Mimesis and Shaping

Imitative Additive Synthesis in Csound

Joachim HEINTZ
Incontri - HMTM Hannover
Emmichplatz 1, 30175 Hannover
Hannover, Germany
joachim.heintz@hmtm-hannover.de

Abstract

This paper is to introduce a realisation of Imitative Additive Synthesis in Csound, which can be employed for the realtime analysis of the spectrum of a sound suitable for additive synthesis. The implementation described here can be used for analysing, re-playing and modifying sounds in a live situation, as well as saving the analysis results for further use.

Keywords

Spectral analysis, sound synthesis, signal processing, realtime resynthesis.

1 What is Imitative Additive Synthesis?

Additive Synthesis is known as the method of synthesizing sound by single sinusoids. Based on a fundamental frequency and a number of sinusoids, the main parameters are

1. a frequency multiplier, and
2. a relative amplitude

for each partial. For instance, the well-known additive synthesized bell by Jean-Claude Risset has the values listed in table 1.[1]

Partial Number	Frequency multiplier	Amplitude multiplier
1	0.56	1
2	0.56 + 1 Hz	0.67
3	0.92	1
4	0.92 + 1 Hz	1.8
5	1.19	2.67
6	1.7	1.67
7	2	1.46
8	2.74	1.33
9	3	1.33
10	3.74	1
11	4.07	1.33

Table 1: Bell spectrum based on Risset 1969

In this case, the frequency/amplitude values were derived from the analysis of a natural sound. This is an example of what I call "Imitative Additive Synthesis", as opposed to creating spectra which do not exist anywhere in the non-electronic world of sound.

In real-world sounds, a table like the one given above is just a snapshot. The actual amplitudes vary all the time. Or, in other words, each partial has its own envelope. It is again Jean-Claude Risset who has described this early, when he analyzed a trumpet tone. The visualization can be done in a three-dimensional way like this:

[1] From Risset's *Introductory Catalogue of Computer Synthesized Sounds* (1969), cited after Dodge/Jerse, Computer Music, 1985, p. 94

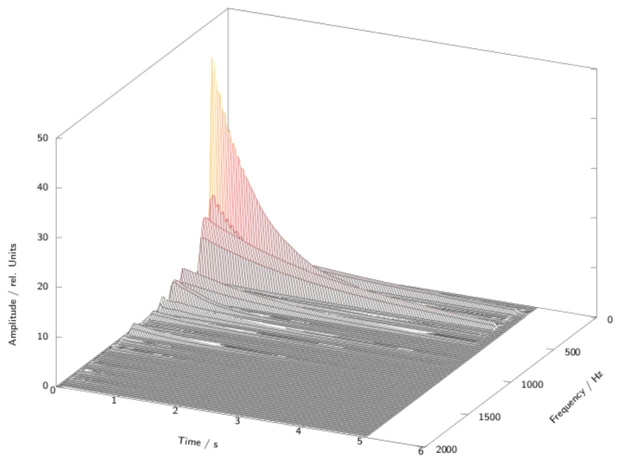


Figure 1: Partial progression of a guitar tone courtesy of Wolfgang Fohl, Hamburg

The method described here will start with the imitation of a spectrum. For this reason it is called imitative additive synthesis. But "imitative" does not mean that the spectral envelopes of the sound are to be imitated in their progression. Rather, the partials can be modified in different ways which is to be described in more detail.

2 Tasks for Performing Imitative Additive Synthesis in Realtime

In order to use Imitative Additive Synthesis in realtime performance, the Csound application should do these jobs:

- Analyze any sound input - sampled or live - and retrieve the N strongest partials.
- Let the user switch between live input or sampled sound. For the latter, let him choose one sample from a bank of sounds.
- If the input is a sound sample, set the position of the analysis by means of a pointer. Provide several options for moving the pointer:
  - User-controlled by a slider.
  - Continuously moving in defined steps forwards or backwards.
  - Jumping randomly in a given range.
- If the input is a live audio stream, analyze it whenever a new trigger signal (a midi button or a mouse click) has been received.
- Resynthesize the sound with additive synthesis with any number of partials up to N, and with possible offset (not starting at the strongest one).
- Allow for variation between resynthesized notes of the same snapshot by way of random frequency deviations, amplitude changes and by modifying the duration of the partials.
- Facilitate playing the synthesis on a usual midi-keyboard:
  - Define one key which plays the sound at the same pitch it has been recorded.
  - Define a transposition range for the other keys.
  - Let the user control a usual ADSR envelope.
- Allow to print out the analysis results, and to export them in different ways for further use.

