CDP EXTEND Functions

(with Command Line Usage)

Functions to EXTEND & segment soundfiles

(Names in brackets mean that these are separate programs. The others are sub-modules of EXTEND.)

BAKTOBAK
Join a time-reversed copy of the sound to a normal copy, in that order
[CERACU]
Repeat the source sound in several cycles that synchronise after specified counts
DOUBLETS
Divide a sound into segments that repeat, and splice them together
DRUNK
Drunken walk through source file (chosen segments read forwards)
[SFECHO ECHO]
Repeat a sound with timing and level adjustments between repeats
FREEZE
Freeze a portion of sound by iteration
[HOVER]
Move through a file, zig-zag reading it at a given frequency
ITERATE
Repeat sound with subtle variations
[ITERLINE]
Iterate an input sound, following a transposition line
[ITERLINEF]
Iterate an input sound set, following a transposition line
LOOP
Loop (repeat [advancing] segments) inside soundfile
[MADRID]
Spatially syncopate repetitions of the source soundfile(s)
REPETITIONS
Repeat source at given times
SCRAMBLE
Scramble soundfile and write to any given length
SEQUENCE
Produce a sequence from an input sound played at specified transpositions and times
SEQUENCE2
Produce a sequence from several sounds played at transpositions and times specified
[SHIFTER]
Generate simultaneous repetition streams, shifting rhythmic pulse from one to another
[SHRINK]
Repeat a sound, shortening it on each repetition
ZIGZAG
Read soundfile backwards and forwards, as you specify

ALSO SEE:

[MCHITER]

Iterate the input sound in a fluid manner, scattering around a multi-channel space

[MCHZIG]

Extend by reading back and forth in the soundfile, while panning to a new channel at each 'zig' or 'zag'

EXTEND BAKTOBAK – Join a time-reversed copy of the sound to a normal copy, in that order

Usage

extend baktobak *infile outfile join_time splice-length*

Parameters

infile – input sound to process
outfile – resultant soundfile
join_time – time in *infile* where join-cut is to be made
splice-length – length of the splice, in milliseconds

Understanding the EXTEND BAKTOBAK Process

I call the outputs of this process (learnt from Denis Smalley), fugu sounds. There is a particular Japanese fish delicacy, the fugu fish, which has a poisonous liver, but tastes best the nearer to the liver you slice the fish (!). Fugu sounds are made using an attack-resonance source (a sound with a sharp attack which then fades away to nothing). A Reverse copy of the sound is made and then spliced onto the original so the sound now grows from nothing until it reaches a peak of loudness and spectral brightness and then fades once again to zero.

Musical Applications

If the crossfade is made just before the peak is reached, the sound is less loud and bright in the centre, and in fact a whole sequence of such musically related sounds can be made, each of different loudness/brightness in the centre. This process allows such sounds to be made in a single pass.

End of EXTEND BAKTOBAK

CERACU – Repeat the source sound in several cycles that synchronise after specified counts

Usage

ceracu ceracu *insndfile outsndfile cyclcnts mincycdur chans outdur echo echshift* [**-o**] [**-l**]

Example command line to create polyphonic repetition-streams:

```
ceracu ceracu in.wav out.wav "cyclecounts.txt" 0.5 4 10 0 0
```

Parameters

insndfile – input soundfile (mono)
outsndfle – output (possibly multi-channel) soundfile
cyclcnts – datafile consisting of a list of integers, being the number of repeats in each cyclestream before the streams resynchronise
mincycdur – the time before the first repeat in the fastest cyclestream. If set to ZERO, it is assumed to be the duration of the input sound.
chans – number of channels in the output – NB: it is not necessarily the same as the number of cyclestreams
outdur – duration of the output. (If set to ZERO, it outputs a single resync-cycle.) The process always outputs a whole number of complete resync-cycles, equal to or greater than the specified output duration. If the true duration is greater than one hour, the sound is curtailed, unless the **-o** flag is set.
echo – single-echo-delay of entire output, in seconds. (Set to ZERO for no echo.)
echshift – Spatial offset of echo-delay (an integer value) – ignored if no echo. 1 = 1 chan to right; 2 = 2 chans to right; -1 = 1 chan to left, etc. Enter '0' if not using.
-o – override the duration restriction, to produce all resync-cycles (CARE!).
-l – output channels are arranged linearly (Default: arranged in a circle.)

Understanding the CERACU Process

CERACU creates polyrhythms (such as the familiar 2-against-3 pattern). In each of several sound streams, the source sound (which must be mono) is re-triggered at a regular time-interval, which is normally different for each stream. After a certain number of repetitions, the streams re-synchronise, completing a full "resync cycle". One complete pass is a 'resync-cycle', e.g., specified as 10, 12 and 15 in the *cyclecnts* textfile. The source repeats 12, 12 and 15 times before the cyclestreams resynchronise. The following diagram illustrates what happens. If the source sound is 'A', the cyclestreams would be:

```
A     A     A    A     A     A      A     A     A     A     A (10 times)
A    A    A    A    A    A   A     A    A    A    A    A    A (12 times)
A   A   A   A   A   A   A   A   A   A   A   A   A   A   A   A (15 times)
|--------Resync cycle = mincycdur (see below)--------|
```

The time-span of the cycle is determined by *mincycdur*, the shortest division (e.g. 3, in 2 against 3: more repeats within the time make them closer together). This division may mean that the source is not played in full within the cycle. Typically, the number of streams might equal the number of output channels; if not, some channels will be silent or contain a rhythm that is the resultant of two or more streams.

The process always outputs a whole number of complete resync-cycles, equal to or greater than *outdur*, the overall minimum output length. The final playing of the source in each channel always runs to its end and an extra tail of silence also seems to be added. If the duration would be greater than 1 hour, the sound is curtailed, unless the "override" [**-o**] flag is set.

Musical Applications

CERACU provides a means to explore rhythmic overlays of the same soundfile, overlays that are not only more complex, but also precisely defined. You can get overlays with the TEXTURE via the *packing* parameter – expecially interesting with TEXTURE MOTIFS – but without the degree of rhythmic timing that you can get with CERACU. The possibility of spreading the outputs across a multi-channel rig is another benefit. In this regard, you might also want to look at TEXMCHAN.

End of CERACU

EXTEND DOUBLETS – Divide a sound into segments that repeat, and splice them together

Usage

extend doublets *infile outfile segdur repets* [**-s**]

Parameters

infile – input soundfile
outfile – output soundfile, with repetitions
segdur – duration of segments
repets – number of times each segment is repeated
[**-s**] – option to have *outfile* try to stay synchronised with the *infile*

segdur may vary over time

Understanding the EXTEND DOUBLETS Process

EXTEND DOUBLETS is a 'slice' function, like the ones we are familiar with in the visual realm. The difference, here, in the temporal realm, is the *repetition* parameter. We specify the length of the segments (slices) and the number of times it repeats.

What we hear depends, as usual, on the sonic material. With voices or conventional music, the effect will be like the needle getting stuck on a vinyl record: a short passage repeats. With more complex sonic material, we would get a pulsing, mechanical effect. Especially note that the length of the segment (*segdur*) parameter can vary over time.

Musical Applications

Here are a few ideas to stimulate the imagination:

- a speaker repeats a phrase for emphasis
- a speaker repeats his phrase, adding more each time
- a sound repeats, unfolding as it does so
- extended time-varying extensions of a sound or syllable
- mechanical throbbing
- extending the material of a sound prior to using other sound transformation processes

End of EXTEND DOUBLETS

EXTEND DRUNK – Drunken walk through source file (chosen segments read forwards)

Splice segments of source file end-to-end; the start times of the segments in the source file are chosen by a 'drunken-walk' through the source file. In Mode **2** the source file plays soberly at holds.

