PERMUTATION SYNTHESIS

Giacomo Valenti
Control and Computer Engineering Dept.
Politecnico di Torino
giacomo.valens@gmail.com

Andrea Valle
CIRMA-StudiUm
Università di Torino
andrea.valle@unito.it

Antonio Servetti
Control and Computer Engineering Dept.
Politecnico di Torino
antonio.servetti@polito.it

ABSTRACT

This paper introduces an experimental digital sound synthesis technique called *Permutation synthesis*, which focuses around creating new waveforms by moving groups of samples (*chunks*) of existing waves. Similarly to granular synthesis, permutation synthesis is a time-based technique, i.e. it operates directly on the discrete waveform: by means of varying the length of the chunks, several perceptual effects can be obtained. The most important parameter in permutation synthesis is the *permutation frequency*, which is inversely proportional to the chunk length; the resolution of this parameter is directly related to the sampling frequency (given the fact that a chunk is always an integer number of samples), thus a time-quantisation error is defined.
An algorithm for real-time permutation implemented as a SuperCollider plug-in is described, consisting of 3 UGens performing permutation synthesis in slightly different ways. The resulting signals are then analysed and their time and spectral effects are justified by defining formulas that analytically quantify the results.

1. PERMUTATION SYNTHESIS AND THE TIME-DOMAIN APPROACH

Permutation synthesis, i.e. a synthesis by permutation of samples, is a time-based technique. Historically, the most well-known synthesis techniques (from additive to subtractive to various members of the modulation family) have their roots in the spectral domain. The time-based family is younger in comparison, and typically many of the techniques belonging to this area share an experimental, empirical approach.
Among them, granular synthesis is perhaps the most notable: taking small grains of existing sounds, shaping them with an envelope and then scattering them in time and frequency with variable rates introduced a whole new perspective in the creation of sounds [1]. Permutation synthesis is similar to a particular variant of granular synthesis, the so-called time-granulation [1]: here grains are taken from one or more existing files, an envelope is applied, and then the grains are reproduced over time. If the source of the grain is a single audio stream, granulation results in scrambling parts of the same signal, which is the principle of permutation synthesis. However, most granulation approaches operate by applying an envelope, thus eliminating most of the discontinuities. Moreover, grains are typically scattered in time following some stochastic distributions. On the contrary, in permutation synthesis time discontinuities are the main feature, and the scrambling process is organised following a precise time-pattern.
Other techniques like synthesis by fragmentation and growth or PSOLA are based on the same principle: moving, replacing, repeating groups of samples whose duration is typically under the time resolution of the human ear, thus resulting in radically new sounds.
Not every newly produced sound must come from an existing one: techniques like dynamic stochastic synthesis and synthesis by instruction give the user full creativity by letting him or her start from scratch: by guiding the position of the next sample in the former case, and by using a logic gate paradigm in the latter (In general see [2],[3]).

Copyright: ©2014 Giacomo Valenti, Andrea Valle, Antonio Servetti. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

2. DESIGN

At its core, permutation synthesis is quite simple, designwise: an array of samples is given as an input to the permutation process, the array is then divided into chunks of a given dimension, the chunks are rearranged in a new order, and the rearranged array is the output (see Figure 1). This rearrangement often creates situations in which two samples that were meant to be away from each other are now adjacent, and this typically results in a waveform discontinuity. As already said, discontinuities are the heart of permutation synthesis: they both distort and enrich the resulting sound; by simply moving block of samples (chunks), a new waveform is created that retains some features of the original one and introduces new ones.
As the process operates in the digital domain, the requirements are minimal: a stream of samples and a buffer where to store them in order to retrieve them for rearranging. The parameters supplied by the user are permutation frequency (internally converted into the chunk size) and the pattern after which the chunks will be scrambled. An interesting side effect of the permutation technique is that the process does not affect global amplitude of the signal, as it simply re-organises the waveform. Thus, no digital distortion (in the sense of clipping) is introduced by the process, and the amplitude peak of the source signal is translated in the amplitude peak of the processed one.