The following description shows how these tasks can be realized in Csound. Andrés Cabrera's QuteCsound frontend will be used. It provides an easy-to-program graphical interface which gives the user a visual feedback and lets him control parameters either by midi or by widgets.

3 Retrieving the N strongest partials of a sound and triggering the resynthesis with M partials

The usual way of analyzing the frequency components of a sound is the Fast Fourier Transform (FFT). Thanks to the "phase vocoder streaming" opcodes, FFT in Csound is both simple and fast. There are several opcodes which transform an audio input (realtime, buffer or soundfile) into an "f-signal" which contains the spectral components.

The problem is how to extract the N strongest frequency components total number of bins.[2] This is done by the following operation:

- After performing the FFT, all the amplitude and frequency values of the analyzed sound snapshot are written in the tables *giamps* and *gifreqs*. This can be done in Csound by the opcode *pvsftw*.
- Then, the amplitude table is examined to return the positions of the N strongest values. These positions are written again in a table

[2] The number of bins or analysis channels of the FFT depends on the size of the analysis window. If the window size is 1024 (which is a common size), you will get 512 values with pairs of frequency and amplitude (bin 0 is omitted).

gimaxindc. This is done by a function which was programmed for this task.[3]

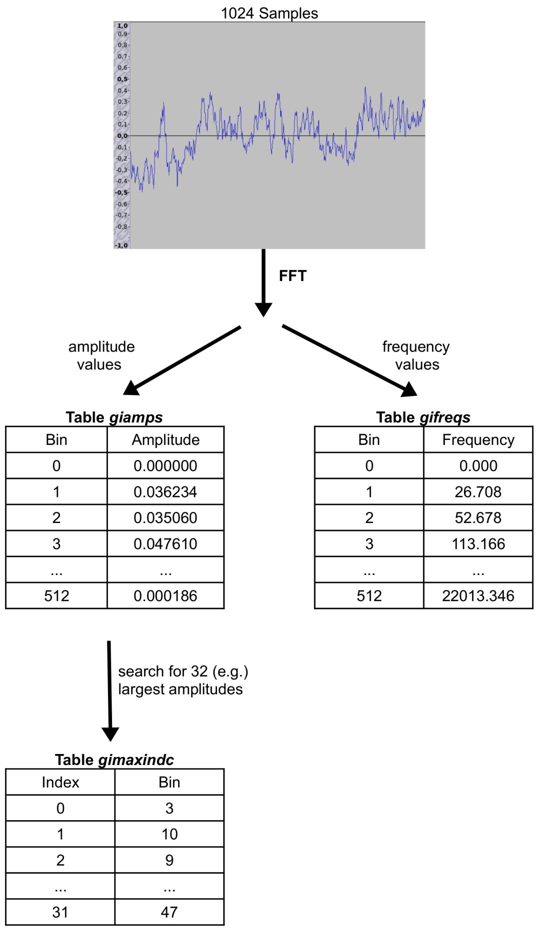


Figure 2: Analysing a sound snapshot

- Whenever a note is to be played, the total sum of the amplitudes required for the resynthesis of M partials[4] is calculated. This is necessary for two reasons:
  - The sound input may have very different levels, but the user will want the output volume to depend only on the velocity of the midi-keyboard.
  - If you decided during playing to reduce the number of sinoids M from say 20 to 2, you may nevertheless want to keep the same output level.
- For each note then, the *gimaxindc* table is read for the first M positions - eventually shifted by an offset -, and for each position one instance of an instrument is called. This instrument plays one partial, and it is fed with the relevant input values: the amplitude and frequency of this bin, the summed amplitude, the midi velocity, the midi pitch.