Usage

extend drunk 1 *infile outfile outdur locus ambitus step clock* [**-s***splicelen*] [**-c***clokrand*] [**-o***overlap*] [**-r***seed*]
extend drunk 2 *infile outfile outdur locus ambitus step clock mindrnk maxdrnk* [**-s***splicelen*] [**-c***clokrand*] [**-o***overlap*] [**-r***seed*] [**r***seed*] [**-l***losober*] [**-h***hisober*]

Modes

1 Drunken walk
2 Play soberly at holds, with lower and upper limits of sobriety

Parameters

infile – input soundfile to process
outfile – output soundfile
outdur – total minimum duration of output soundfile (seconds)
locus – time in *infile* at which the drunken walk occurs (seconds) – this location can move through the source

> In breakpoint files, the LEFT HAND column refers to time locations in *outfile* of *outdur* duration, and the RIGHT HAND column refers to time locations in *infile*.
> This is also true for the *ambitus*, *step* and *clock* parameters, except that the right hand column in these cases contains timing data, not times.
> When used as a constant (single value), the *locus* time refers to a time location in the *infile*.

ambitus – half-width of the region from within which the sound segments are read (seconds)
step – maximum length of (random) step between segment reads (> 0.002 seconds); this always falls within the *ambitus*: it is automatically adjusted where too large
clock – time between segment reads: this is the segment duration (> *splicelen* * 2) (seconds)
mindrnk – minimum number of clock ticks between sober plays (1 - 32767 Default: 10)
maxdrnk – maximum number of clock ticks between sober plays (1 - 32767 Default: 30)
-s*splicelen* – length in milliseconds of the splice slope (Default: 15ms)
-c*clokrand* – randomisation of clock ticks (Range: 0 to 1 Default: 0)
-o*overlap* – mutual overlap of segments in output (Range: 0 to 0.9900 Default: 0)
-r*seed* – any set value gives **reproducible** output
-l*losober* – minimum duration of sober plays (seconds) (Range: > 0 to duration of *infile*+. If >= duration of *infile*, all sober plays go to the end of the source.

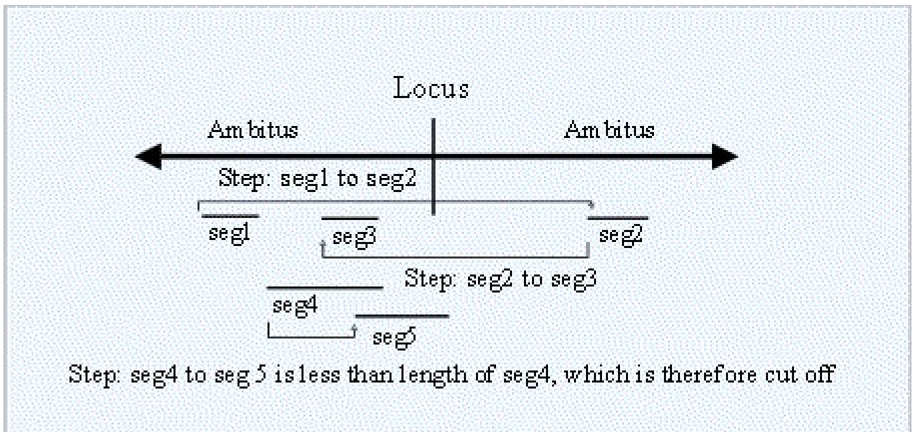
-h*hisober* – maximum duration of sober plays (seconds)
(Range: > 0 to duration of *infile*+.

All params except *outdur*, *splicelen* and *seed* may very in time.

Understanding the EXTEND DRUNK Process

Another approach to segmentation, EXTEND DRUNK takes a series of segments selected from the *infile* and splices them together to form the *outfile*. The process starts at some time in the file, called the *locus* and selects a segment, randomly, from **within** an 'ambit'. The length of 'ambit' is 2 * *ambitus*, and stretches to both sides of the *locus* position. Once a segment is read, the program moves (randomly) to a new position in either direction, **within** the ambit, and **not more than** *step* from the start location of the previous segment, from where it starts the next read. This is called a 'random walk'; hence the name 'drunk'.

The function is based on a drunken walk algorithm implemented by Miller Puckette.


Locus
Ambitus
Ambitus
Step: seg1 to seg2
seg1
seg3
seg2
Step: seg2 to seg3
seg4
seg5
Step: seg4 to seg 5 is less than length of seg4, which is therefore cut off

EXTEND DRUNK Locus, Ambitus & Step parameters

While it is doing this walk, one can shift the *locus*, e.g., progressing slowly through the file. (NB: note above about times in the breakpoint file.) The 'ambit' – the portion of the soundfile being used at any one time, can be varied by altering the size of the *ambitus*, which is one-half the full 'ambit' width. The *step*, which is the maximum distance between the start of one read and the start of the next read (but must lie within the ambit), can also be varied.

For example, if the *ambitus* is small, segments very close to one another will be selected. Or if the *step* is much smaller than the segment size (a slow *clock* produces longer segments), selected segments will tend to overlap, producing random echoes or pre-echoes. The length of the segment, determined by the *clock* parameter is NOT constrained to the size of the ambit, so segments may begin within the ambit and end outside it. Segments which would end beyond the specified *outdur* are truncated.

To summarise, then, *locus*, *ambitus* and *step* all refer to **start locations**. *Clock* refers to **segment length**.

The process continues until *outdur* is filled, which makes it a useful program with which to generate material.

Musical Applications

This is all about the fragmentation and texturing of sound. You can use this function to fragment a specific portion of a soundfile. Or you could make large jumps (*step*) in order to create surprise areas of fragmentation. Or EXTEND DRUNK can be used to churn up source material to varying degrees. The parameters allow a great deal of scope for variation, so some methodical study will be well rewarded. The following are some pointers regarding key parameters.

Musically, the issue is how much will the original soundfile be broken up. The various parameters contribute to this, each in their own way: the size of *ambitus*, the size of *step*, the length of the segment (*clock* and *clokrand*), and the position where this takes place (the *locus*). Introducing breakpoint files for time-varying effects adds another dimension again. Hence, for example, the *locus* might move gradually through the file, or moves back and forth in the file.

Clokrand randomises *clock*. If not used the output will consist of fixed lengths, a continuous stream of regular bursts of sound. *Clokrand* makes the segment lengths vary – in a time-varying way if a *time clokrand* breakpoint file is used.

Note that *clock* determines the length of the segments because the read continues until the next 'tick'. *Overlap* will increase the rate at which the segments come past: i.e., proportionately (0 to 0.9900) less than the length of each segment.

The *splicelen* (which can be quite long) can be used to soften the joins, thereby smoothing the output.

The *seed* parameter makes it possible to create a reproducible sequence. The 'random' sequence of numbers takes a fixed and hence repeatable form.

Finally, Mode **2** offers a hold mechanism whereby the unaltered *infile* continues to be read from where the read marker happens to be at the time. This can vary within limits, as set by *mindrnk*, *maxdrnk*, *losober* and *hisober*.

6 examples, 2-6 with breakpoint combinations

The *infile* used here is *balsam.wav* (1.997120 sec, mono, SR=44100), a vocal sound supplied with CDP's *GrainMill*.

To run these examples in ***Sound Loom***, you should **copy** this sound (or a similar sound of the same length) into the Workspace directory (on the Workspace Page). You should also **copy** the files:

locus3.brk, locus4.brk
ambitus1.brk, ambitus2.brk, ambitus3.brk, ambitus4.brk
step1.brk, step2.brk, step3.brk, step4.brk, step5.brk and
clock1.brk, clock2.brk, clock3.brk, clock4.brk, clock5.brk

onto the Workspace. You will find these files in the Support Pack *drunkexamples.zip* – NB: the command line .bat files are not needed for *Sound Loom*, which uses Patches. Then put *balsam.wav* on the Chosen Files list, press **Process** and select the **EXTEND drunkwalk** process.