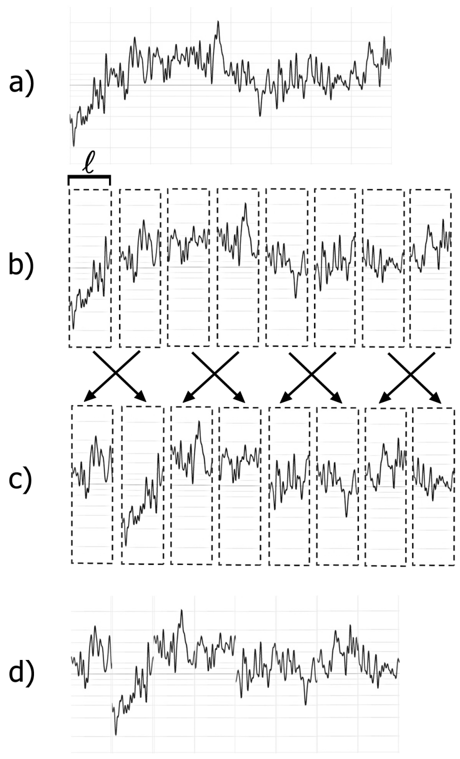

Figure 1. Permutation steps: a) the original audio source; b) division into chunks of length l; c) chunk scrambling; d) the permuted audio output.

2.1 Permutation frequency

Scrambling chunks is a trivial task, but indeed the time information related to a sample depends on the sampling frequency. A size of 50 samples does not provide any information on how much the chunk lasts; the chunk size parameter is then set implicitly by defining the permutation frequency (f_p) that indicates how often a new chunk occurs in time (to put it in another way: it is the discontinuity frequency). The size in samples of each chunk is then obtained by dividing the sampling frequency by f_p.

2.2 Time quantisation error

The conversion from frequency to chunk size leads indeed to non-integer values, that are not legal for the discrete domain. As a main concern is to maintain permutation frequency constant, all chunks are forced to have the same integer size by rounding. As an example, for $f_s = 48$ KHz and $f_p = 850$ Hz the chunk length is approximately 56.47 samples which gets rounded to 56. This means that different permutation frequencies could lead to the same rounded chunk size, which just in one case is the exact result of the division; in fact, the actual permutation frequency given by a 56 samples-long chunk is around 857 Hz. The difference between the expected f_p and the actual f_p can be seen as a time quantisation error.
The time quantisation error can be seen as an analogous in the time domain to digital quantisation error for amplitude, as it decreases with higher sampling frequencies. Moreover, for a given f_s it (globally) increases along with f_p (see Figure 3); time quantisation errors is more relevant with low sampling frequencies and/or high permutation frequencies, as shown in Figure 3.

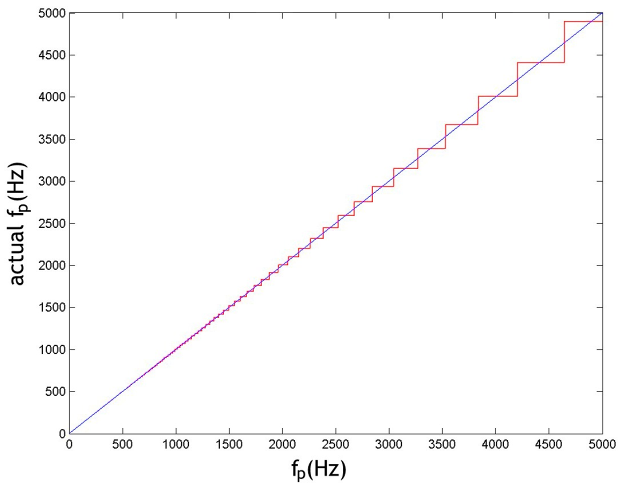

Figure 2. Desired f_p (straight line) versus actual f_p (stairs) given by a sampling frequency of 44100 Hz.

As f_p increases, the time quantisation error varies back and forth, always returning to zero but less and less frequently; however it is clear that its local (absolute) maximum increases with the permutation frequency and transversely decreases with higher sampling frequencies (see Figure 3).

