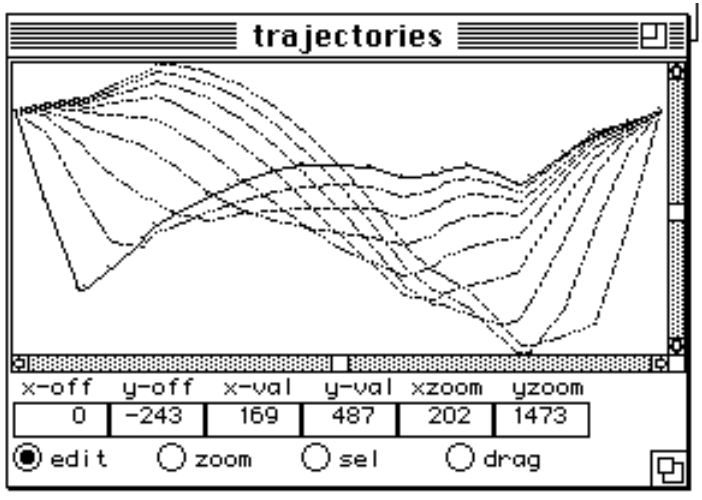
PatchWork

GenUtils

Continuous Functions Library

Version 1.0



First English Edition, April 1996

IRCAM Centre Georges Pompidou

This manual was written by Hans-Peter Stubbe-Teglbjaerg, and was produced under the editorial responsibility of Marc Battier, Marketing Office, Ircam.

PatchWork was conceived and programmed by
Mikael Laurson, Camilo Rueda, and Jacques Duthen.

The GenUtils library was conceived and programmed by Hans-Peter Stubbe-Teglbjaerg.

First edition of the documentation, April 1996.
This documentation corresponds to version 1.0 of the library, and to version 2.5 or higher of PatchWork.

Ircam
1, place Igor-Stravinsky
F-75004 Paris
Tel. (33) (1) 44 78 49 62
Fax (33) (1) 42 77 29 47
E-mail ircam-doc@ircam.fr

IRCAM Users' group

The use of this software and its documentation is restricted to members of the Ircam software users' group. For any supplementary information, contact:

Département de la Valorisation
Ircam
Place Stravinsky, F-75004 Paris

Tel. (1) 44 78 49 62
Fax (1) 42 77 29 47
E-mail: bousac@ircam.fr

Send comments or suggestions to the editor:
E-mail: bam@ircam.fr
Mail: Marc Battier,
Ircam, Département de la Valorisation
Place Stravinsky, F-75004 Paris

To see the table of contents of this manual, click on the Bookmark Button located in the Viewing section of the Adobe Acrobat Reader toolbar.

Contents

Résumé

La librairie GenUtils offre à PatchWork un ensemble de ressources pour la manipulation de fonctions continues. Elle permet de créer des fonctions de contrôle qui évoluent dans une dimension (le temps, par exemple). En ce sens, elle se rapproche conceptuellement des fonctions classiques des compilateurs acoustiques, telles que les Gen de Csound ou de Music V. Elle est aussi dérivée de la notion de commande à variation lente, comme la fonction LFO des synthétiseurs, et enfin de procédures graphiques comme le module **BPF** de PatchWork.

A l'aide de GenUtils, PatchWork permet, de façon sophistiquée, l'évolution de paramètres utilisés pour la synthèse, comme des enveloppes ou des transitions, et des fonctions de transformation pour les variations de tempo et les interpolations.

Le manuel présente d'abord les fonctions, qui sont regroupées en quatre catégories :

- CURVES : fonctions polynomiales connues sous le nom de « splines ».
- SHAPES : fonctions trigonométriques.
- xBPFs : extensions aux fonctions par segment de PatchWork.
- uTOOLs : outils pour la manipulation des fonctions ci-dessus.

A la suite de cette section de référence, quatre sections décrivent en détail la théorie et la mise en œuvre des fonctions.

1 Overview

The GenUtils Library is based on the idea of sampling of continuous functions. It provides a set of tools to create *control functions* that evolve in one dimension, such as time.It is largely based on many classical signal processing-ideas, reminiscent of the gen-routines in Csound or Music V, the generalized idea of low frequency oscillators and function drawing procedures as the **BPF** module in PatchWork. It seeks to allow elaborate control of the evolution of synthesis- parameters, such as envelopes or transitions, and of functional transforms in the domain of Computer-Assisted Composition (CAC), such as tempo-variations or interpolations. The Library is thus not an end in itself, not an application, but an aid to apply ideas.

A good working knowledge of PatchWork is assumed, as GenUtils is intended for those who urge for tools to shape dynamic evolutions, as it could be applied in spectral, temporal, gestural or any other animation-domain. Newcomers to signal-processing might also find it useful for pedagogical reasons.

The library shelves some 50 functions grouped as modules into 4 submenus, described as:

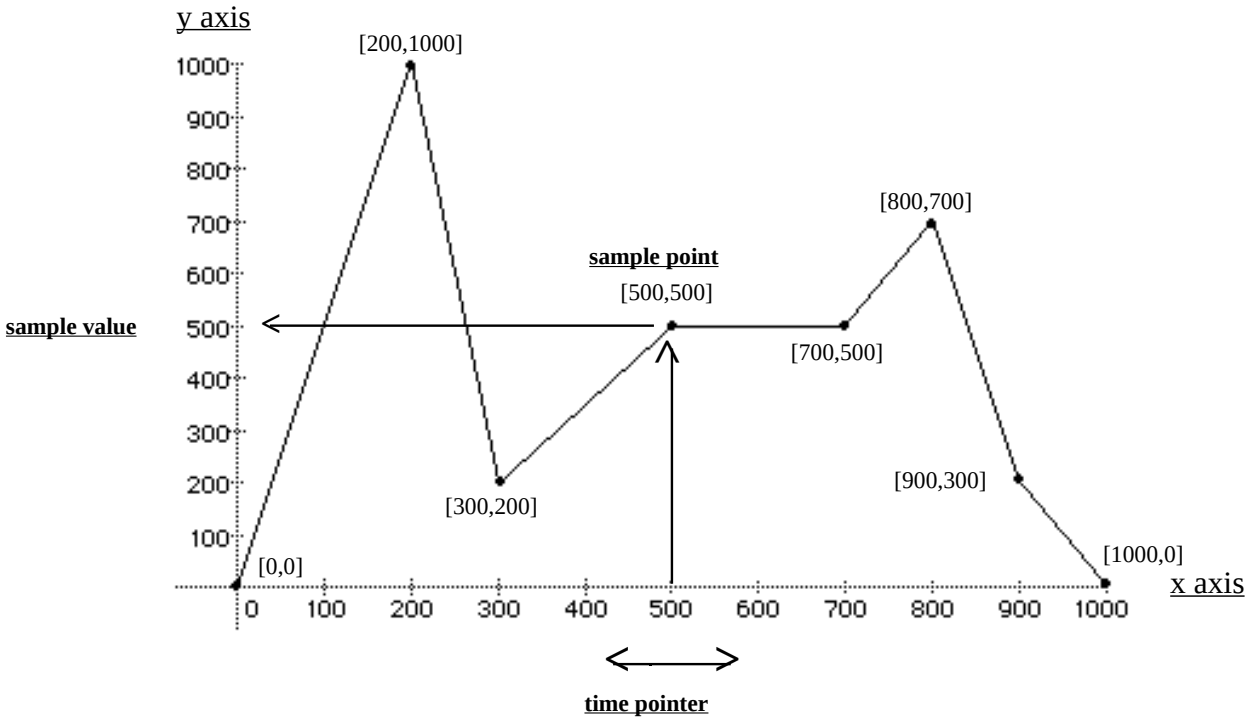
- CURVES: a set of polynomic functions known as "splines".
- SHAPES: a set of trigonometric functions, standard and non-standard ones.
- xBPFs : a set of extensions to the break point function in PatchWork.
- uTOOLs: a set of tools to operate on the functions above.