For ***Soundshaper***, make sure that that the sound is in your sounds directory, and that all the other files listed above are in your TXT directory. Then load the Presets (same names as the Patches) and run Soundfiles / Extend / Drunk. (Check the path to the .brk files to ensure that the Presets match your setup.)

For **Command Line** operation, put all files in the current directory and run the batch (**.bat**) files.

EXAMPLE 1 – fairly large segments are from widely spaced locations in the *infile*. Fixed values are used. *Sound Loom/SoundShaper*: Load the Patch/Preset *drunk1*.

```
               INFILE OUTFILE  LENGTH LOCUS AMBITUS STEP CLOCK
extend drunk 1 balsam getdrnk1 25     1     .6      .2   .5
```

EXAMPLE 2 – move gradually through the *infile*, with segment size (*clock*) decreasing, while the scattering (*step*) increases. *Sound Loom/SoundShaper*: Load the Patch/Preset *drunk2*.
LOCUS: left column is time in *outfile*, right column is time in *infile*

```
locus3.brk  ambitus1.brk  step1.brk  clock1.brk
 0  .3       0  .1          0  .05     0  .3
10  .6      10  .2         10  .1     10  .2
15 1.2      15  .3         15  .15    15  .1
20 1.8      20  .2         20  .2     20  .05

extend drunk 1 balsam getdrnk2 25 locus3.brk ambitus1.brk step1.brk clock1.brk
```

EXAMPLE 3 – short segments (fast clock) are well-scattered within the full width (2 * *ambitus*) around each locus. *Sound Loom/SoundShaper*: Load the Patch/Preset *drunk3*.

```
locus3.brk  ambitus2.brk  step2.brk  clock2.brk
 0 .3        0  .2         0  .1       0  .05
10 .6       10  .2        10  .2      10  .1
15 1.2      15  .2        15  .3      15  .15
20 1.8      20  .2        20  .4      20  .1

extend drunk 1 balsam getdrnk3 25 locus3.brk ambitus2.brk step2.brk clock2.brk
```

EXAMPLE 4 – longish segments (slow clock) which are located very close to each other (tiny steps) around each locus. *Sound Loom/ SoundShaper*: Load the Patch/Preset *drunk4*.

```
locus3.brk  ambitus2.brk  step3.brk  clock3.brk
 0  .3       0  .3         0  .05     0  .2
10  .6      10  .2        10  .08    10  .3
15 1.2      15  .3        15  .1     15  .2
20 1.8      20  .2        20  .06    20  .3

extend drunk 1 balsam getdrnk4 25 locus3.brk ambitus2.brk step3.brk clock3.brk
```

EXAMPLE 5 – expansion outward from the centre: central *locus*, and the other parameters move from small to large. *Sound Loom/SoundShaper*: Load the Patch/Preset *drunk5*.

```
locus4.brk  ambitus3.brk  step4.brk  clock4.brk
 0  0.8      0  .1         0  .035    0  .05
 8  1.2     10  .2        10  .09    10  .15
16  0.8     15  .3        15  .15    15  .2
20  1.2     20  .4        20  .20    20  .3

extend drunk 1 balsam getdrnk5 25 locus4.brk ambitus3.brk step4.brk clock4.brk
```

EXAMPLE 6 – contraction inwards towards the centre: central *locus*, and the other parameters move from large to small. *Sound Loom/SoundShaper*: Load the Patch/Preset *drunk6*.

```
locus4.brk  ambitus4.brk  step5.brk clock5.brk
 0  0.8       0  .4         0  .2      0  .3
 8  1.2      10  .3        10  .15    10  .2
16  0.8      15  .2        15  .09    15  .15
20  1.2      20  .1        20  .035   20  .05
```

```
extend drunk 1 balsam getdrnk6 25 locus4.brk ambitus4.brk step5.brk clock5.brk
```

My thanks to Eitan Teomi whose queries helped me to understand DRUNK better and improve the documentation, and to Trevor Wishart for converting the command lines to *Sound Loom* Patches. [AE]

ALSO SEE:
DRUNK in the spectral set, which moves about in a similar way through analysis windows.
DRUNK TUTORIAL

End of EXTEND DRUNK

SFECHO ECHO – Repeat a sound with timing and level adjustments between repeats

Use **MODIFY REVECHO** if overlapping delays are required.

Usage

sfecho echo *insndfile outsndfile delay attenuation totaldur* [**-r***rand*] [**-c***cutoff*]

Example command line to create ... :

```
sfecho echo in.wav out.wav 6 0.6 12
```

Parameters

insndfile – input soundfile
outsndfile – output soundfile
delay – time in seconds between echo repeats (Range: greater than the length of *insndfile* to 3600 sec. [one hour]; thus *delay* cannot be less than the input duration)
attenuation – relative (diminishing) level of each repeat (Range: 0 to 1)
totaldur – maximum output duration (actual duration may be less); it must be a minimum of 2 x *delay*.
-r*rand* – randomisation of echo times (Range: 0 to 1)
-c*cutoff* – dB level at which decaying echoes cut off (Range: 0 to -96dB, Default: -96dB, i.e., silence)

Delay, *attenuation* and *rand* may vary over time.

Understanding the SFECHO ECHO Process

CDP's EXTEND LOOP enables you to step through a soundfile while adding each step-segment to an output soundfile. It does not allow you to specify an endtime beyond the end of the input sound (it cuts off). EXTEND REPETITIONS enables you to repeat a whole soundfile, whether overlapping or with a gap between repetitions: i.e., the time of repetition is beyond the end of the input sound. This new ECHO function complements these two features by placing the repeats *after* the end of the input soundfile.

End of SFECHO ECHO

EXTEND FREEZE – Freeze a segment of a sound by iteration in a fluid manner

Usage

extend freeze 1 *infile outfile outduration delay rand pshift ampcut starttime_of_freeze endtime gain* [**-s***seed*]
OR
extend freeze 2 *infile outfile repetitions delay rand pshift ampcut starttime_of_freeze endtime gain* [**-s***seed*]

Modes

1 Specify output duration
2 Specify number of repetitions

Parameters

infile – input soundfile
outfile – output soundfile
outduration – Desired duration of resultant soundfile.
repetitions – Number of repetitions of frozen segment.
delay – The (average) delay between iterations:< = length of frozen segment.
rand – Delaytime randomisation. Range: 0 to 1. Default: 0.
pshift – Maximum of random pitchshift of each iteration. Range: 0 to 12 semitones. E.g., 2.5 = 2.5 semitones *up or down*.
ampcut – Maximum of random amplitude reduction on each iteration. Range: 0 to 1. Default: 0.
starttime_of_freeze – Time where the frozen segment begins in the original sound.
endtime – Time where the frozen segment ends in the original sound.
-s*seed* – The same *seed* number will produce identical output on rerun. Default: 0 – random sequence is different every time.

Understanding the EXTEND FREEZE Process

Extend a specific part of a sound using the iteration procedure. **This tends to give a more convincing time-stretching result than any of the other time-stretch procedures**, particularly as the non-time-stretched portions of the sound are not subject to any processing. The internal proportions of a sound event can be manipulated using this process.

Musical Applications

The *start* and *end* times of the freeze enable you to focus on very specific parts of the sound, such as the 'a' in 'star' or the 's' in 'star'. With this program you can extend these to form sounds such as 'staaaaaaaaaaaaaaaaaaaar' or 'sssssssssssssssssstar'. Given the claim that it gives 'a more convincing time-stretching result than any of the other time-stretch procedures', it is a program well worth exploring thoroughly.

End of EXTEND FREEZE

HOVER – Move through a file, zig-zag reading it at a given frequency

Usage

hover hover *infile outfile frq loc frqrand locrand splice dur*

Parameters

infile – input soundfile (mono)
outfile – resultant soundfile
frq – rate of reading source-samples (in Hz).