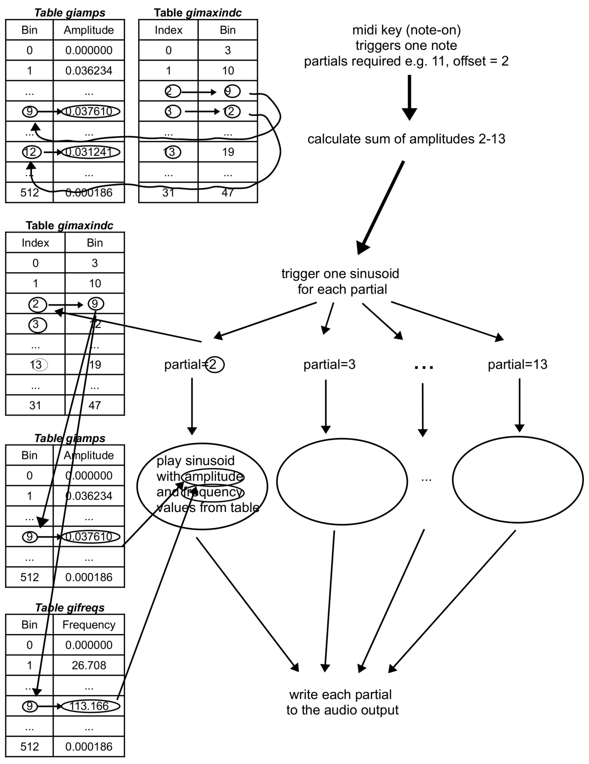


Table *giamps*
Table *gimaxindc*
Table *gifreqs*
midi key (note-on) triggers one note partials required e.g. 11, offset = 2
calculate sum of amplitudes 2-13
trigger one sinusoid for each partial
partial=2
partial=3
...
partial=13
play sinusoid with amplitude and frequency values from table
write each partial to the audio output

Figure 3: Triggering the resynthesis

4 Input and Time Pointer Options

Input can be selected from either a bank of soundfiles, or live input. There is a switch to determine which is to be used.

If soundfiles are used, the most important decision is in what way time pointer should be applied. One option is to manually adjust the position, either by mouse or by a midi-controller. But it can also be nice to "hop" through a sample, starting at a certain position, in a variable hop size. Each time a note is played, the pointer moves a bit

[3] In Csound, defined functions are called User Defined Opcodes. After defining in the orchestra header or an #include file, they can be used like usual opcodes. For more information, see the page OrchUDO.html in the Canonical Csound Manual (available online at www.csounds.com/manual/).

[4] In the range 1 to N

forward or backward. Fast repetitions can cross the whole file, like a flip-book. The third option implemented is a random jumping of the pointer, getting a new position after each note.

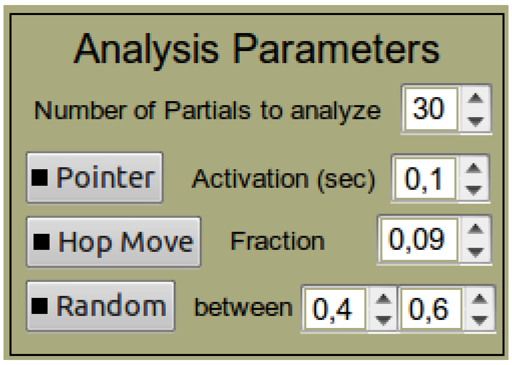


Figure 4: Analysis and pointer parameters

Live input is streamed continually, and if the user pushes a button, the incoming sound is analyzed in the way described above, performing an FFT and identifying the strongest partials for additive resynthesis. As long as the button has not been pushed again, the previous input remains the basis for the synthesis.