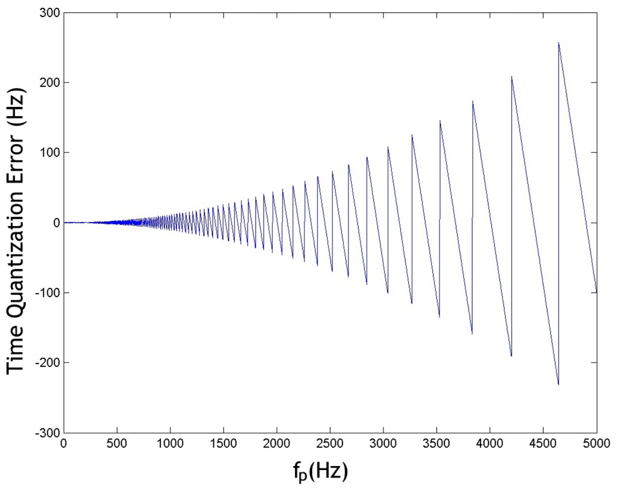

Figure 3. Time quantisation error for a sampling frequency of 44100 Hz.

2.3 Latency

If permutation synthesis is implemented in a real-time context, the presence of a buffer brings up a latency issue. An efficient indexing approach can be devised, in which samples are immediately written on the buffer according to the rearranged pattern, then the buffer is read from beginning to end and sent out to the audio stream. Changing the order of the samples inevitably leads to store sample chunks that are going to be played later, waiting for incoming samples to be played first. This means that to further minimise latency the buffer could be read while it is being written on, but never in a way that would lead to reading a position of a sample that still has to happen, and this is highly dependent on the chosen pattern.

3. PERMUTATION RESULTING EFFECTS

Until now it has been simply stated that discontinuities are the heart of permutation synthesis because they enrich and distort the sound. A more analytical discussion follows.

3.1 Perceptual effects

In permutation synthesis the more interesting perceptual phenomena take place in the permutation range $[5, 30]$ msec, in which various possibilities occur, from a continuous perceptual scenario to a discrete one. Depending on the permutation frequency, four perceptual behaviours emerge:

- $2 < f_p < 20 \Rightarrow$ *arpeggio effect*: discontinuities are clearly audible as clicks and the chunk length is such that an entire note or syllable could be swapped with another.
- $20 < f_p < 100 \Rightarrow$ *modulation effect*: the timbre is affected and notes are distorted with a sort of tremolo effect, each discontinuity is perceived as a new attack phase of the note.
- $100 < f_p < 6000 \Rightarrow$ *harmonic distortion effect*: a sort of robotic buzz is audible, proportional to f_p, which affects the pitch of each note; discontinuities are no longer perceived.
- $f_p > 6000 \Rightarrow$ *dynamic distortion effect*: from this values onward, the permutation yields a similar effect to dynamic saturation filters even though not even a single sample amplitude has been changed.

The transition between subsequent behaviours is very smooth and strongly depends on the listener perception and the source sound, to the point that in some cases there is a large range of values in which two adjacent behaviours overlap.
However it is still worth noting that for sounds that have a periodic spectrum like pitched notes f_p highly affects the pitch when it is around the fundamental, whereas for percussive sounds the effect resembles a notch filter, flavoring sounds according to f_p.

3.2 Temporal effects

The temporal effects, or more properly the waveform effects, are of course more predictable even without actually hearing the permuted sounds. However, for the sake of readability of the resulting waveforms, from now on the simplest scenario has been chosen: a sinusoid as the source and an even-odd ([1,0]) pattern for the permutation. If the permutation frequency is set as an even multiple of the sinusoid frequency, we are in fact permuting inside the period, creating a new waveform with the same pitch as the original but a different timbre, as seen in figure 4. If the

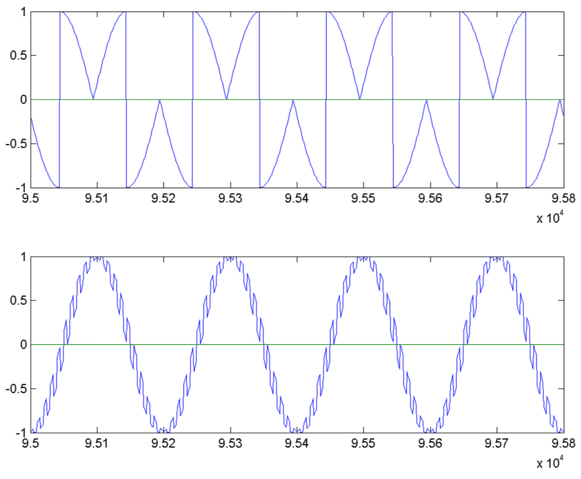

Figure 4. Permuting inside the period: with a sampling frequency of 44100 Hz a sinusoid at 220.5 Hz changes its waveform by being permuted with $f_p = 882$ Hz (top) and $f_p = 8820$ Hz (bottom).

permutation frequency is unrelated to the fundamental (for example by simply choosing a value which is not an integer multiple) a waveform whose real period is the least common multiple between the fundamental and f_p is obtained, but there can be still a noticeable pattern given by the periodic occurrence of discontinuities and the swapped portions of period of the original sinusoid, as seen in figure 5.

