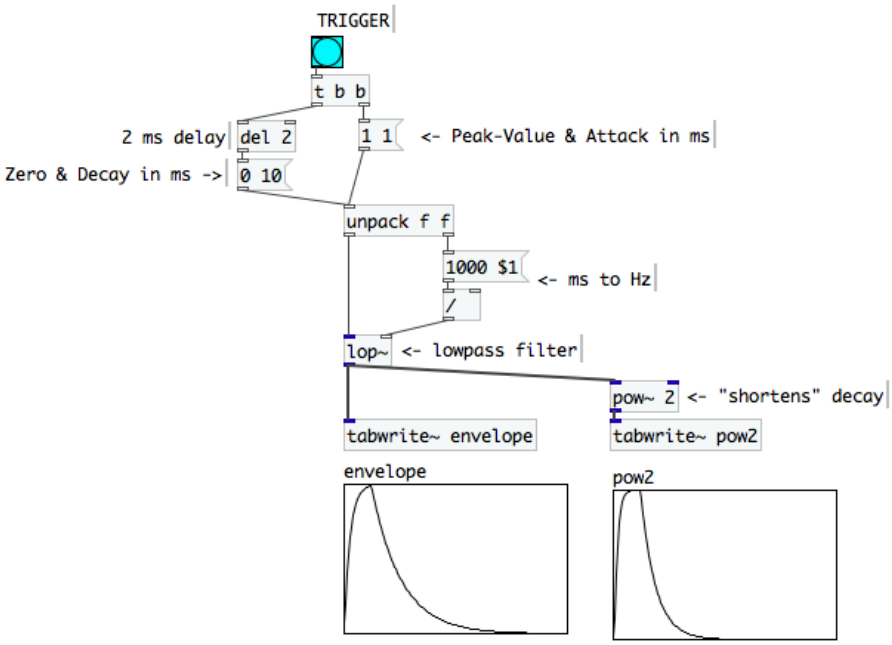
FIGURE 5 – Amplitude implementation.



After all the amplitude values have been assigned, it is time for implementing the envelopes. Similar to the frequency component section, it is possible to make individual envelope objects for each harmonic using [vline~], but there's also a way to control the decay of each harmonic with a single envelope. This envelope can be made with a one pole low pass filter like the object [lop~]. To trigger this envelope, one must send four values: the peak amplitude value, the time of attack, the final amplitude value (zero) and the decay time.

In Pure Data this second set of values has to be delayed by a minimum of two milliseconds in order to work. The result is an exponential Attack Decay envelope (AD envelope). The output provides a ramp with a "clicky" attack and natural long decay. Then, by raising its output to the power of *n* we can shorten the decay time, effectively having two decay times with a single envelope.

FIGURE 6 – Exponential Attack Decay Envelope with curve shaping using pow~.

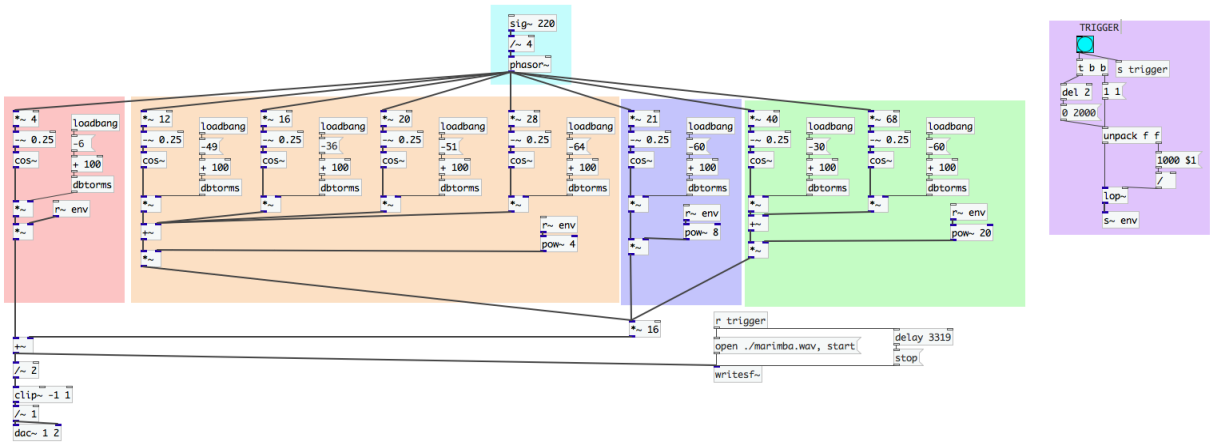


It is also important to note the relationship between frequency and decay. Mapping the measured decay of each frequency usually yields good results. Nevertheless, setting the decay as the wavelength of the frequency in milliseconds (t = 1000/f) and raising that to the power of *n* also creates an almost natural decay for each frequency. Likewise, limiting the decay to a certain range with the use of the clip object can also help.