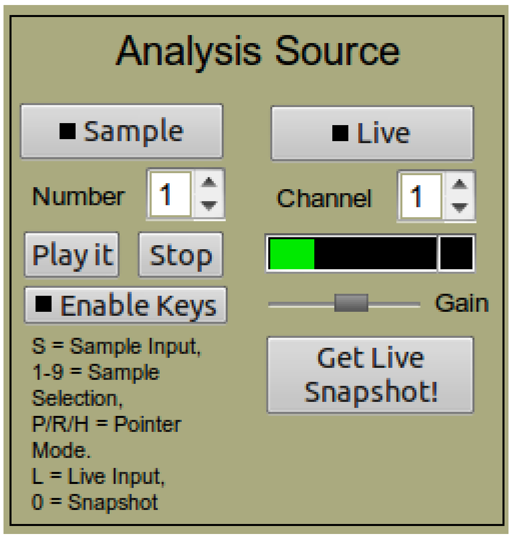


Figure 5: Input parameters

As can be seen in Figure 5, the user can switch between the input and pointer choices by keyboard shortcuts.

5 Playback Options

There are many options for playing back the sound snapshot. Some basic features have been implemented; some more ideas are discussed at the end of this section.

The user can decide how many partials he wants to use for the additive resynthesis. This part of the instrument is separated from the analysis, so the snapshot can be probed for the strongest 32 partials, and then all 32, or 10, 5, or just one can be used. An offset parameter allows to start not always at the strongest partial, but at a weaker one. So you can synthesize a rich spectrum or a more sine-like one, and choose whether you want to prefer the most significant or the lesser significant partials. But the partials will always remain ordered by their respective amplitudes.

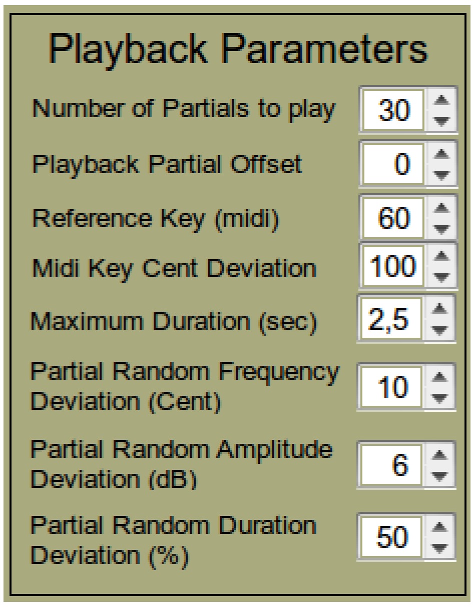
To avoid always producing the same sound, the user can add random deviations, both to the frequency and to the amplitude multipliers. The maximum frequency deviation is given in cent values, so 100 means that each partial can reach a frequency deviation of up to one semitone. The maximum amplitude deviation is given in deciBel, so 6 means that an amplitude multiplier of 0.5 can vary in the range between 0.25 and 1. With these values, you will get a set of sounds which are different each time, but nevertheless recognizable as one sound.

This random deviation within a certain range is also applied to the individual durations of the partials. Like in natural sounds, each partial has its own "life span". The simplest way of doing this is to assign random deviations to each partial. This is technically possible, because synthesis is carried out by one instance of a sub-instrument for each partial. So it is no problem to give each partial its own duration. The input is given in percent values, 100 meaning that the main duration of 2 seconds can vary between 0.5 and 4 seconds.

A common user-definable adsr envelope is applied, defining the attack time, the decay time, the sustain level, and the release time. It is the same shape for all the partials, but because of the duration differences, the overall shape will differ between the partials.

For playing the sounds via a usual midi keyboard, a key must be defined which plays the sound snapshot at the same pitch as it has been recorded. This reference key can be set by the user arbitrarily. Every other key will transpose the sound. The degree of transposition can also be adjusted by the user, to a semitone, or to any other value. If you set this parameter to 50, the next key on the midi keyboard will shift the sound by 50 cent (a quartertone). If you set it to 200, the next key shifts by a whole tone. If you type 0, the sound will not be transposed at all, so it will be at the same pitch on all keys.