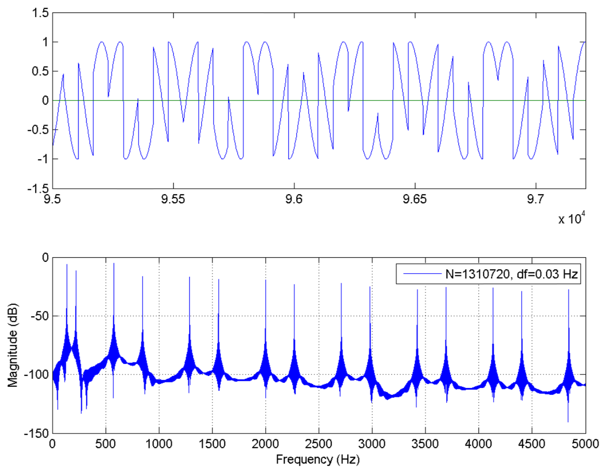

Figure 5. Waveform and spectrum of a 220.5 Hz sinusoid permuted with $f_p = 715$ Hz.

3.3 Spectral effects

The effects of permutation synthesis on the sound spectrum are here analyzed to provide a theoretical background to the perceptual features presented in Section 3.1. Here we keep using a simple waveform and an odd-even permutation pattern in order to have figures with an easily readable

spectrum and a simpler analytic demonstration of the phenomenon in section 4.
The spectrum in Figure 5 shows that, if a sinusoid is permuted with $f_p = 715$ Hz, then pairs of harmonic peaks are added to the fundamental frequency. These pairs of peaks appear near the odd harmonics of $f_p/2$ (which are not present) respectively at the frequencies $(2n+1)\frac{f_p}{2} - f$ and $(2n+1)\frac{f_p}{2} + f$. Obviously, those peaks that result in a negative frequency are aliased on the positive frequencies of the spectrum. Equation (1) represents this effect in an analytical form.

$$\sin(2\pi f t) \rightarrow \sin(2\pi f t) + \sum_{n=0}^{+\infty} \sin\left(2\pi\left[(2n+1)\frac{f_p}{2} - f\right]t\right) + \\ + \sin\left(2\pi\left[(2n+1)\frac{f_p}{2} + f\right]t\right) \quad (1)$$

4. ANALYTICAL DEMONSTRATION

If the permutation process is thought of as a kind of modulation of the source with a rectangular wave, then all the empirical observations presented in the previous sections can be justified by the following analytical demonstration.
Let us rewrite the process of permuting a chunk of samples with frequency f_p as follows:

- take two copies of the source.
- apply a rectangular periodic windowing function (whose period is twice the chunk length) to the first source, obtaining the even chunks.
- apply the same windowing function, but phase-inverted, to the second source, obtaining the odd chunks.
- translate the first windowed copy (i.e., the odd chunks) forward and the second windowed copy (the even chunks) backward by an amount of samples equal to the chunk length.
- sum the two signals.

If we invert the roles of the two signals and we consider the windowing function as the carrier signal and the source as the modulator, the whole process can be seen as a ring modulation of an unipolar square wave with a 50% duty cycle.
This perspective justifies the position of the peak pairs, in fact, the modulated signal spectrum, i.e. the square wave, contains only components of odd-integer harmonics frequencies centered at integer multiples of its fundamental frequency $f_p/2$.
Moreover, we see the peak pairs and not the the square wave odd harmonics themselves because ring modulation suppresses the carrier signal and creates sidebands of the positive and negative modulator frequency centered around the carrier frequencies [2]. Lastly, the presence of the modulator original spectrum frequencies is due to the fact that the unipolar square wave is not zero mean and so it has a continuous component at 0 Hz that is suppressed, but produces the peak at $+f$. All of the above considerations were made concerning the odd-even pattern, but the same logic and steps apply to any other pattern with few minor changes: the windowing function would then be a sum of step functions which would be translated according to each index of the chosen pattern.