> *Frq* determines the width (in samples) of the zigzag-read; for example, at a sample-rate of 44100:
> - frq = 1 Hz: reads 22050 samples forward and 22050 samples back.
> - frq = 10 Hz: reads 2205 sampless forward and 2205 samples back.

loc – time in *infile* from which samples are read.
frqrand – degree of random variation of frequency (range 0-1).
locrand – degree of random variation of location (range 0-1).
splice – length of the splice (range: 0-100 milliseconds).

> *splice* length must be less than 1 over twice the maximum *frq* used, e.g. <5 ms for 100 Hz.

dur – total output duration.

> *frq* and *loc*, *frqrand* and *locrand* may vary through time.
> Time in any breakpoint files is time in the ***output*** file.

Understanding the HOVER Process

HOVER is a variant of **ZIGZAG**, but instead of jumping about in the file, it hovers around a given time-point (*loc*), reading forwards and backwards from this point at a given speed, which also determines the width of the reading. Note that the location point is time-variable, so the pointer can move through the file over time or indeed move to any time-point you wish. You can also randomly vary the frequency and the location point.

Musical Applications

HOVER gives considerable scope for prolonging a sound, by reading the file in a controlled zig-zag fashion. It might be used for extending short-lived percussive sounds of an inharmonic timbre; producing a series of ebb-and-flow shapes (each like **BAKTOBAK**); or prolonging a highly textured sound which is difficult to loop. A number of different HOVERings of the same sound mixed together should also produce an interesting texture out of the one source.

End of HOVER

EXTEND ITERATE – Repeat sound with subtle variations

Usage

extend iterate 1 *infile outfile outduration* [**-d***delay*] [**-r***rand*] [**-p***shift*] [**-a***ampcut*] [**-f***fade*] [**-g***gain*] [**-s***seed*]
OR:
extend iterate 2 *infile outfile repetitions* [**-d***delay*] [**-r***rand*] [**-p***shift*] [**-a***ampcut*] [**-f***fade*] [**-g***gain*] [**-s***seed*]

Modes

1 Iterate to a specified duration
2 Iterate a specified number of times

Parameters

infile – input soundfile
outfile – output soundfile
outduration – length in seconds of *outfile*
repetitions – number of repetitions in the iteration
-d*delay* – (average) delay between iterations in seconds (Default: length of *infile*)
-r*rand* – delay-time randomisation (Range: 0 to 1, Default: 0)
-p*pshift* – maximum random pitchshift of each iteration in semitones (Range: 0 to 12 semitones; e.g., 2.5 = 2.5 semitones up or down)
-a*ampcut* – maximum random amplitude reduction on each iteration (Range: 0 to 1, Default 0)
-f*fade* – (average) amplitude fade between iterations (Range: 0 to 1, Default 0)
-g*gain* – overall gain (Range: 0 to 1, Default: 0, which gives the best guess for no distortion)
-s*seed* – the same seed number will produce identical output on rerun (Default: 0 – the random sequence is different every time)

Understanding the EXTEND ITERATE Process

EXTEND ITERATE was written as a way of achieving more natural sounding iterations of a soundfile by introducing a randomisation of the delay time between each iterated segment, and slight variations in pitch or amplitude between the segments, as would occur in a naturally iterating source (e.g., a rolled 'rr' vocal sound). These randomisations can be selected (e.g., one might omit pitch variation, or not apply randomisation to the delay times), or applied in an exaggerated fashion, to achieve a number of different musical results.

The *rand* parameter introduces slight variations in *delay* between iterations, which may increase the 'naturalness' of the result. Omitting the *rand* parameter will produce a more mechanical echo effect.

The *gain* parameter allows some control over the amplitude of the mixed portions; the amount of *gain* suitable is dependent on the amplitude of the signal at the beginning and end of the soundfile (where the repeated units overlap). This can be examined with a soundfile viewer (such as VIEWSF, which can display the amplitude of each individual sample), and the *gain* adjusted accordingly if the defaults don't seem to be handling it properly. When randomisation is used, the *gain* is further reduced in the expectation that there will be a greater degree of overlap.

Musical Applications

This function produces a series of (usually overlapping) repeats of a soundfile. The nature of the attack portion of the soundfile – sharp or gradual – will greatly affect the way these repetitions are perceived. The use of a very short soundfile, e.g., 0.2 seconds, especially one with a sharp attack, will result in a rapid-fire succession of easily perceived iterations.

End of EXTEND ITERATE

ITERLINE – Iterate an input sound, following a transposition line

Usage

iterline iterline mode *insndfile outsndfile tdata outduration* [**-d***delay*] [**-r***rand*] [**-p***pshift*] [**-a***ampcut*] [**-g***gain*] [**-s***seed*] [**-n**]

Example command line to create transposed repetitions of a soundfile :

```
    iterline iterline 1 "tdata.txt" 10 1 0 0 0 0 0
```

Modes

1 Interpolate between transpositions (glissandi)
2 Step between transpositions (discrete pitch changes)

Parameters

insndfile – input soundfile
outsndfile – output soundfile
tdata – text file of *time transposition* pairs, with the transpositions given in (possibly factional) semitones
outduration – duration of the output soundfile
-d*delay* – the (average) delay betwen iterations
-r*rand* – randomisation of the delay time. Range: 0 to 1. Default: 0 (no randomisation)
-p*pshift* – the maximum value for the random pitch shift of each iteration. Range: 0 to 12 semitones. For example, a value of 2.5 means 2.5 semitones up or down.
-a*ampcut* – the maximum value for a random amplitude reduction on each iteration. Range: 0 to 1. Default 0 (no randomisation)
-g*gain* – overall gain. Range: 0 to 1. Note that **0** is a special value that produces the maximum acceptable level. This will be overridden by the **-n** normalisation flag – see below.
-s*seed* – the same seed-number will produce identical output on each rerun. Default: 0 (random sequence is different every time)
-n – normalise the output: the maximum output level is the same as the maximum input level. This normalised output will be greater than the input level only if *gain* is NON-zero.

Understanding the ITERLINE Process

EXTEND ITERATE repeats a sound over and over until *outduration* is reached, possibly with an amplitude reduction with each iteration. ITERLINE adds to this the facility to have these iterations follow a time-varying linear contour, either with glissandi or with discrete steps between the iterations.

End of ITERLINE

ITERLINEF – Iterate an input sound set, following a transposition line

Usage

iterlinef iterlinef mode *insndfile outsndfile tdata outduration* [**-d***delay*] [**-r***rand*] [**-p***pshift*] [**-a***ampcut*] [**-g***gain*] [**-s***seed*] [**-n**]

Example command line to create transposed repetitions of a soundfile :

```
    iterline iterline 1 "tdata.txt" 10
```

Modes

1 Interpolate between transpositions (glissandi)
2 Step between transpositions (discrete pitch changes)

Parameters

insndfile – input soundfile
outsndfile – output soundfile
tdata – text file of *time transposition* pairs, with the transpositions given in (possibly factional) semitones
outduration – duration of the output soundfile
-d*delay* – the (average) delay betwen iterations
-r*rand* – randomisation of the delay time. Range: 0 to 1. Default: 0 (no randomisation)
-p*pshift* – the maximum value for the random pitch shift of each iteration. Range: 0 to 12 semitones. For example, a value of 2.5 means 2.5 semitones up or down.
-a*ampcut* – the maximum value for a random amplitude reduction on each iteration. Range: 0 to 1. Default 0 (no randomisation)
-g*gain* – overall gain. Range: 0 to 1. Note that **0** is a special value that produces the maximum acceptable level. This will be overridden by the **-n** normalisation flag – see below.
-s*seed* – the same seed-number will produce identical output on each rerun. Default: 0 (random sequence is different every time)
-n – normalise the output: the maximum output level is the same as the maximum input level. This normalised output will be greater than the input level only if *gain* is NON-zero.

Understanding the ITERLINEF Process

Here we need to understand what an 'input sound set' is. It must consist of 25 transpositions of a source at intervals of one semitone, in ascending order. The input sounds must be of approximately equal duration.