This is the user interface for all these options:



Playback Parameters

Number of Partials to play	30
Playback Partial Offset	0
Reference Key (midi)	60
Midi Key Cent Deviation	100
Maximum Duration (sec)	2,5
Partial Random Frequency Deviation (Cent)	10
Partial Random Amplitude Deviation (dB)	6
Partial Random Duration Deviation (%)	50

Figure 6: Playback parameters

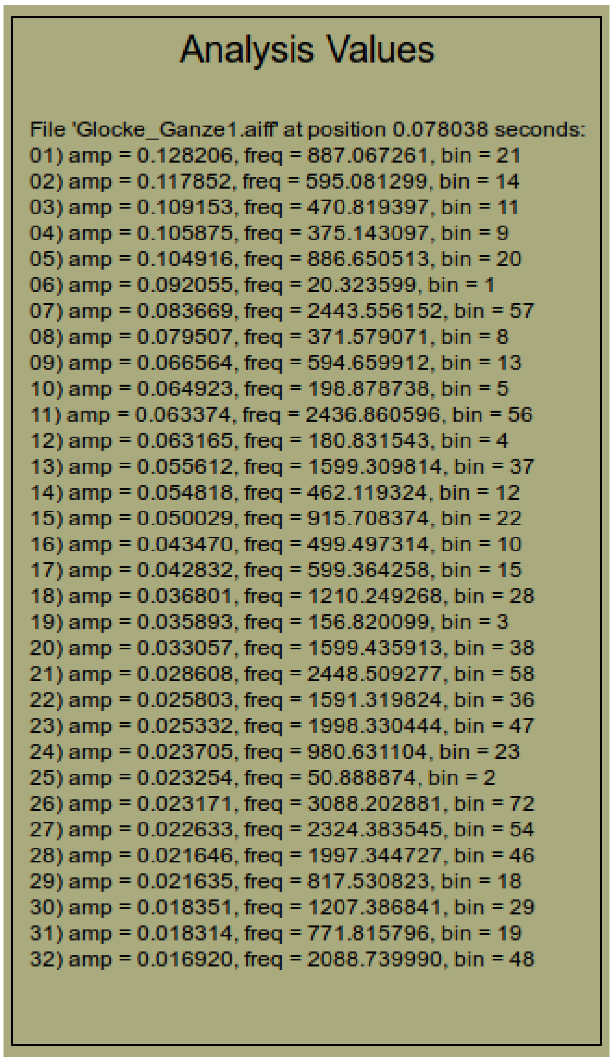
The playback options very much depend on which kind of music the user wants to play, and how they want to use the instrument. These are some ideas for other possibilities:

- Different tuning systems instead of equal steps from one key to the next.
- Manipulations of the partial relationship to become more harmonic or more inharmonic.
- Make partial durations depend on the pitch so that higher partials decay faster.

6 Export Options

The instrument described here can also be used to perform an FFT analysis and to query for the N strongest bins in this situation. For later use of the analysis results, some export options are implemented.

First, the user can choose to see the results in the gui. This is just a list of values for frequencies and amplitudes, like this:



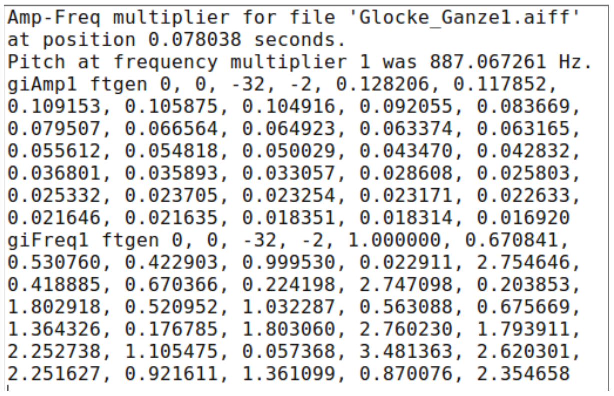
Analysis Values

```
File 'Glocke_Ganze1.aiff' at position 0.078038 seconds:
01) amp = 0.128206, freq = 887.067261, bin = 21
02) amp = 0.117852, freq = 595.081299, bin = 14
03) amp = 0.109153, freq = 470.819397, bin = 11
04) amp = 0.105875, freq = 375.143097, bin = 9
05) amp = 0.104916, freq = 886.650513, bin = 20
06) amp = 0.092055, freq = 20.323599, bin = 1
07) amp = 0.083669, freq = 2443.556152, bin = 57
08) amp = 0.079507, freq = 371.579071, bin = 8
09) amp = 0.066564, freq = 594.659912, bin = 13
10) amp = 0.064923, freq = 198.878738, bin = 5
11) amp = 0.063374, freq = 2436.860596, bin = 56
12) amp = 0.063165, freq = 180.831543, bin = 4
13) amp = 0.055612, freq = 1599.309814, bin = 37
14) amp = 0.054818, freq = 462.119324, bin = 12
15) amp = 0.050029, freq = 915.708374, bin = 22
16) amp = 0.043470, freq = 499.497314, bin = 10
17) amp = 0.042832, freq = 599.364258, bin = 15
18) amp = 0.036801, freq = 1210.249268, bin = 28
19) amp = 0.035893, freq = 156.820099, bin = 3
20) amp = 0.033057, freq = 1599.435913, bin = 38
21) amp = 0.028608, freq = 2448.509277, bin = 58
22) amp = 0.025803, freq = 1591.319824, bin = 36
23) amp = 0.025332, freq = 1998.330444, bin = 47
24) amp = 0.023705, freq = 980.631104, bin = 23
25) amp = 0.023254, freq = 50.888874, bin = 2
26) amp = 0.023171, freq = 3088.202881, bin = 72
27) amp = 0.022633, freq = 2324.383545, bin = 54
28) amp = 0.021646, freq = 1997.344727, bin = 46
29) amp = 0.021635, freq = 817.530823, bin = 18
30) amp = 0.018351, freq = 1207.386841, bin = 29
31) amp = 0.018314, freq = 771.815796, bin = 19
32) amp = 0.016920, freq = 2088.739990, bin = 48
```

Figure 7: Analysis printout

This list can also be exported to a text file. Either this file contains the same information as the gui printout, or the plain frequency and amplitude data.

If the user wants to use the data in any Csound context, it can be useful to have them transformed in two generalized function tables: one containing the data for the frequency multipliers, one for the amplitude values, like this:



```
Amp-Freq multiplier for file 'Glocke_Ganze1.aiff'
at position 0.078038 seconds.
Pitch at frequency multiplier 1 was 887.067261 Hz.
giAmp1 ftgen 0, 0, -32, -2, 0.128206, 0.117852,
0.109153, 0.105875, 0.104916, 0.092055, 0.083669,
0.079507, 0.066564, 0.064923, 0.063374, 0.063165,
0.055612, 0.054818, 0.050029, 0.043470, 0.042832,
0.036801, 0.035893, 0.033057, 0.028608, 0.025803,
0.025332, 0.023705, 0.023254, 0.023171, 0.022633,
0.021646, 0.021635, 0.018351, 0.018314, 0.016920
giFreq1 ftgen 0, 0, -32, -2, 1.000000, 0.670841,
0.530760, 0.422903, 0.999530, 0.022911, 2.754646,
0.418885, 0.670366, 0.224198, 2.747098, 0.203853,
1.802918, 0.520952, 1.032287, 0.563088, 0.675669,
1.364326, 0.176785, 1.803060, 2.760230, 1.793911,
2.252738, 1.105475, 0.057368, 3.481363, 2.620301,
2.251627, 0.921611, 1.361099, 0.870076, 2.354658
```

Figure 8: Export as table values

7 Conclusion

This paper is to show how Imitative Additive Synthesis in realtime can be implemented in QuteCsound. The different options presented here likely strain the limits of accessibility; wanting to show what is possible. For really playing it as a live instrument, each user will adapt the code and the gui to their needs, omitting some features and concentrating on others.

8 Acknowledgements

Thanks to Anna for reading the manuscript.

References

[1] Charles Dodge, Thomas A. Jerse: Computer Music. Synthesis, Composition, and Performance. NewYork / London 1985

[2] The Canonical Csound Manual online: http://www.csounds.com/manual/

The QuteCsound file described here is part of the official QuteCsound distribution (Examples > Synths > Imitative_Additive) since march 2011 in the reporitories, or in future releases (after 0.6.1): http://sourceforge.net/projects/qutecsound

It can also be found for download at http://joachimheintz.de/soft/qcsfiles/Imitative_Additive.csd.