The choice of PatchWork as a software platform stems mainly from the modularity of structure of the underlaying Lisp language, which - virtual as it is - allows for a free experimentation of interconnections of the above functions, and from the aid of graphical programming existing in PatchWork, which - as its main feature - is found helpfull in organising complex ideas. Consequently the notion of data-types, like numbers, lists, objects is made as transperant to the user as possible, all major operations being applicable to all types.

For further theoretical reading please refer to the reference texts listed at the end of this manual.

2 Sampling a time function

To benefit the most from the **GenUtils** library one should understand how a function can be"sampled. In Patch-Work there is a module called **BPF**, which stands for "break-point-function" and is usually visualised in diagram-form. This is typically a two-dimensional coordinate system, where the horizontal line correspond to the *"x-axis"* and the vertical line to the *"y-axis"*. A *"point"* in this system is always represented by two values, it's *x*- and *y*-value, and is often written in square brackets with the *x*-value first and the *y*-value second. A serie of points can be inserted and you can draw a line between them to "see" how they look like. This is equavalent of saying that you "draw a function". A **BPF** is such a collection of points, and its name tells you that the line "breaks" at each new "point" where it migth turn then for a new direction. In the diagram you see the visual representation of a collection of points for a typical envelope.



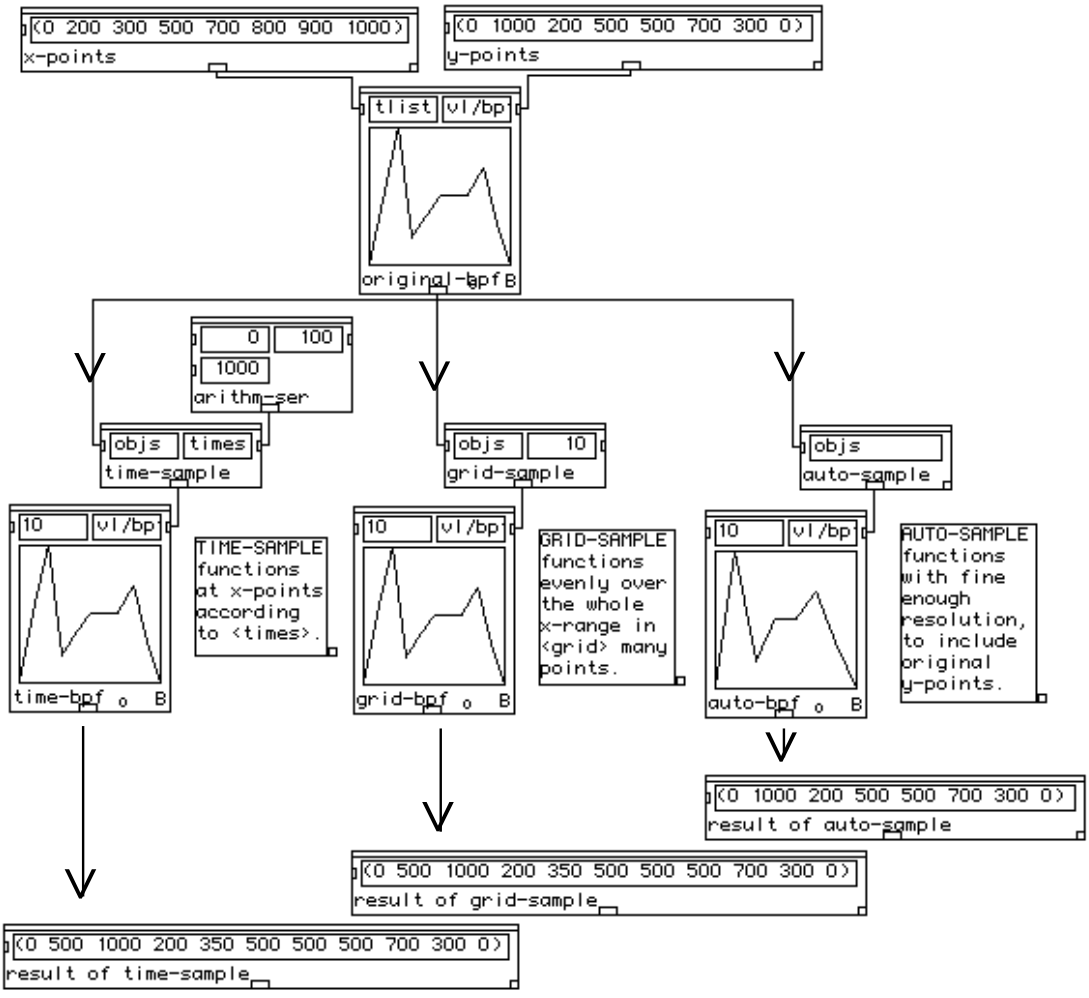
To understand what a function is you normally think of something being a function of something else, or more precise of "*y* as a function of *x*". This means, if you know *x* you migth try to find its corresponding *y* according to some mathematical expression. Or you could also return to the diagram and position your finger somewhere along the *x-axis*, and draw a straigth line upwards from there until you meet the line of the envelope. Draw from here another straigth line to the left to "read" the *y*-value from the *y-axis*. You have just *"sampled"* a value of a function. If the diagram represent an envelope with the *x-axis* associated with time and the *y-axis* with amplitude, you could say that you have "sampled the amplitude as a function of time". Indeed the *x-axis* very often is meant

to represent "time" in the **GenUtils** library, and therefore your finger becomes a "time-pointer" to the *x-axis.* The resulting *y*-value of pointing in this manner is called the "sample-value". And both of the *x*- and *y*-value together a "sample-point".