5. IMPLEMENTATION

Permutation synthesis has been proposed preliminary without any analytical treatment in [4], that introduces a Non Real-Time implementation in the SuperCollider language, based on swapping chunks of values from an array representing the signal. The implementation was intended as a proof of concept, is computationally inefficient and not suited for Real-Time. A general algorithm for real-time permutation synthesis is shown in Figure 6.

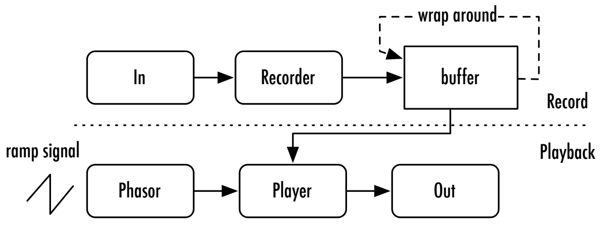

Figure 6. A general implementation for permutation synthesis.

The process is split into two parallel phases, Record and Playback. On top, a source (In) is recorded onto a recirculating buffer that wraps around overwriting old values at its beginning with new incoming ones. The second phase is Playback. Here a Player reads values from the buffer: its index must be provided by an opportune signal generator (Phasor) that is fed with a sawtooth-like signal. Buffer dimension is not particularly crucial if only it is able to store some periods of the incoming signal. This schema is apt to be implemented in various standard synthesis packages, such as Csound, PD, Max, SuperCollider, at the user level. However, in order to accurately check what happens at the sample level (typically not directly or easily accessible to the user in the aforementioned software packages), a real-time implementation of permutation synthesis has been written as a SuperCollider plug-in, i.e. as an extension of SuperCollider which consists of one or more UGens, a UGen being a basic specialised processing unit. By implementing a UGen, substantially a new audio primitive, also a greater efficiency is gained.
The SuperCollider environment[1] [5] has proven to be an apt choice as it allows to seamlessly bridge sound synthesis and algorithmic composition, thus providing also a high-level layer by which immediately experimenting with permutation synthesis in a musical context.
The SuperCollider architecture consists of two components: a server (named scsynth), dedicated to audio synthesis and processing, and a client (sclang), providing a language interface to the server. Every set of UGens must have an SC class on the language side that is bound to a C++ class, extending the server side [6].
The *PermUGens* plug-in is defined as a set of 3 different

[1] http://supercollider.sourceforge.net

UGens which allow the user to permute an existing audio source:

- **PermMod** swaps couples of chunks in an odd-even manner for a given f_p.
- **PermModArray** lets the user choose the number of chunks and the permuting pattern by providing an array of integers. As an example, the array $[1, 0, 2, 3]$ will swap the first two chunks while leaving the second two as they are.
- **PermModT** is similar to *PermMod* but adds a second frequency parameter whose ratio with f_p causes the introduction of a tail, a spare chunk with a different length, that is combined to the standard chunk.

The client side of each UGen is a simple SC class which defines the input parameters and does a first rough check of their validity. As an example, PermMod has an audio-rate (**ar*) method which takes an audio input (*in*), f_p (*freq*, set to 100 if not specified) and the standard SuperCollider *mul* and *add* parameters; if *in* is not audio-rate no information is even sent to the server module and no UGen is instantiated.

The server side is where audio processing really happens, written in C/C++ code. Inside the C++ class every UGen has several modules, the most important being the calculation functions to which the input audio samples are fed and the output is generated. Before talking into detail about the calculation function(s) of one of the PermUGens, it is worth noting that also the constructor has a very important role: for example in the case of *PermModArray* besides checking the validity of every input parameter and calculating the chunk length, it also calculates the jumping pattern from the array of integer fed by the user (more on that later); moreover it decides which calculation function to use and tests it by feeding it one single sample of audio. Since *PermModArray* is the most complete of the PermUGens, it is worth having a look at some extracts of the code to understand how it works.
In the actual implementation every PermUGen utilizes two buffers of the same size: while one is written the other is read and vice versa, therefore this is not the optimal setup latency-wise (see 2.3). Further versions of the algorithm should improve this feature.
As stated in section 2, every PermUGen relies on an indexing mechanism rather than actually moving the samples: this means that there is a single *read* variable that cycles throughout the buffers from beginning to end and back; the difference being that when it comes to the writing buffer *read* is summed to a *jump* amount in order to write samples according to the permuting pattern, while at the following cycle (which is when that buffer becomes the reading one) it gets read in sequential order as shown below:

```
if(swapped==false){

   swapbuf1[read+(pattern[index]*chunkl)]=in[i];
   out[i]=swapbuf2[read];
   }
else{

   swapbuf2[read+(pattern[index]*chunkl)]=in[i];
   out[i]=swapbuf1[read];
}
```