End of ITERLINEF

EXTEND LOOP – Loop (repeat [advancing] segments) inside soundfile

Usage

extend loop 1 *infile outfile start len step* [**-w***splen*] [**-s***scat*] [**-b**]
extend loop 2 *infile outfile dur start len* [**-l***step*] [**-w***splen*] [**-s***scat*] [**-b**]
extend loop 3 *infile outfile cnt start len* [**-l***step*] [**-w***splen*] [**-s***scat*] [**-b**]

Modes

1 Segment advances in soundfile until soundfile is exhausted
2 Specify *outfile* duration (shortened if looping reaches end of *infile*)
3 Specify number of loop repeats (reduced if looping reaches end of *infile*)

Parameters

infile – soundfile to process
outfile – output soundfile
dur – duration of *outfile* required (in seconds)
cnt – number of loop repeats required
start – time (in seconds) in *infile* at which the looping process begins
len – length of looped segment (in milliseconds)
[**-l**]*step* – advance in *infile* from the start of one loop to the next (in milliseconds)

May be **zero** in Modes **2** and **3** but **not** in Mode **1**. When zero, repeating loops of the same material are created.

-w*splen* – length of splice (in milliseconds) (Default: 25ms)
-s*scat* – make *step* advance irregularly, within the timeframe given by *scat*
-b – play from beginning of *infile* (even if looping doesn't begin there)

Understanding the EXTEND LOOP Process

The key feature of this process is that it joins together, end-to-end, a series of segments taken from the *file*, each with a splice slope to avoid clicks. These segments are all of the same length, so one way or another, the result may appear to have some degree of regular pulsation. This does not (usually) result from the presence of splices, but rather is a perceptual result caused by the repetition of sonic material.

The most salient parameters are *step* and *len*. *Step* is the timestep in the sourcefile between the start of one selected segment and the next.

- If the *step* is zero, the selected segment will simply be repeated.
- If the *step* is > 0 but smaller than the segment length, the selected segments will share much in common, but each will begin at a point in the sourcefile a little later than the last, producing a progressing echo effect. The *outfile* will be longer than the *infile*.

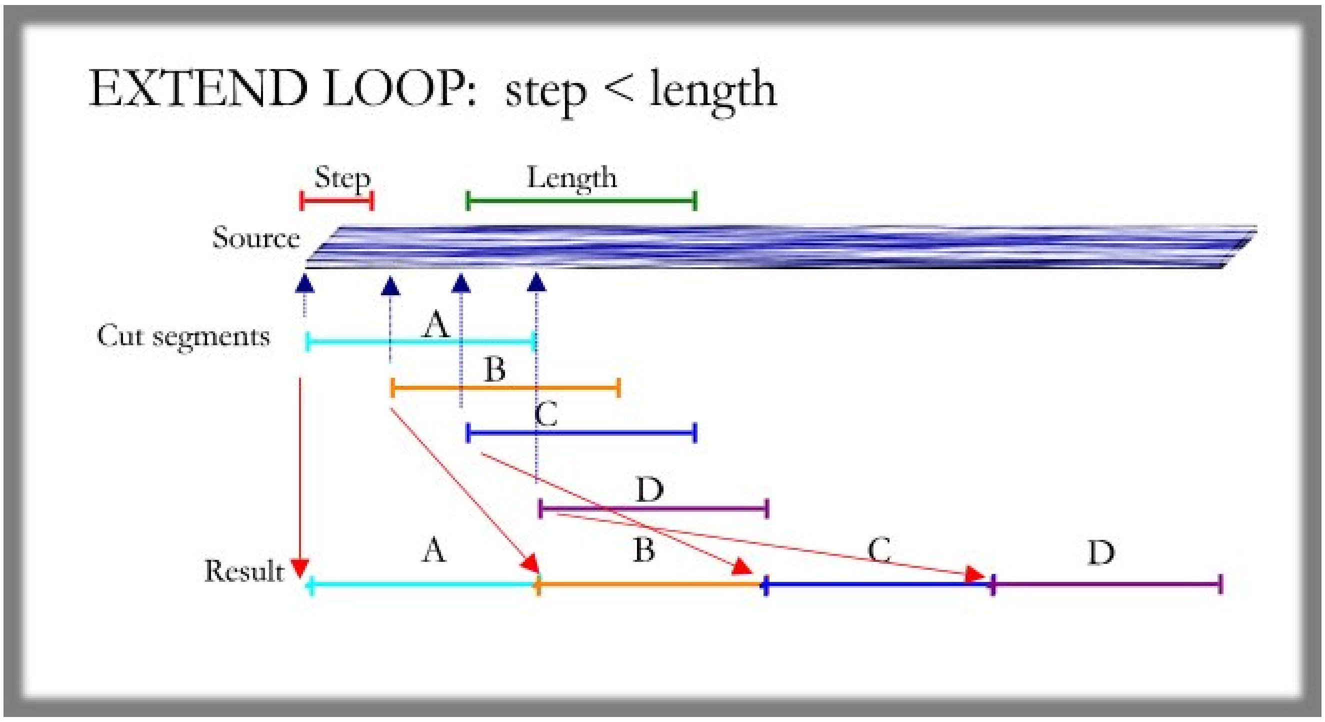


Diagram 1: shows how the segments are taken from overlapping locations in the source, due to the short steps

- If the *step* is larger than the segment length, the process will leap through the file, omitting bits of it, and rapidly get to the end. The result will be shorter than the *infile*.

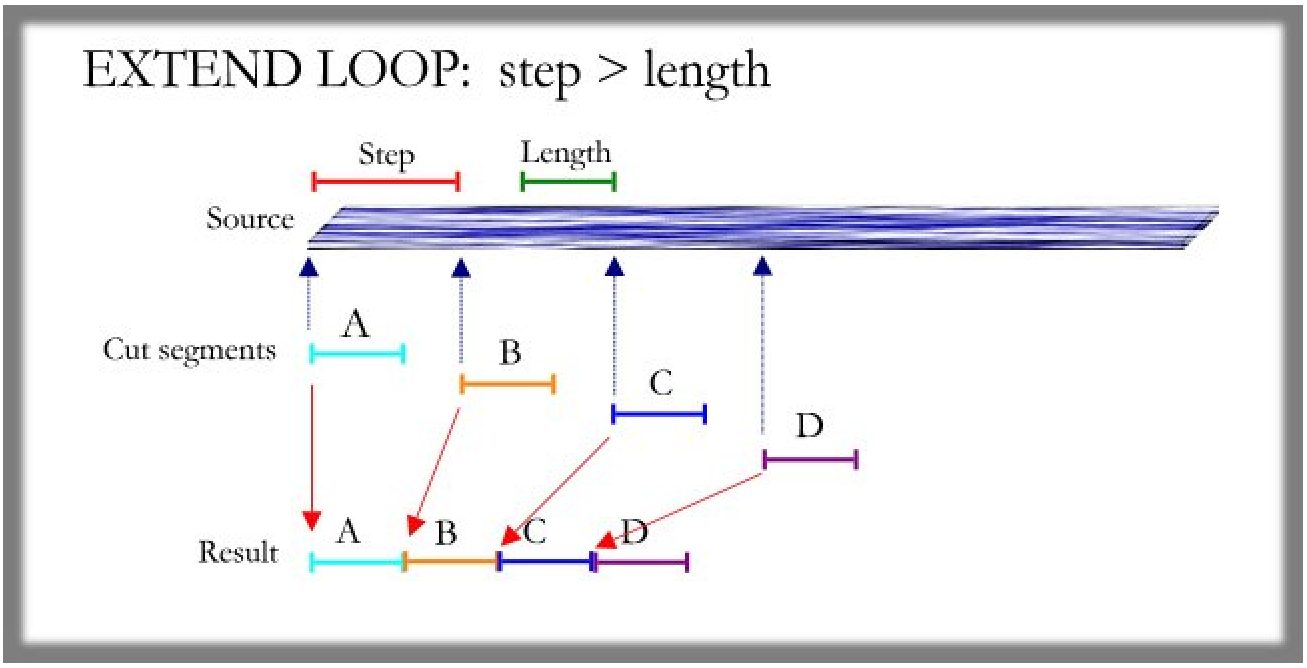


Diagram 2: shows how the segments are taken from the source after a gap, due to the long steps

The *scat* parameter randomises the length of *step* (within a small range), producing a less mechanical result.