So the user specifies a number of sample-points and uses then a sliding time-pointer to sample a number of values. You can position the time-pointer anywhere you want along the x-axis. If you happend to be "in-between" two sample-points, say *x* equal to 100, the sample-value will be the the *y*-value at exactly that point. So there you see one of the reasonings behind the **BPF**; you can, with the specification of a relatively small number of sample-points, get access to a wide range of "in-between" values, depending on how fine the time-pointer is sliding. Is is normal to "slide" with some interval, because it would otherwise have to be infinite small steps, giving rise to infinite many values. Note also the "direction" of sliding the time-pointer. Most commonly is to slide from the very beginning to the end at a consant speed, as in synthesis, where the speed is given by the sampling rate. But in the virtual world of Patchwork there is nothing which prevent us from sliding backwards or progres-sing in circels or in a fractal pattern. You migth also use two diagrams, one that you slide normally to get values of how the time-pointer should move in the the second diagram. The first diagram is then called a transfer-map.

Important is it to undertstand that "sampling of functions in time" is a way to simulate and work with so called continuous functions in the discrete world of computers.

In the **GenUtils** library there exist three methods for sampling functions. They vary in degree of possible interaction from user-responsible to semi-automatic to full-automatic sampling. Here is a patch to demonstrate them by tour. Names in <....> marks user-changeable parameters.



The most general method and the base for all others is called **time-sample** (left). It is used to sample functions at *x*-points according to <*times*> as explaned earlier. The user is fully responsible of "moving" the time-pointer, whether it be evenly or unevenly or if it goes forwards or backwards. The only build-in help is when trying to sample outside of the *x*-range output will be forced to end-points. Its great advantage is that "assumption" or "knowledge" about how to sample are avoided, so that, using this module, any arbitrary sampling technique migth be employed. It is most often used for signal-sampling, but also for transfer-maps, granular or hopping techniques, or even for point-by-point decision-makings.

Next, **grid-sample** (middle) is a handy module to sample functions evenly over the whole *x*-range of the function. *<grid>* tells how many points one wants the hop interval being equal to (xmax-xmin)/grid. It is useful when one wants to obtain a fixed number of samples as when sampling trigonometric functions. Some problems of deformations migth occur when the number of steps are less than number of sample-points, especially when the points are unevenly distributed along the *x-axis.* Typically when one wants to sample a function so that originaly-points are included herein, one might encounter minor surprises. This means it is usually not well adapted as a reduction-method, whilst is is perfect as an explosion or fine-zoom method. In contrast to **time-sample** the module can be considered a semi-automatic sampler.

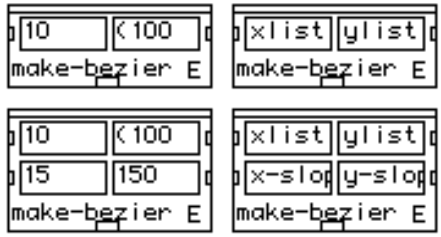
Finally, **auto-sample** (right) is a module to sample functions according to distributions of *x*-points, so that original *y*-points will be included in the result. It does so by making a grid of even intervals, that has a fine enough resolution to touch on original *x*-points. The grid is calculated with an interval equal to the greatest-common-divisor of the *x*-points. One is certain to have at least all the original *y*-points back (meaning that the *x*-points were evenly distributed). For very scattered floating *x*-points it may take a while with number of sample-values likely to increase drasticly. In contrast to **time-sample** and **grid-sample** it is a full-automatic sampler. Though less used it is good for sampling of fractal functions.

3 GenUtils Library : Reference

Modules are listed in the order they are found in the menus. For on-line help select the module and type t. That will open a mini-tutorial with some additional explanations. For more applicative patches please refer to the tutorials situated in the folder of the GenUtils library.

Menu *curves*

make-bezier



Syntax

xlist ylist &Optional x-slope y-slope

Inputs

xlist	fix/fl/list	default-value 10
ylist	list	default-value (100 200)

Optional

x-slope	fix/fl/list	default-value 15
y-slope	fix/fl/list	default-value 150

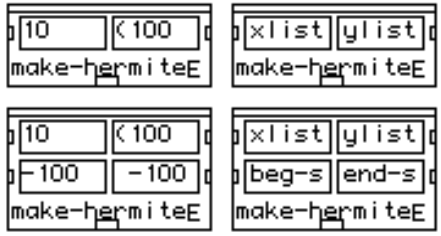
Output

c-curve-function

Explanation

Makes a Bezier function to be used by time-sample, grid-sample or auto-sample. Input *ylist* must be a list of 2 numbers describing start and end value in a transition. If *xlist* is given it should also be a list of 2 numbers. Optionally one can specify up to 2 control points given as separate lists of *x-slope* and *y-slope* towards which the curve is attracted.

make-hermite



Syntax

xlist ylist &Optional beg-slope end-slope

Inputs

xlist	fix/fl/list	default-value 10
ylist	list	default-value (100 200)

Optional

beg-slope	fix/float	default-value -100
end-slope	fix/float	default-value -100

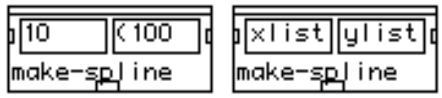
Output

c-curve-function

Explanation

Makes a Hermite function to be used by time-sample, grid-sample or auto-sample. Input *ylist* must be a list of 2 numbers describing start and end value in a transition. If *xlist* is given it should also be a list of 2 numbers. Optionally one can specify 2 slopes given as separate numbers of *beg-slope* and *end-slope* that tells how the curve departs or arrives.

make-spline



Syntax

xlist ylist

Inputs

xlist	fix/fl/list	default-value 10
ylist	list	default-value (100 200)

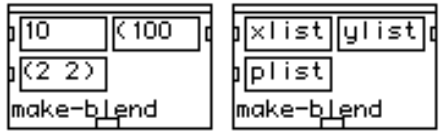
Output

c-curve-function

Explanation

Makes a spline function to be used by time-sample, grid-sample or auto-sample. If input is just a *ylist* then *xlist* is an interval (as with bpf), otherwise *xlist* and *ylist* should be of equal lengths.

make-blend



Syntax

xlist ylist plist

Inputs

xlist	fix/fl/list	default-value 10
ylist	list	default-value (100 200)
plist	list	default-value (2 2)

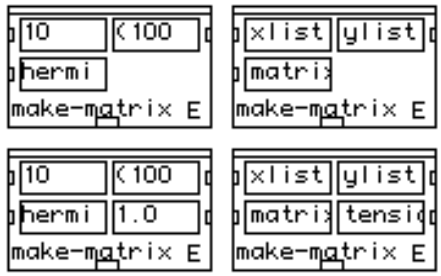
Output

c-curve-function

Explanation

Makes a blend function to be used by time-sample, grid-sample or auto-sample. If input is just a *ylist* then *xlist* is an interval (as with bpf), otherwise *xlist* and *ylist* should be of equal lengths. For the use of *plist* please refer to the tutorials.

make-matrix



Syntax

xlist ylist matrix &Optional tension

Inputs

xlist	fix/fl/list	default-value 10
ylist	list	default-value (100 200)
matrix	menu	(list of options see below)