In the case of *PermModArray* the permuting pattern is inserted by the user when instantiating the UGen and it consists of an array of integers of length n: to be valid it must contain every cardinal number from 0 to $n-1$ (for example [2,1,3,0]); as anticipated in the previous paragraph, the *jumping pattern* (which is simply called *pattern[]* in the previous code extract) is calculated from the permuting one, and it is just another array of integers of the same size, but containing the *jump* amounts instead of the desired order of the chunks.
The most important thing when implementing a real-time synthesis UGen is allowing the user to change parameters on-the-fly. The rate at which this can occur is called *control rate* (**kr*) and its period corresponds to the audio buffer (typically 64 samples).
Since the chosen calculation function is called every control period, it also bears the responsibility of checking if a parameter has changed and to adjust the pipeline to reflect these changes. The algorithm allows to change f_p as well as the permuting pattern in real time and change the buffer size accordingly, that is, every time the new (actual) chunk length is different from the previous or the number of chunks has changed. For example if a value of the permuting pattern changes, the jumping pattern must be recalculated; whether f_p changes, the buffers have to be deallocated and reallocated and the chunk length recalculated for the next cycle. Here is an extract of the code triggered by a change of f_p:

```
if(newchunkl!=chunkl || numchunks!=newnumchunks){

    RTFree(unit->mWorld, unit->swapbuf1);
    RTFree(unit->mWorld, unit->swapbuf2);

    unit->chunkl = newchunkl;
    read = 0;
    index= 0;
```

6. EXAMPLES

The SuperCollider Permutation UGens can be used in various ways: they may serve as a distortion filter as well as a component of more complex UGen topologies of a virtual synthesizer (in SuperCollider called a *SynthDef*) [2].

6.1 Basic usage

A basic test unit has been provided in the help files, that allows to experiment with permutation effect by controlling the signal and the permutation frequency by means of the mouse:

```
{PermMod.ar(SinOsc.ar(MouseY.kr(0,440)),
    MouseX.kr(500, 8800))}.play
```

In the above example the source audio fed to the UGen is a sinusoid whose frequency is changed in real time every control period *(*kr)* proportionally to the Y coordinate of the mouse; f_p is also changed the same way but proportionally to the X coordinate. With this configuration, every position of the arrow cursor on screen leads to a different distortion of a different note and the interesting part

[2] Some audio examples are available here: http://www.fonurgia.unito.it/wp/?page_id=606

is hearing the effects generated by the real-time variation of the aforementioned parameters. Figure 7 is a sonogram of a signal resulting from the previous code, in which the user is controlling via mouse the two frequencies. It clearly shows that spectral richness can be obtained also by a minimal example. Discrete steps in the spectrum depends indeed by the chunk integer size constraint.

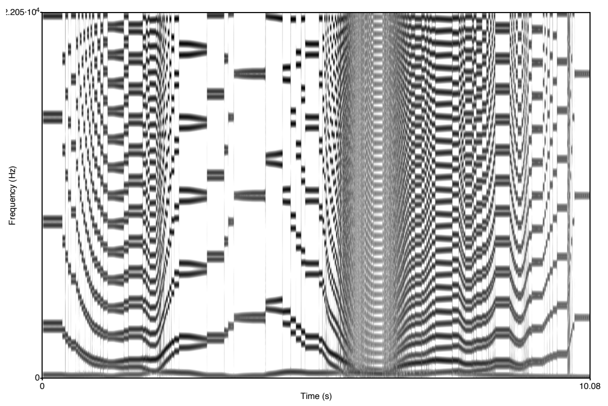

Figure 7. Effects of permutation on a sinusoid.