The length of the segments (*len*) – as well as the size of the *step* – affects the recognisability factor of the original source material.

Musical Applications

EXTEND LOOP can be used for pulsating a sound in a regular way. The incremental movement through the source can be illustrated by moving through the word 'anchovies' with a *step* that moves ahead one letter at a time and a segment length which encompases 4 letters:

```
anch-ncho-chov-hovi-ovie-vies
```

Because of the limited nature of these facilities, EXTEND LOOP can be used to play with sounds in a controlled way. Some of the effects resulting from very small steps and segment lengths will be surprising and approach brassage techniques.

For (much) more flexibility in brassage and granulation, see **MODIFY BRASSAGE** and its graphic counterpart *GrainMill* (on PC sytems). Other functions which carry out **fragmentation** in some way include **EXTEND DRUNK**, **EXTEND SCRAMBLE** and **EXTEND ZIGZAG**.

In the spectral domain, also see **BLUR DRUNK**, **BLUR SHUFFLE**, and **BLUR WEAVE**. **COMBINE INTERLEAVE** fragments by interpolating *N* analysis windows from two different files, functions such as **FOCUS FREEZE** and **FOCUS STEP** fragment by holding material according to a time frame pattern. **MORPH BRIDGE** and **MORPH GLIDE** break up a file by working with specified analysis windows, and **SPEC GRAB** and **SPEC MAGNIFY** can isolate and extend a single analysis window. Finally, the whole **DISTORT** package fragments material by creating pseudo-wavecycles from sonic material found between zero crossings.

The question of 'recognisability' is relevant to these and other CDP processes which radically alter the source material. For further thoughts on this subject, see the Appendix: Recognisability and Sound Transformation.

End of EXTEND LOOP

MADRID – Spatially syncopate repetitions of the source soundfile(s)

Usage

madrid madrid 1 *insndfile1 [insndfile2 ...] outsndfile dur ochans strmcnt delfact step rand* [**-s***seed*] [**-l**] [**-e**] [**-r** | **-R**]
madrid madrid 2 *insndfile1 insndfile2 [insndfile3 ...] seqfile outsndfile dur ochans strmcnt delfact step rand* [**-s***seed*] [**-l**] [**-e**]

Example command line to create spatially separated repetitions:

```
madrid madrid 1 in1.wav in2.wav in3.wav out.wav 10 4 8 1 1.0 0
```

Modes

1 Random output file order
2 Use *segfile* to determine the order of output files

Parameters

insndfile – input soundfile or soundfiles (mono)
outsndfile – output (multi-channel) soundfile
segfile – textfile containing a list of numbers in the range 1 to count-of-infiles which determines the sequence in which the infiles are used in the output
dur – duration of the output sound
ochans – number of channels in the output sound (Range 2 to 16)
strmcnt – number of spatially distinct streams (2 to 64)
delfact – proportion of items to (randomly) delete. Values between 0 and 1 delete that proportion of events in the various streams. For values greater than 1, the proportion of events at a single location increases. (Range: 0 to 1000)
step – time between event repetitions (Range: 0 to 60 sec.)
rand – randomisation of *step* size (Range: 0 to 1)
-s*seed* – value to initialise the randomisation of the *delfact* deletions. With a non-zero value, rerunning the process with the **same** parameters will produce the same output. Otherwise, the deletions are always different.
-l – for *ochans* > 2, the loudspeaker array is assumed to be circular. The **-l** flag forces the array to be linear, with defined left and right ends.
-e – allow **empty** events: i.e., sound is absent at some of the repeat-steps.
-r – randomly permutate the order of input sounds used in the output. (ALL input sounds are used ONCE before the next order permutation is generated.)
-R – randomly select the next input sound: the selection is unrelated to the previous selection.

-r and **-R** cannot be used in combination. When only one input sound is used, neither flag has any effect.

delfact, *step* and *rand* can vary over time.

Understanding the MADRID Process

MADRID achieves its syncopated repetitions by randomly deleting items from the spatially-separated repetition streams. The program sets up several sound streams. In each sound stream the same source sounds are repeated at the same time-interval. By randomly deleting repetitions from the various streams, the output appears to be spatially syncopated, as stress is transferred from one stream to another, or to some combination of streams, changing the apparent spatial location of the source.

End of MADRID

EXTEND REPETITIONS – Repeat source at given times

Usage

extend repetitions *infile outfile timesfile level*

Parameters

infile – input soundfile
outfile – output soundfile
timesfile – Textfile of times (in seconds) at which the source plays.
level – Level of output. Range: 0 to 1.

Level may vary over time.

Understanding the EXTEND REPETITIONS Process

This program can be thought of either as a more controlled looping function or a simple rhythm sequencer.

Musical Applications

Controlled looping or rhythmic sequencing.

End of EXTEND REPETITIONS

EXTEND SCRAMBLE – Scramble soundfile and write to any given length

Usage

extend scramble 1 *infile outfile minseglen maxseglen outdur* [**-w***splen*] [**-s***seed*] [**-b**] [**-e**]
extend scramble 2 *infile outfile seglen scatter outdur* [**-w***splen*] [**-s***seed*] [**-b**] [**-e**]

Modes

1 Cut random chunks from *infile* and splice end to end
2 Cut *infile* into random chunks and rearrange; repeat differently, etc.

Parameters

infile – input soundfile to process
outfile – output soundfile
minseglen – minimum chunksize to cut
maxseglen – maximum chunksize to cut (Range: 0.045 to length of *infile* – must be > *minseglen*)
seglen – average chunksize to cut
scatter – randomisation of chunk lengths (>= 0)

Cannot be greater than infilesize/*seglen* (rounded **down**)

outdur – duration of *outfile* required (> *maxseglen*
-w*splen* – duration of splice in milliseconds Default: 25ms)
-s*seed* – the same seed number will produce identical output on rerun (Default: 0, random sequence is different every time)
-b – force start of *outfile* to be beginning of *infile*
-e – force end of *outfile* to be end of *infile*

Understanding the EXTEND SCRAMBLE Process

With EXTEND SCRAMBLE, segments of soundfile are selected from a wide variety of locations in the *infile*, jumping back and forth a great deal.

Mode **1** takes the *infile*, chooses a random chunk of it, and then chooses another random chunk of it **which may overlap with the first choice**, then another chunk **which may overlap with either of the other two** ... etc. Then it splices them all together. Thus, any bits of the file may be repeated quite quickly if overlapping material is selected in consecutive chunks, and some bits may not appear at all if never randomly selected.

The size of the chunks will be a random length somewhere between *minseglen* and *maxseglen*.

Mode **2** cuts the entire file into random-length chunks which do **not** overlap. It arranges these at random. The process is then repeated, but the random cuts are of course in different positions in the file. Consequently, the **entire** file is used, and used **only once**, before the process starts to use the file again.

In Mode **2** an average chunksize is specified plus a random factor (*scatter*). The formula which shows what the maximum *scatter* factor can be reveals that Mode **2** can be used to make chunks which vary a great deal in length. For example, if the *infile* is 2 seconds long and *seglen* is 0.3, the maximum value for *scatter* will be 6.0 (rounded down). (This value was accepted – and worked – even with an *outdur* of 4.0.)

The 2nd Mode also provides the option to rerun with identical output.