Optional

tension	fix/float	default-value 1.0

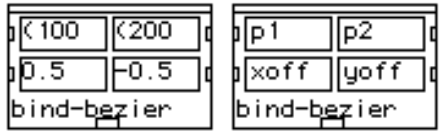
Output

c-curve-function

Explanation

Makes a Matrix function to be used by time-sample, grid-sample or auto-sample. If input is just a *ylist* then *xlist* is an interval as with bpf), otherwise *xlist* and *ylist* should be of equal lengths. *Tension* scales the *matrix* to allow for looser or tigther trajectories. For the use of menu *matrix* please refer to the tutorials.

bind-bezier



Syntax

p1 p2 xoff yoff

Inputs

p1	list	default-value (100 100)
p2	list	default-value (200 200)
xoff	fix/float	default-value 0.5
yoff	fix/float	default-value -0.5

Output

c-curve-function

Explanation

Makes a Bezier function with easy input of control-points. *P1* is an xy-coordinate of the origine, and *p2* that of the target. *Xoff* and *yoff* describes an xy-point relative to the linear transition between *p1* and *p2* towards which the curve will be attracted. Outputs an object containing *xlist* and *ylist*.

bind-hermite

Syntax

p1 p2 xoff yoff

Inputs

p1	list	default-value (100 100)
p2	list	default-value (200 200)
xoff	fix/float	default-value 0.5
yoff	fix/float	default-value -0.5

Output

c-curve-function

Explanation

Makes a Hermite function with easy input of control-points. *P1* is a xy-coordinate of the origine and *p2* that of the target. *Xoff* and *yoff* describes a xy-point relative to the linear transition between *p1* and *p2* towards which the curve will be attracted. Outputs an object containing only *ylist*.

bind-2points

Syntax

p1 p2 xoff yoff

Inputs

p1	list	default-value (100 100)
p2	list	default-value (200 200)
xoff	fix/float	default-value 0.1
yoff	fix/float	default-value 0.1

Output

list of 4 coordinate pairs

Explanation

From a fictive mid-point between 2 coordinate-pairs *p1* and *p2*, calculates 2 new points as relative offsets according to *xoff* and *yoff*. The mid-point is in the exact middle between *p1* and *p2*. Offsets *xoff* and *yoff* are given as relative values between 0 and 1. The function returns a list of 4 coordinate pairs and is used to insert extra points into a transition.

bind-4points

Syntax

p1 p2 xoff yoff width slope

Inputs

p1	list	default-value (100 100)
p2	list	default-value (200 200)
xoff	fix/float	default-value 0.5
yoff	fix/float	default-value 0.5
width	fix/float	default-value 0.1
slope	fix/float	default-value 0.1

Output

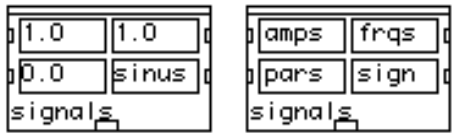
list of 6 coordinate-pairs

Explanation

Calculates, from 2 coordinate-pairs *p1* and *p2*, 4 new points with *xoff* and *yoff* describing a relative mid-point. Between *p1* and this mid-point, 2 new points are calculated according to *width* and *slope* given as relative values between 0 and 1. Likewise other 2 points are calculated from the same mid-point to *p2*. The function returns a list of 6 coordinate-pairs, and is used to insert extra points into a transition.

Menu *shapes*

signals



Syntax

amps frqs pars sign

Inputs

amps	fix/fl/list	default-value 1.0
frqs	fix/fl/list	default-value 1.0
pars	fix/fl/list	default-value 0.0
sign	menu	(list of options see below)

Output

c-signal-function

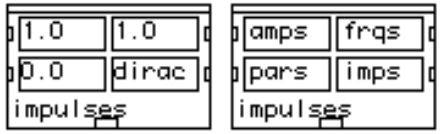
Explanation

Signal control module that generates a signal function according to choice of menu *sign*. Set *amps* to control magnitude, *frqs* for number of periods and *pars* for phase. Note that *pars* for menu-choice *slope* and *power* controls exponential slopes. Ouput is a signal object and is normally fed to one of the time modules for sampling. For that use please refer to the online help and tutorials.

The signal-types available in *menu* and the value ranges for *amps*, *frqs* and best *pars* are:

sinus	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
cosinus	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
power	= [-inf, +inf],	[-inf, +inf],	[-inf, +inf] (slope:1.0)
slope	= [-inf, +inf],	[-inf, +inf],	[-inf, +inf] (slope:1.0)
phasor	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
triangle	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
square	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
impulse	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (decay:1.0)
window	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (edges:0.5)

impulses



Syntax

amps frqs pars imps

Inputs

amps	fix/fl/list	default-value 1.0
frqs	fix/fl/list	default-value 1.0
pars	fix/fl/list	default-value 0.0
imps	menu	(list of options see below)

Output

c-signal-function

Explanation

Impulse control module that generates an impulse function according to choice of menu *imps*. Set *amps* to control magnitude, *frqs* for number of periods and *pars* for phase. *Dirac* is the "perfect" impulse, *impuls* an exponetial one with an extreme decay, *sinc* and *cosc* are lope-type impulses and *neutron* a randomly scattered peak. *Frqs* and time resolution should best be multiples of eachother (to allow the internal detection of time=0 start of impulse to be optimal). Ouput is a signal object and is normally fed to one of the time modules for sampling. For that use please refer to the online help and tutorials.

The impulse-types available in *menu* and the value ranges for *amps*, *frqs* and best *pars* are:

dirac	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
impuls	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (decay:1.0)
sinc	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
cosc	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
neutron	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (decay:1.0)

transients

Syntax

amps frqs decs expo

Inputs

amps	fix/fl/list	default-value 1.0
frqs	fix/fl/list	default-value 1.0
decs	fix/fl/list	default-value 1.0
expo	menu	(list of options see below)

Output

c-signal-function

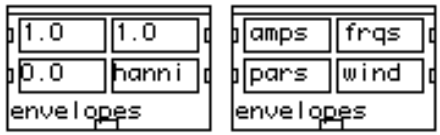
Explanation

Classic transient control module (adapted from S.Tempelaars) that generates an exponential + signal function according to choice of menu *expo*. All the functions exist in double as a sin and cos pair, creating a carrier signal controlled by *amps* and *frqs*. To that is added an exponentially decaying envelope or a beating signal both controlled by *pars*. Note that phase is not available but can be optained by adding an offset to output of one-time. The general idea is that of generating functions with transitional behaviour. Ouput is a signal object and is normally fed to one of the time modules for sampling. For that use please refer to the online help and tutorials.