6.2 Wavetable permutation

In the example below, *PermMod* is a module of a more complex SynthDef that is capable of playing different notes by repitching an existing wave file. The pitch of each note is extracted and forced to chromatic scale by the *.cpsmidi* method, then this data is used to pilot the f_p parameter, therefore creating a new timbre.

```
c = Buffer.read(s, "C:/Folder/Shamisen-C4.wav");
SynthDef(\dist, { arg rate = 1;
    var sig = PlayBuf.ar(1, c, rate:rate,
    doneAction:2) ;
    var pt = Pitch.kr(sig)[0].cpsmidi ;
    Out.ar(0, PermMod.ar(
       sig,LFPulse.kr(
          1+(pt-36).midicps,0,0.5,
          pt.midicps+1,(pt+24).midicps)))
    }
).add ;
```

The example shows an application of permutation as a processing stage rather than as a pure synthesis technique.

6.3 Sequential Random Modulation

The next example consists of several stages of daisy chained modulations of a sine wave in which almost every signal and/or real-time parameter is driven by one or more of the three PermUGens. Since the code is too long to be included in the present paper, Figure 8 shows a diagram of its modules[3]. The main concept is that the last PerModArray UGen receives as its input a sinusoidal signal that has been already modulated by a previous PerModArray. This kind of iterated arrangement has been proposed many times for standard modulation techniques, in particular frequency and phase modulations[2] and shows how permutation is apt to be used as possible replacement in the same arrangements. An empirical investigation shows that the permutation algorithm introduces a DC bias, that in the example is removed as the final processing stage.

[3] We thank Marinos Koutsomichalis for providing the example and the diagram.

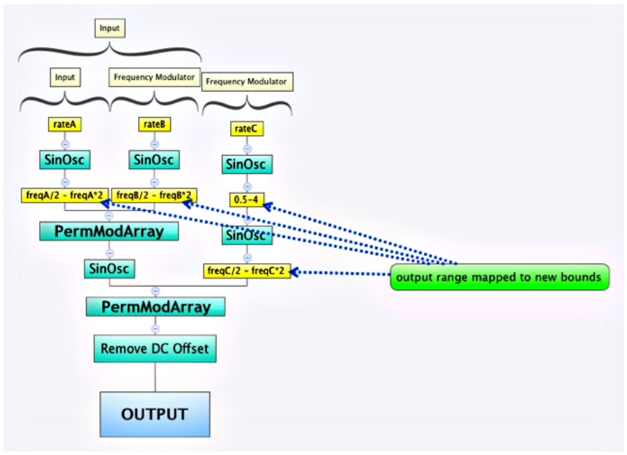

Figure 8. Modulation-Permutation diagram.

7. CONCLUSIONS

Permutation synthesis belongs to the family of non-standard synthesis technique. It does not refer to the electronic signal paradigm per se, rather it fully exploits the discrete nature of digital signals. It is a fairly simple method, indeed a purely digital member of the modulation family: nevertheless, it allows to obtain complex signals that show rich spectra but also specific temporal features, that depend on the time-domain operation at its basis. The permutation synthesis SuperCollider plug-ins have been compiled for Windows and Mac (but Linux porting is not an issue), and will be made available via the Quarks extension system in SuperCollider: user feedback is indeed relevant to assess the interest of the technique and to provide new examples of musical usage.

8. REFERENCES

[1] C. Roads, *Microsound*. Cambridge, Mass. and London: The MIT Press, 2001.

[2] C. Roads, *The Computer Music Tutorial*. MIT Pres, 1996.

[3] E. R. Miranda, *Computer Sound Design Synthesis techniques and programming*. Focal Press, 2nd Edition, 2002.

[4] A. Valle, *The SuperCollider Italian Manual*. CC Attribution-Non commercial-Share alike 2.5, available at www.cirma.unito.it/andrea/sc.html, 2008.

[5] S. Wilson, D. Cottle, and N. Collins, eds., *The SuperCollider Book*. Cambridge, Mass.: The MIT Press, 2011.

[6] D. Stowell, *The SuperCollider Book*, ch. Writing Unit Generator Plug-ins, pp. 691–720. Cambridge, Mass.: The MIT Press, 2011.