The ability to write to any length of *outfile* makes it possible to give the process plenty of time to make full use of the *infile*.

Musical Applications

EXTEND SCRAMBLE provides a relatively automated way to fragment a soundfile in a random way, tending to swing back and forth from the beginning and end portions of the *infile*. The jumping about is likely to be extreme, so applied to vocal material, the results will be somewhere between wild and funny. Applied to pitched material, the result can sound like an improvisation.

For a more carefully defined zigzagging motion through a soundfile, see **EXTEND ZIGZAG**.

End of EXTEND SCRAMBLE

EXTEND SEQUENCE – Produce a sequence from an input sound played at specified transpositions and times

Usage

extend sequence *infile outfile sequence-file attenuation*

Parameters

infile – input soundfile
outfile – output soundfile, being a sequence made from the (one) input soundfile, according to the instructions in the *sequence-file*.
sequence-file – contains 3 values on each line, separated by tabs or spaces, one line per event: *output-time semitone-transposition loudness* value triples, where *loudness* is a loudness multiplier. There needs to be one value-triple for each event in the sequence.
attenuation – overall attenuation to apply to the source, should *outfile* overload

Understanding the EXTEND SEQUENCE Process

This program works like a simple conventional sequencer except that it takes only one input (see EXTEND SEQUENCE2 for multiple soundfile input. The process takes a *sequence-file* of triple-values:

1. *output-time* – the time when you want the soundfile to come in again in the *outfile*
2. *transposition* – the pitch-level of that entry, given in (possibly fractional) semitones
3. *amplitude* – the relative level of that sound in the output sequence, louder (> 1.0) or softer (< 1.0)

for each event in the sequence. The source sound is then copied at each *output-time*, transposed by each *transposition*) amount in (fractional) semitones, and attenuated to the *level* specified. The result is a sequence of events derived from the one source sample.

An example *sequence-file*:

```
[time transp loudness]
0.0   0.0    0.25
1.5   3.25   0.50
3.0   7.75   1.00
```

Musical Applications

This function was used, for example, to make the underlying sequence of the 'Gamelan' in Trevor Wishart's compositon, Imago. Transposition or time sequences might be derived from data from other sounds, generated in the *Sound Loom* Table Editor or with COLUMNS, or entered by hand in a text file. EXTEND SEQUENCE is therefore a useful way to create rhythmic textures, whether simple or very intricate.

End of EXTEND SEQUENCE

EXTEND SEQUENCE2 – Produce a sequence from several sounds played at specified transpositions and times

Usage

extend sequence2 *infile1 infile2 [infile3 ...] outfile sequence-file attenuation*

Parameters

infile1 – input soundfile
infile2 – 2nd required input soundfile
[*infile3*] – optional 3rd or more additional input soundfiles. **All input files must have the same number of channels.**
outfile – resultant output sequence of soundfiles
sequence-file – data file in which the first line contains notional MIDI pitch values for each input soundfile and each subsequent line contains 5 values on each line, given in order from left to right, separated by spaces:

1. *input-sound-number* – these numbers follow the order in which they are given as inputs
2. *output-time* – time in the *outfile* when this event is to begin
3. *MIDI pitch* – MIDI pitch level at which to perform the sound, relative to the notional pitch given in Line 1 – it may be fractional, i.e., microtonal, such as 60.5, which is ½ semitone (50 cents) higher than Middle C.
4. *loudness* – relative loudness multiplier for that event. Range: 0 to 1.
5. *duration* – a duration for that event: it can curtail it (i.e., truncate the source sound), but cannot extend it. Note that transposition needs to be taken into account: the maximum event length = the (transposed) duration of the sound chosen. (The transposed length will be the source length * the transposition ratio – the Music Calculator converts between transposition in (fractional) MIDI pitches and transposition in ratios.)

attenuation – overall attenuation to apply to the source, should *outfile* overload

Understanding the EXTEND SEQUENCE2 Process

An example *sequence-file* serves as a reminder of how the data is put together:

```
60 60
[Snds Stime MPV Level Dur]
1     0.0   60  0.25  1
2     0.5   62  0.50  2
1     1.5   63  0.25  1.5
2     2.5   64  0.50  2
```

Here we have a convenient way to arrange several different files in a rhythmic way, with several additional parameter settings.

- There are two input sounds. The first one given to the program will be No. 1 and the second one will be No. 2.
- The *Start times* set the times at which they begin to play.
- The MPVs (MIDI Pitch Values) specify the pitch level relative to the notional MPVs given in the first line, one for each file, which may or may not be the actual pitch level of the sound.
- Then the *amplitude level* is specified, so that certain sounds can be emphasised, different original amplitudes readjusted, etc.
- Finally, the duration of the note event is given. It may be shorter than the original sound, but, of course, not longer. This provides an easy way to work with the attack portion of a sound, or to layer longer sounds.

Musical Applications

We are familiar with standard **MIX files**. They specify the sound by using its name. Here we specify the sound by using a number. This means that it is easy to list and order the sounds, as well as to use a numerical pattern generated in some other way, such as algorithmically. Also different from the standard MIX files are the pitch transposition and duration fields.

If the sound is fairly clearly pitched, such as a bell sound, then EXTEND SEQUENCE2 enables us quickly to:

- Create a melody
- Create a lively rhythmic pattern, with the same or different pitches
- Pattern the loudness in a sequence of sounds
- Use the attack of the sound we want while discarding the remainder
- Create some 'changes' as used by bell-ringers – or any other intricate sequence of notes

Note that the same sound can be repeated. The functionality of the program is shown by combining its use with the table editing software in the CDP System. Thus we can massage the columnar data with the **Table Editor** in *Sound Loom* (= **DATA, Columns** in *Soundshaper* or just **columns** on the Command Line). Several different versions could be made and then each one realised with EXTEND SEQUENCE2 by loading in the various *sequence-files* in turn. For example, a structural *ritardando* could be made by adding a value to the *start_time* column and subtracting a value from the *pitch_level* column 3 or 4 times. The result is a series of output soundfiles in which several sounds repeat, placed further and further apart in time, while getting closer together in pitch.

End of EXTEND SEQUENCE2

SHIFTER – Generate simultaneous repetition streams, shifting rhythmic pulse from one to another

Usage

shifter shifter 1 *infile outfile cycles cycdur ochans subdiv linger transit boost* [**-z | -r**] [**-l**]
shifter shifter 2 *infile1 infile2 [infile3 ...] outfile cycles cycdur ochans subdiv linger transit boost* [**-z | -r**] [**-l**]

Example command line to create a shifting rhythmic pulse among repeating simultaneous streams:

```
shifter shifter 1 in.wav out.wav "cycles.txt" 1 10 4 4 0 1 1
```

Modes

1 Use the same input sound for all cycles
2 The number of input files must equal the number of cycles. The program assigns the input files, in order, to the cycles, in order.

Parameters

infile – input soundfile or, Mode **2**, soundfiles (mono)
outfile – output (multi-channel) soundfile
cycles – a textfile listing the number of beats in each cycle.
cycdur – the duration of one complete cycle
dur – the required duration of the output sound
ochans – the number of channels in the output soundfile (mono or multi-channel, Range: 2 to 16)
subdiv – the minimum division of the beat: it needs to be > 4 and a multiple of 2 and/or 3
linger – the number of cycles that are to remain in a fixed focus
transit – the number of cycles that are to make a transition to the next focus. The sum of *linger* and *transit* must be >= 1.
boost – with standard stream level "L", add `boost * L` to focus stream level.
-z – This flag causes focus to ZIGZAG through the cycles. For examples, with cycles 11, 12, 13, focus moves like this: 11, 12, 13, 12, 11, 12, 13, 12 etc.
-r – This flag causes focus to select a RANDOM order of the cycles. For example, cycles 12, 11 & 13 move through those, then another random order is selected, etc.
-l – If the number of output channels is greater than 2, the loudspeaker layout is assumed to be surround-sound. The **-l** flag changes the loudspeaker arrangement to a linear array, with a leftmost and rightmost loudspeaker.