The transient-types available and the value ranges for *amps*, *pins* and best *decs* are:

*e*sin*	= [-inf, +inf],	[-inf, +inf],	[0.0, +inf] (decay:1.0)
*e*cos*	= [-inf, +inf],	[-inf, +inf],	[0.0, +inf] (decay:1.0)
b+sin	= [-inf, +inf],	[-inf, +inf],	[0.0, +inf] (beats:1.0)
b+cos	= [-inf, +inf],	[-inf, +inf],	[0.0, +inf] (beats:1.0)
atsin	= [-inf, +inf],	[-inf, +inf],	[0.0, +inf] (decay:1.0)
atcos	= [-inf, +inf],	[-inf, +inf],	[0.0, +inf] (decay:1.0)
sinos	= [-inf, +inf],	[-inf, +inf],	[0.0, +inf] (decay:1.0)
sonis	= [-inf, +inf],	[-inf, +inf],	[0.0, +inf] (decay:1.0)

envelopes



Syntax

amps frqs pars wind

Inputs

amps	fix/fl/list	default-value 1.0
frqs	fix/fl/list	default-value 1.0
pars	fix/fl/list	default-value 0.0
wind	menu	(list of options see below)

Output

c-signal-function

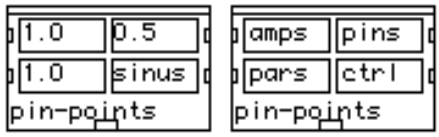
Explanation

Window control module that generates a window function according to choice of menu *wind*. Set *amps* to control magnitude, *frqs* for number of periods, and *pars* for phase. Note that *hanning* and *hamming* are special cases of the general *window*, and that *rectangle* can be made triangular by the control of *pars*. Ouput is a signal object and is normally fed to one of the time modules for sampling. For that use please refer to the online help and tutorials.

The envelope-types available and the value ranges for *amps*, *pins* and best *pars* are:

hanning	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
hamming	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
window	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (alpha:0.5)
rectangle	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (width:0.5)

pin-points



Syntax

amps pins pars ctrl

Inputs

amps	fix/fl/list	default-value 1.0
pins	fix/fl/list	default-value 0.5
pars	fix/fl/list	default-value 1.0
ctrl	menu	(list of options see below)

Output

c-signal-function

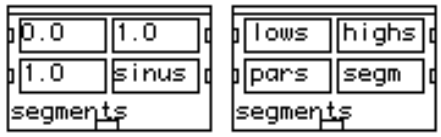
Explanation

Envelope control module that generates 1 period of a function according to choice of menu *ctrl*. Set *amps* in any range to control magnitude, *pars* in the range [0, 1] to control phase, and *pins* in the range [0, 1] to control the position of the center axis called the "pin-point". If *pins* is close to 0, the resulting curve will tend to fall left-wards (fast attack), while being close to 1 it will tend rigth-wards (soft attack, fast decay). When set to 0.5 the chosen function will be unaffected. Allows an easy creation of a family of envelopes. Ouput is a signal object and is normally fed to one of the time modules for sampling. For that use please refer to the online help and tutorials.

The pin-point-types available and the value ranges for *amps*, *pins* and best *pars* are:

sinus	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
cosinus	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
triangle	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
square	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
hanning	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
hamming	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
window	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (alpha:0.5)
rectangle	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (width:0.5)

segments



Syntax

lows highs pars segm

Inputs

lows	fix/fl/list	default-value 0.0
highs	fix/fl/list	default-value 1.0
pars	fix/fl/list	default-value 1.0
segm	menu	(list of options see below)

Output

c-signal-function

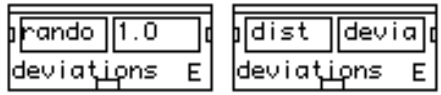
Explanation

Segment control module that generates 1 period of a function according to choice of menu *segm.* The function scales to lie in the range of *lows* and *highs* with phase set by *pars*; more segments can be joined to make a function lay inside moving boundaries. Ouput is a signal object and is normally fed to one of the time modules for sampling. For that use please refer to the online help and tutorials.

The segment-types available and the value ranges for *lows*, *highs* and best *pars* are:

sinus	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
cosinus	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
power	= [-inf, +inf],	[-inf, +inf],	[-inf, +inf] (slope:1.0)
slope	= [-inf, +inf],	[-inf, +inf],	[-inf, +inf] (slope:1.0)
phasor	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
triangle	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
square	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (phase:0.0)
impulse	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (decay:1.0)
window	= [-inf, +inf],	[-inf, +inf],	[0.0, 1.0] (edges:0.5)

deviations



Syntax

dist devia &Optional param

Inputs

dist menu	(list of options see below)	
devia	fix/fl/list	default-value 1.0

Optional

param	fix/fl/list	default-value 0.5

Output

float

Explanation

Deviation control module picking a random value according to choice of menu *dist*, in the range of *devia* (usually = 1) and with an optional *param* when choosing *beta*.

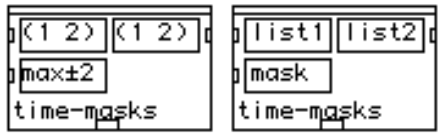
The distribution-types available are:

random	= uniform distribution, such that 0 x = *devia*
linear	= linear distribution of parameter *devia*
exponential	= variant on exponential distribution of density *devia*
triangular	= triangular distribution of average *devia*
arcsin	= arc sinus distribution of parameter *devia*
hypercos	= hyperbolic cosinus distribution of parameter *devia*
gauss	= variant on gaussian distribution that never passes *devia*
beta	= beta distribution of parameters *devia* and *param* (please extend the module).

For *devia* = *param* = 1 the result is a uniform distribution, for *devia* and *param* greater than 1 the result is similar to *gauss*.

For more elaborate information on distribution functions please refer to the "PW-alea" library or the online help and tutorials.

time-masks



Syntax

list1 list2 mask

Inputs

list1	list	default-value (1 2)
list2	list	default-value (1 2)
mask menu	(list of options see below)	

Output

list

Explanation

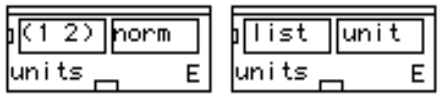
Mask control module that applies a function according to choice of menu *mask* to *list1* and *list2*, on an item-to-item basis the lists that may be of any but preferably of equal structure. The idea is to "merge" 2 lists in a way that retains aspects of both of them. F.ex. taking the maximum of 2 "peaky" control function would be a way to combine them.

The mask-types available are:

max±2	= take the abs/neg maximum of the 2 values.
min±2	= take the abs/neg minimum of the 2 values.
mean2	= take the average of the 2 values.
amax2	= take the absolute maximum of the 2 values.
amin2	= take the absolute minimum of the 2 values.
*pow*2*	= take the power 2 of the product of the 2 values.
*sqt*2*	= take the square root of the product of the 2 values.

For more information please refer to the online help and tutorials.

units



Syntax

list unit &Optional y*min y*max

Inputs

list	list	
unit	menu	(list of options see below)

Optional

*y*min*	fix/float	default-value 1.0
*y*max*	fix/float	default-value 1.0

Output

list

Explanation

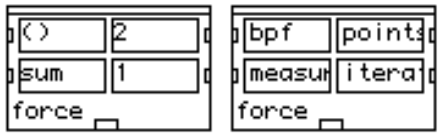
Scales *list* of (almost) anything to a range set according to choice of menu *unit*. By extending the module one can access the parameters *y*min* and *y*max*, which are used to scale the range of *list.* For that use see Explanations of menu choices below.