4. Final implementation

For the final implementation, it is advisable to sum the harmonics with similar decays into a bus, so they use the same envelope curve. As we listen to it and compare it to the original sample, it may not sound as similar. That's when an AB analysis and comparison with a spectrogram helps correct the missing details. This can be done by recording our model with the [writesf~] object with the same duration as our reference sample. In this case, all the overtones were missing amplitude, raising their volume around 24 dB fixed that issue. The decay of the main envelope was also longer than the reference sample, the real decay of the marimba was actually around 2 seconds rather than 3 seconds. Overall, with these corrections, the synthesized marimba model started to sound similar to the real instrument.

FIGURE 7 – Final implementation with grouped harmonics



Finally, velocity implementation also needs to be addressed. Since we were working with a marimba played in *fortissimo,* then the dynamic can only go down from that. Normally for idiophones, the reduction in dynamics does two things: less overtone amplitude and less decay. The first can be done by summing all overtones into a single bus and linking the velocity to its amplitude. Reducing the amplitude by 12 dB or more should achieve the sense of dynamic. It should also be noted that the amplitude of the fundamental also needs to be reduced, perhaps less than the overtones. For the decay it can be as easy as linking the velocity to the decay, reducing it by 10 percent or more or using a pow~ object for changing the main envelope's curve.

5. MKMR library

The instrument detailed in this article (marimba~) and many more are part of the mkmr library. This library started as a repository to store abstractions compatible with pd vanilla like mmb or heavylib, mainly to suffice the necessity for resonant filters in pd vanilla[9]. Then, I started designing instruments for my algorithmic compositions with the goal of making music using only synthesis. My necessity of having realistic sounding instruments based on samples was born.

[9] mmb is a library of general abstractions for Pure Data and heavylib is a library of abstractions compatible for the hvcc compiler. Available at: <https://github.com/dotmmb/mmb> & <https://github.com/enzienaudio/heavylib> Accessed on: 26 Nov 2021

Eventually, this library ended up being my main tool kit for most of my projects. It will continue to be updated on Deken and GitHub with new instruments and helpful abstractions, as well as new articles or videos detailing my methods[10]. It is also worth mentioning some of these instruments are available as audio plugins using Camomile such as Euklid and EP-MK1.

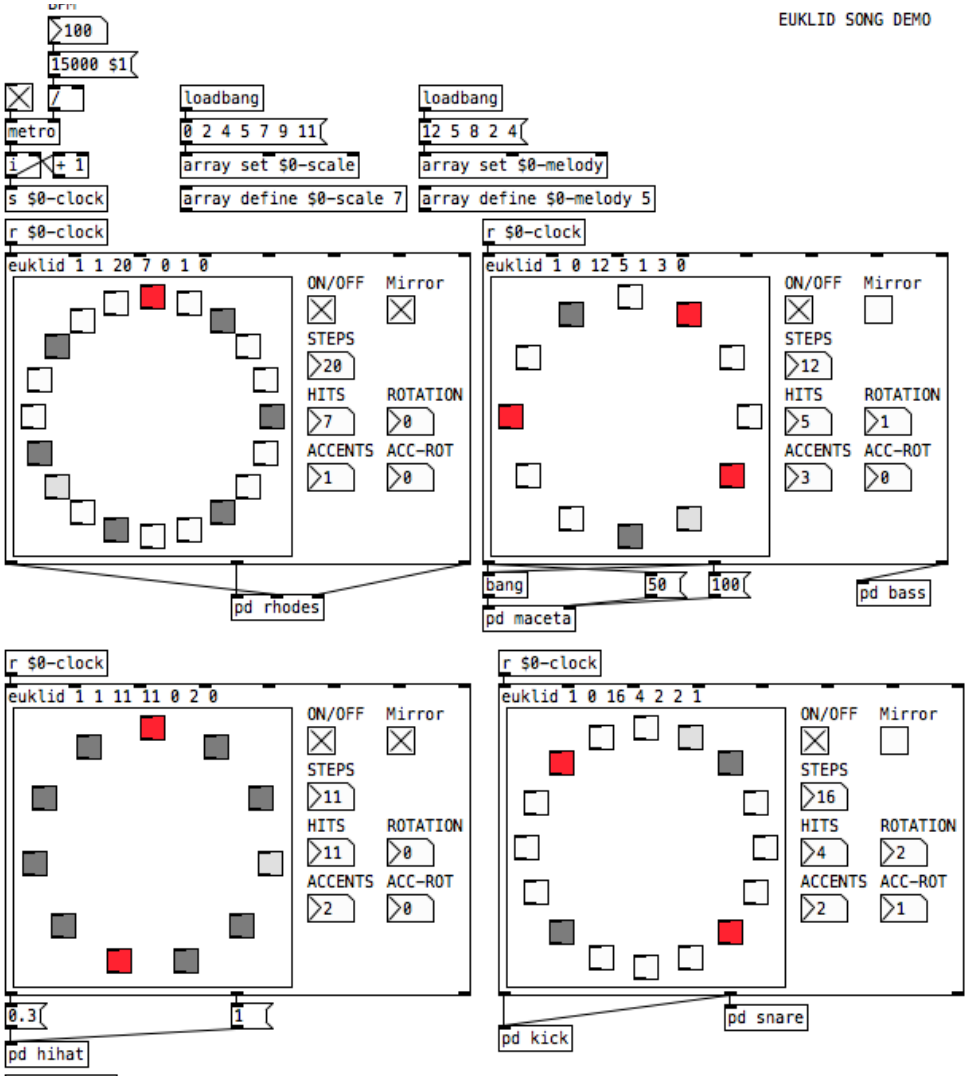
Here's a list of some notable instruments and other abstractions:

- marimba~ : a synthesized marimba instrument made with the method detailed in this article.
- gamelan~ : a synthesized gamelan instrument made with the method detailed in this article.
- mct~ : a plant pot instrument made with the method detailed in this article.
- mymembrane~ : a drum instrument made with extended karplus-strong synthesis[11].
- apcym~ : a cymbal instrument made with extended karplus-strong synthesis.
- str~ : a string ensemble made with karplus-strong synthesis.
- piano~ : a synthesized piano instrument made with karlpus-strong synthesis.
- cmbcym~ : a cymbal instrument made with comb filters (karplus-strong).
- ep-mk1 : an electric piano made with a combination of modal synthesis and distortion.
- os.wavetable~ : a continuous table lookup for cross-fading between multiple waveforms.
- fx.downsample~ : a down sampling effect with interpolation.
- euklid : a euclidean rhythm generator with a circular GUI made using dynamic patching
- ws.wavefolder : a linear wavefolder.

[10] Some of these articles will be published on my personal website <https://mikemorenodsp.github.io/>

[11] The extended karplus strong algorithm emulates string tension with the use of an all pass filter in the feedback section of the delay <https://ccrma.stanford.edu/~jos/pasp/Extended_Karplus_Strong_Algorithm.html> Accessed on: 28 Nov 2021

FIGURE 8 –Euklid: euclidean rhythm generator.



6. Conclusion and further work

Even though this is a pretty decent method for synthesizing idiophones, it is not perfect in many ways. Inharmonic partials can be somewhat achieved by raising the "resolution" of the oscillator, but ultimately lacks the precision. Also, some high-end frequencies may be missing for the strike phase (around the first 10 to 50 ms). Sound design-wise, these missing harmonics can be created with a mixture of phase modulation synthesis, wave shaping and even subtractive synthesis. If more "strike" needs to be added, a short decay envelope would suffice. Sometimes coupled with a soft clipping module or a Phase Modulator. In addition, a Low Pass Gate could simulate the strike and fall in amplitude of the harmonics without the need of multiple curve shapers for the envelope.

There's also the possibility of making these sounds even more inharmonic with the use of frequency shifting methods that could "stretch" or "contract" the harmonics. Of course, these solutions would require separate oscillators or a filter which increases a bit of CPU load.

REFERENCES

DOEL; PAI. *Modal Synthesis for Vibrating Objects*. Vancouver, Canada. University of British Columbia. Available at: <https://persianney.com/kvdoelcsubc/publications/modalpaper.pdf>. Accessed on: 23 Nov 2021

FRITTS, Lawrence. *Music Instrument Samples*. Iowa, U.S.A. University of Iowa. Available at: <http://theremin.music.uiowa.edu/Mismarimba.html>. Accessed on: 23 Nov 2021

GUILLOT, Pierre. *Camomile:* creating audio plugins with pure data. Saint-Denis, France. University Paris 8. Available at: <https://lac.linuxaudio.org/2018/pdf/44-paper.pdf>. Accessed on: 26 Nov 2021

ABOUT THE AUTHOR

Miguel Moreno. Sound designer and audio programmer. For more than 7 years, he has explored the sonic capabilities of digital sound synthesis with Pure Data. He studied Music Production Engineering at TEC de Monterrey. He has given workshops and presentations at several mexican music schools such as CEDART, CCM, Centro Cultural España and TEC de Monterrey. Within his experience, he has developed and designed synthesizers and drum machines for various independent musical technology companies. Moreno's latest open-source work is an extensive library of synthesized instruments for Pure Data. ORCID: https://orcid.org/0000-0003-0643-0805. E-mail: mikemorenodsp@gmail.com