Understanding the SHIFTER Process

SHIFTER sets up several sound streams. In each sound stream the source sound repeats at a fixed tempo in (specified) cycles. The repetition-times for each cycle are arranged so that the streams will resynchronise (all start at the same instant) after a specified number of cycles in each stream. For example, with cycles 11,12,13, three streams are set up which repeat the sound 11,12 and 13 times, respectively, before the streams resynchronise. (This represents three streams with their tempi in the relationship 11:12:13.)

Note that the sounds themselves are NOT time-stretched; only the timings-between-repetitions are different in the different cycles. The various "focusing" parameters determine which of the simultaneous tempi is the most prominent at any time.

Care should be taken to keep the input level very low, to avoid overflow as sounds are mixed in the SHIFTER process. Modify loudness 4 (Force level) might be set to e.g. 0.05.

End of SHIFTER

SHRINK – Repeat a sound, shortening it on each repetition

Usage

shrink shrink 1-3 *infile outfile shrinkage gap contract dur spl* [**-s***small*] [**-m***min*] [**-r***rnd*] **-n -i**
shrink shrink 4 *infile outfile time shrinkage gap contract dur spl* [**-s***small*] [**-m***min*] [**-r***rnd*] **-n -i**
shrink shrink 5 *infile generic-outfile-name shrinkage wsiz contract aft spl* [**-s***small*] [**-m***min*] [**-r***rnd*] [**-l***len*] [**-g***gate*] [**-q***skew*] **-n -i -e -o**
shrink shrink 6 *infile generic-outfile-name peaktimes shrinkage wsiz contract aft spl* [**-s***small*] [**-m***min*] [**-l***len*] [**-g***gate*] [**-r***rnd*] **-n -i -e -o**

Example command line to create compressing sound events:

```
shrink shrink 1 inf.wav outf.wav 0.25 5 0.8 30 15
```

Modes

1 Shrink from the end
2 Shrink around the midpoint
3 Shrink from the start
4 Shrink around a specified time
5 Shrink around found peaks and output each segment as a separate soundfile, also creating a mixfile with which to assemble them
6 Shrink around specified peaks and output each segment as a separate soundfile, also creating a mixfile with which to assemble them

Parameters

infile – input soundfile
outfile – output soundfile
generic-outfile-name – Modes **5** & **6**: rootname for several soundfile outputs; a numeral is appended to the rootname
shrinkage – shortening factor of sound from one repeat to the next. Shrinkage stops once events become too short for splices.
time (Mode **4**) – time around which shrinkage takes place
gap – initial timestep in seconds between output events (Range: 4.899932 to 60.0)
peaktimes (Mode **6**) – a textfile list of the times where peaks occur in the input soundfile
contract – shortening of gaps between output events: **1.0** = events are equally spaced, **< 1.0** = events become closer together. **Events cannot overlap, so the minimum contraction is the maximum shrinkage**.
dur – the (minimum) duration of the output
aft (Modes **5-6**) – time after which the shrinkage begins
spl – splice length in milliseconds
wsiz (Modes **5-6**)– windowsize in milliseconds for extracting the envelope (Range: 1 to 100, Default: 100)
-s*small* – the minimum sound length, after which sounds are of equal length
-m*min* – the minimum event separation, after which events are regular in time
-r*rnd* – randomisation of timings **after** which events are regular in time

-l*len* – the minimum segment length before sound squeezing can begin; used with the **-e** flag
-g*gate* – the level relative to max below which found peaks are ignored (Range: 0 to 1, Default: 0)
-q*skew* – how the envelope is centred on the segment (Range: 0 to 1, Default 0.25; 0.5 = central position and a zero value switches the flag off.)
-n – equalise the maximum level of output events (if possible)
-i – Inverse: reverse each segment in the output. Note that then reversing the outfile creates a stream of unreversed segments where segments expande/accelerate rather than shrink/contract.
-e (Modes **5-6**) – Even Squeeze: sounds shorten in a regular manner starting with the first squeezed segment. Note that squeezed sound lengths are not dependent on the length of the input segments.
-o (Modes **5-6**) – omit any events that are too quiet once a fixed end tempo has been reached

Understanding the SHRINK Process

With SHRINK a sound is repeated, and at each repetition it gets shorter in duration because some of it is removed, and the gap between repetitions shrinks. In Modes **5** & **6** the events in the sound are also isolated. The range for the *gap* between repetitions starts at a relatively high value (4.899932 sec.) to allow scope for the time-compression of start times of the repetitions.

Musical Applications

The example command line above used an input file that was 4.9 seconds in duration. An output duration of 30 sec. was specified, giving SHRINK plenty of time to create its repetitions of ever decreasing size as well as increasingly closer together. At the end of the resultant soundfile, it was like something bouncing or rotating very rapidly just before coming to a halt. SHRINK allows similar effects to be achieved in an automated way, another example of the semi-algorithmic nature of many of the CDP programs.

End of SHRINK

EXTEND ZIGZAG – Read soundfile backwards and forwards, as you specify

Usage

extend zigzag 1 *infile outfile start end dur minzig* [**-s***splicelen*] [**-m***maxzig*] [**-r***seed*]
OR:
extend zigzag 2 *infile outfile timefile* [**-s***splicelen*]

Modes

1 Random zigzags: start and end at beginning and end of *infile*
2 Zigzagging follows times supplied by the user

Parameters

infile – input soundfile to process
outfile – output soundfile
start – together with ...
end – define the time interval in which the zigzagging occurs
dur – total duration of output sound required
minzig – minimum acceptable time between successive zigzag timepoints
-s*splicelen* – length of splice slope in milliseconds (Default: 25ms)
-m*maxzig* – maximum acceptable time between successive zigzag timepoints
-r*seed* – number to generate a replicable random sequence (> 0 – Default: 0, random sequence is different every time)

Entering the same number on the next program run generates the same sequence.

timefile – text file containing sequence of times to zigzag between

Each step-between-times must be **>** (3 * *splicelen*). **NB:** Zigsteps moving in the same (time-)direction will be concatenated.

Understanding the EXTEND ZIGZAG Process

What is special about this process is that it actually reads the source soundfile backwards when it moves from a later to an earlier timepoint.

It is expected that a user *timefile* will normally alternate between earlier and later times. Moving in the same direction is a bit pointless. If two steps move in the same direction, a **Warning** message is generated, on the assumption that this was an accidental entry by the user, e.g., a time was omitted. However, it still produces an output.

The *timefile* is written as a series of times in seconds, either horizontally with spaces between the times, or with each time on a separate line.

Musical Applications

Because moving from later to earlier times reads the source backwards, the effect of the zigzag transforms the sonic material much more than with EXTEND SCRAMBLE. The degree to which this happens depends on the duration between earlier and later times when moving from earlier to later, i.e., how much of the source is actually read forwards.

EXTEND ZIGZAG is a good way to begin to transmute sonic material into something more abstract. It can also be handled in such a way as to create a powerful warping effect: by making the forwards and backwards reads over fairly long durations. Short backwards and forwards movements create a stuttering or repeated note effect. (See diagram below, by Louisa Yong).

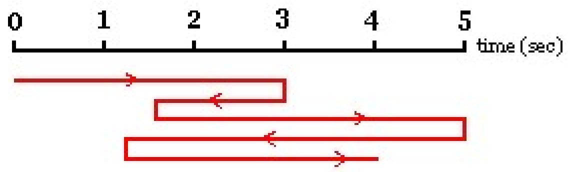


Diagram of a Zigzag

Also see: **MCHZIG**: multichannel version of ZIGZAG.

End of EXTEND ZIGZAG

Last Updated 11 Sep 2016 -- HTML5 version
Documentation: Archer Endrich, revised Robert Fraser