The scaling-types available are:

norm	= classic normalisation in the range [0, 1]
unit	= normalisation in the unit range [-1, 1]
invs	= inversion by scaling ymin, ymax of *list* to the range [*y*max, y*min*]
shift	= shift *list* to positive range only by scaling [0, *y*max-y*min*]
intgs	= scale *list* to interger range [0, 1000] fit for bpfs
self	= scale *list* symmetrically around zero in the range [*-y*max*, *y*max*]
none	= return *list* unaltered

For more information please refer to the online help and tutorials.

force



Syntax

bpf points measure iterations

Inputs

bpf	list	default-value ()	type-list (bpf list)
points	fix=0	default-value 2	
measure	menu	(list of options see below)	
iterations	fix>0	default-value 1	

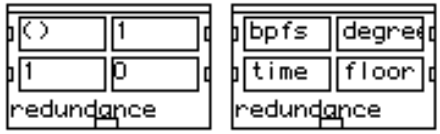
Output

list

Explanation

Module to calculate a measure of average force in *bpf*. Type of measure is calculated according to choice of menu *measure*. The function calculates from each point in *bpf* a measure over the next *points* items; that is, it progresses only one point at a time. The function repeats as many times as set by *iterations*. Note that for each iteration *bpf* will be reduced by *points*-1 items, meaning that *iterations***points*-1 should not be larger than length of *bpf*. Input to *bpf* can be (almost) anything and output is a shortend list.

redundance



Syntax

bpfs degree time floor

Inputs

bpfs	list	default-value ()	type-list (bpf list)
degree	fix/float	default-value 1	
time	fix/float	default-value 1	
floor	fix/float	default-value 0	

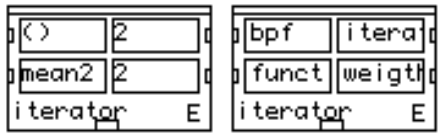
Output

c-break-point-function

Explanation

Remove redundant data from *bpfs* by eliminating adjacent points that has an angle less than *degree*, or a x-interval less than *time* and a y-value less than *floor*. Start and end values of *bpfs* are never removed. Useful in conjunction with the module **force**.

iterator



Syntax

bpf iterations function weigths &Optional permut

Inputs

bpf	list	default-value ()	type-list (bpf list)
iterations	fix/fl/list	default-value 2	
function	menu	(list of options see below)	
weigth	fix/fl/list	default-value 2	

Optional

permut	menu	(list of options see below)

Output

list

Explanation

Funny recursive function that applies a *function* on a term-by-term basis to a *bpf* and a permutated copy of itself. The action is repeated *iterations* number of times, the results being accumulated, with the last result used for successive applications of the *function*. The user can write new functions provided that they take three parameters val1, val2 and weigth, the latter used internally for, for instance, some scaling purpose. The module is extensible to include a menu *permut* which allow choises of how to permutate the list, f.ex. circular, randomly, reverse, inverse. It is meant for forcing a special tendency on a multiple copy of a list, as f.ex. going towards minimum or a maximum value, or towards mean, square, sum, etc... Input to *bpf* can be (almost) anything, and output is a flattend list of all the accummulated results.

The function-types available are:

mean2	= add 2 values and divide by *weigth* (f.ex. = 2, then it is just the average).
amax2	= take the absolute maximum of 2 values and *weigth.*
amin2	= take the absolute minimum of 2 values and *weigth.*
*pow*2*	= take the product of 2 values to the power of *weigth.*
*sqt*2*	= take the product of 2 values to the root of *weigth.*

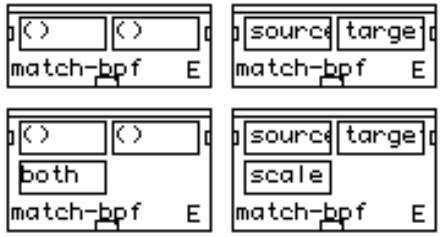
The permutation-types available are:

circular	= circular permutation, equal to the module permut-circ.
randomly	= random permutation, equal to the module permut-random.
reversed	= reverse list, equal to the module reverse.
inversed	= inverse values of list in the anbitus of itself.
x-points	= get a linear list equal to the length of itself.
copyself	= just make a copy of itself.

For further information please refer to the online help and the tutorials.

Menu *xbpfs*

match-bpf



Syntax

source target &Optional scale

Inputs

source	list	default-value ()	type-list (bpf list)
target	list	default-value ()	type-list (bpf list)

Optional

scale	menu	(list of options see below)

Output

c-break-point-function

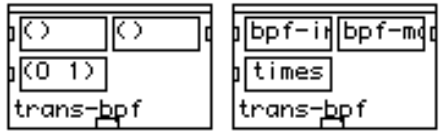
Explanation

Scale *source* to the xy-range of *target*. Useful when f.ex. *bpf-in* and *bpf-map* must match in trans-bpf. Input anything to *source* and *target* and output a bpf. An optional menu allows to choose type of scaling-target.

The types of scaling-targets available are:

both	= scale xy-range of *source* to xy-range of *target*.
xlist	= scale only x-range of *source* to x-range of *target*.
ylist	= scale only y-range of *source* to y-range of *target*.
none	= does not scale anything, allows scaling to be bypassed.

trans-bpf



Syntax

bpf-in bpf-map times

Inputs

bpf-in	list	default-value ()	type-list (bpf list)
bpf-map	list	default-value ()	type-list (bpf list)
times	list	default-value (0 1)	type-list (bpf list)

Output

c-break-point-function

Explanation

Sample *bpf-in* by *times* as a input to be transfered by *bpf-map*; a sampling-cascade takes place: first sample *bpf-map* by *times*, use that output to sample *bpf-in*. The *bpf-map* correspond to the y-axis with which *bpf-in* is going to sampled. Could be thougth of as an xy-mirror. Input anything to *bpf-in* and *bpf-map* and *times* and the module outputs a new bpf. Too see effect of Waveshaping recover y-points only using get-slot.

bias-bpf

Syntax

bpfs bias &Optional grid

Inputs

bpfs	list	default-value ()	type-list (bpf list)
bias	fix/fl/list	default-value 0.5	

Optional

grid	fix0	default-value 2

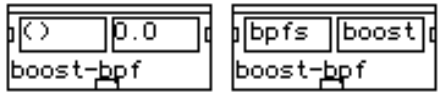
Output

list of c-break-point-functions

Explanation

Pull/push *bpfs* according to position [0, 1] of *bias*; 0 means pulling *bpfs* towards left, 1 pushes towards right. If *bias* is 0.5 then *bpfs* if left untouched. An optional *grid* tells minimum points allowed when estimating the resampling resolution. Input anything to *bpfs*, a num/list to *bias* and the module outputs per bpf in *bpfs* as many bpfs as number of *biases*.

boost-bpf



Syntax

bpfs boost

Inputs

bpfs	list	default-value ()	type-list (bpf list)
boost	fix/fl/list	default-value 1.0	

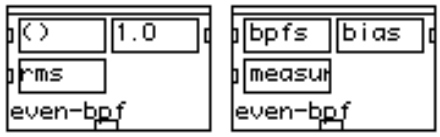
Output

list of c-break-point-functions

Explanation

Boost/cut *bpfs* according to position [-1, 1] of *boost*; -1 means reenforcing peaks, alias boost'em, 1 even-out differences, alias cut'em; 0 forces *bpfs* to an equilibre. The module ismeant to perform a self normalising exponential function. Input anything to *bpfs*, a num/list to *boost* and the module outputs per bpf in *bpfs* as many bpfs as number of *boosts*.

even-bpf



Syntax

bpfs bias measure

Inputs

bpfs	list	default-value ()	type-list (bpf list)
bias	fix/fl/list	default-value 1.0	
measure	menu	(list of options see below)	

Output

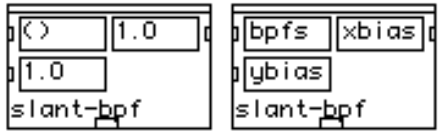
list of c-break-point-functions

Explanation

Interpolate *bpfs* towards axis as given by measure at position [-1, 1] of *bias*. Input anything to *bpfs*, a num/list to *bias* and outputs per bpf in *bpfs* as many bpfs as number of *biases*. The various interpolation reference values of menu *measure* are:

rms	= root mean square energy.
mean	= average energy.
pow2	= product power2 energy.
sum	= addup all energy.

slant-bpf



Syntax

bpfs xbias ybias

Inputs

bpfs	list	default-value ()	type-list (bpf list)
xbias	fix/fl/list	default-value 1.0	
ybias	fix/fl/list	default-value 1.0	

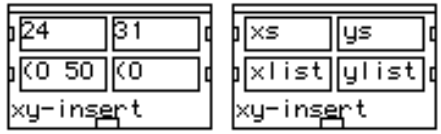
Output

list of c-break-point-functions

Explanation

Exponentially distorts *bpfs* according to factors of *ybias* and *xbias*; if factor(s) are larger than 1, then peaks are enhanced, if less, Then differences are compressed; 1 leaves *bpfs* untouched. Note that it generally works best if y-points in *bpfs* goes from min to max. Input anything to *bpfs*, a num/list to *xbias* and *ybias* and outputs per bpf in *bpfs* as many bpfs as the longest of *xbias* and *ybias*.

xy-insert



Syntax

xs ys xlist ylist

Inputs

xs	fix/fl/list	default-value (0 50 100)
ys	fix/fl/list	default-value (0 100 0)
xlist	list	default-value (0 50 100)
ylist	list	default-value (1 100 1)

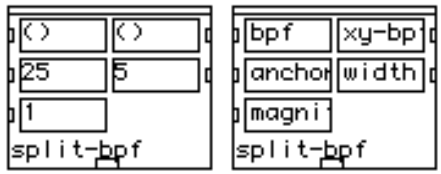
Output

list

Explanation

Non-destructive function that makes a sorted insert of *xs* and *ys* into *xlist* and *ylist*.

split-bpf



Syntax

bpf xy-bpf anchor width magnitude

Inputs

bpf	list	default-value ()	type-list (bpf list)
xy-bpf	list	default-value ()	type-list (bpf list)
anchor	fix/fl/list	default-value 25	
width	fix/fl/list	default-value 5	
magnitude	fix/fl/list	default-value 1	

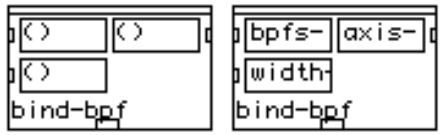
Output

c-break-point-function

Explanation

Inserts *xy-bpf* into *bpf* at x-position *anchor*; x-range of *xy-bpf* is scaled to fit within *anchor* plus-minus *width*/2, while y-range is scaled to *magintude* before being added to an interpolation between previuos and next y-points; In this way one is certain that end-points will meet so that insertion will be smooth. Note that when inserting outside of existing x-domain, the module calculates a linear extrapolation from limit values to achieve new y-values.

bind-bpf



Syntax

bpfs-list axis-list width-list

Inputs

bpfs-list	list	default-value ()	type-list (bpf list)
axis-list	list	default-value ()	type-list (bpf list)
width-list	list	default-value ()	type-list (bpf list)

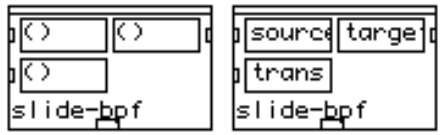
Output

c-break-point-function

Explanation

Function to "hang" a *bpfs-list* each around a time-evolving *axis-list*, degree of deviation determined by a time-evolving *width-list*. The general effect is a bpf scaled inside moving boundaries, if *width-list* = 0 a linear interpolation between points in *axis-list* occurs.

slide-bpf



Syntax

source-bpf target-bpf transi-bpf

Inputs

source-bpf	list	default-value ()	type-list (bpf list)
target-bpf	list	default-value ()	type-list (bpf list)
transi-bpf	list	default-value ()	type-list (bpf list)

all-types

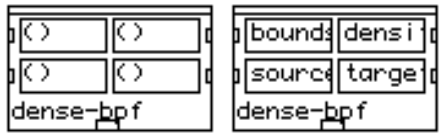
Output

list of c-break-point-functions

Explanation

Interpolate from *source-bpf* to *target-bpf* according to trajectory of *trans-bpf*.

dense-bpf



Syntax

bounds density source target

Inputs

bounds	list	default-value ()	type-list (bpf list)
density	list	default-value ()	type-list (bpf list)
source	list	default-value ()	type-list (bpf list)
target	list	default-value ()	type-list (bpf list)

all-types

Output

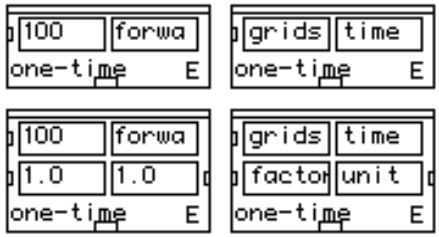
list of c-break-point-functions

Explanation

A distribution module with variable boundaries, density and distribution form. It performs subsequent sub-divisions of low/high-range of *bounds* by sampling number of steps from *density* to resample distribution-forms interpolated from *source* to *target*.

Menu *utools*

one-time



Syntax

grids time &Optional factor unit

Inputs

grids	fix/fl/list	default-value 100
time	menu	(list of options see below)

Optional

factor	fix/float	default-value 1.0
unit	fix/float	default-value 1.0

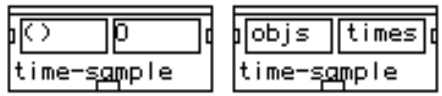
Output

list of floats

Explanation

Time module ramping in *grids* steps inside ranges set by the menu *time*; *forward* [0, 1], *backward* [1, 0], *reverse* [0, -1], *inverse* [-1, 0]. Allows for exponential ramps when extended, where *factor* set to 1 will result in linear ramps, less than 1 = acc, larger than 1 = rit. Finally an optional unit allows for scaling of result to *unit*. The module is meant to feed the *time* parameter of signal modules.

time-sample



Syntax

objs times

Inputs

objs	symbol	default-value
times	fix/fl/list	default-value 0

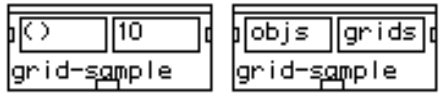
Output

list of floats

Explanation

Sample *objs* in the x-range at points given by *times*; *times* might be a number or a list of any depth and order. Sampling outside of the x-range forces output to end-points.

grid-sample



Syntax

objs grids

Inputs

objs	symbol	default-value
grids	fix/fl/list	default-value 10

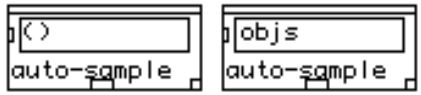
Output

list of floats

Explanation

Sample *objs* evenly over the whole x-range in *grids* steps; *grids* might be a number or a list of any depth and order reflected in the output.

auto-sample



Syntax

objs

Inputs

objs symbol default-value

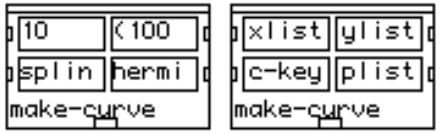
Output

list of floats

Explanation

Automatic sampling of *objs* at greatest-common-divisor-interval of x-points. It will always maintain original y-points, thus preventing sampling errors when x-intervals are uneven. May take a while if x-points of *objs* are very scattered or numbers with many decimals.

make-curve



Syntax

xlist ylist c-key plist

Inputs

xlist	fix/fl/list	default-value 10
ylist	list	default-value (100 200)
c-key	menu	(list of options see below)
plist	menu	(list of options see below)

Output

c-curve-function

Explanation

Global make of a curve function according to choice of menu *c-key*. If input is just a *ylist* then *xlist* is an interval (as with bpf), otherwise *xlist* and a *ylist* should be of equal lengths. For the use of the menu *plist* please refer to the tutorials.

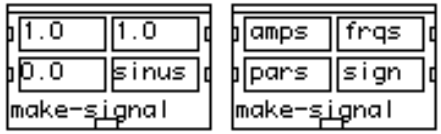
The curve-types available are:

spline = see **make-spline**.
blend = see **make-blend**.
matrix = see **make-matrix**.
hermite = see **make-hermite**.
bezier = see **make-bezier**.

When choosing for a matrix-function, the matrix-type available are:

hermite = see Explanations in **make-matrix**.
parabol = see Explanations in **make-matrix**.
bspline = see Explanations in **make-matrix**.
catmull = see Explanations in **make-matrix**.
tension = see Explanations in **make-matrix**.

make-signal



Syntax

amps frqs pars sign

Inputs

amps	fix/fl/list	default-value 1.0
frqs	fix/fl/list	default-value 1.0
pars	fix/fl/list	default-value 0.0
sign	menu	(list of options see below)

Output

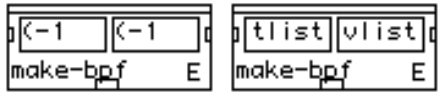
c-signal-function

Explanation

Global make of a signal function according to choice of menu *sign*. Set *amps* to control magnitude, *frqs* for number of periods, and *pars* for phase. For further use please refer to the explanations listed below.

sinus	= see explanation in **signals**.
cosinus	= see explanation in **signals**.
power	= see explanation in **signals**.
slope	= see explanation in **signals**.
phasor	= see explanation in **signals**.
triangle	= see explanation in **signals**.
square	= see explanation in **signals**.
impuls	= see explanation in **impulses**.
hanning	= see explanation in **envelopes**.
hamming	= see explanation in **envelopes**.
window	= see explanation in **envelopes**.
rectangle	= see explanation in **envelopes**.
dirac	= see explanation in **impulses**.
impuls	= see explanation in **impulses**.
sinc	= see explanation in **impulses**.
cosc	= see explanation in **impulses**.
neutron	= see explanation in **impulses**.
*e*sin*	= see explanation in **transients**.
*e*cos*	= see explanation in **transients**.
b+sin	= see explanation in **transients**.
b+cos	= see explanation in **transients**.
atsin	= see explanation in **transients**.
atcos	= see explanation in **transients**.
sinos	= see explanation in **transients**.
sonis	= see explanation in **transients**.

make-bpf



Syntax

tlist vlist &Optional logs

Inputs

tlist	list	default-value (-1 1)
vlist	list	default-value (-1 1)

Optional

logs	menu	default-value 10

Output

c-break-point-function

Explanation

Makes a floating break point function from *tlist* and *vlist*. For floating points to be seen in a bpf-editor points are internally scaled to a range choosen by (optional) menu *logs* (original xy-points are unaffected).

Available scaling types are:

log10	= write 10 or choose log10 : scales to [-1000, 1000].
log2	= write 2 or choose log2 : scales to [-16383, 16383].
log1	= write 1 or choose none : no scaling.

make-xbpf

Syntax

thing

Inputs

thing symbol default-value ()

Output

c-break-point-function or a list of c-break-point-functions

Explanation

Automatically make xbpfs according to type of *thing*. Types can be numbers, single or double lists, single or multi bpfs, single or multi curves, single or multi signals or any combination of these.

get-x-points

Syntax

objs

Inputs

objs symbol default-value ()

Output

list of numbers

Explanation

Gets all x-points from *objs* whatever the type might be. Types can be numbers, single or double lists, single or multi bpfs, single or multi curves, single or multi signals or any combination of these.

get-y-points

Syntax

objs

Inputs

objs symbol default-value ()

Output

list of numbers

Explanation

Gets all y-points *objs* whatever the type might be. Types can be numbers, single or multi lists, single or multi bpfs, single or multi curves, single or multi signals or any combination of these.

4 Implementation

The GenUtils Library is next to being a library in itself, also meant to provide a set of basic tools in the development of new libraries. Thus a few word of its implementation and organisation.

Firstly it is based on CLOS using objects as its basic data-type. Everything is here objects; numbers, list, functions, signal-objects. The technique of generic methods has been vastly employed, so that the basic commands should apply to all data-types necessary for PatchWork, with the exception of symbols, characters and strings. It means that type-differences are taken care of by the Lisp-system and should be entirely transperant to the user. Besides using the basic data-types as numbers and list, a small class system has been developed for the splines, signals and functions. Its root-class is called **c-gen**, and that's where basic methods are defined:

The methods lists as follows:

```
(defmethod nlength ((self c-gen)) 1)
(defmethod give-x-points ((self c-gen)) 1)
(defmethod give-y-points ((self c-gen)) 1)
(defmethod do-any ((self c-gen) method &rest args) (apply method self args))
(defmethod signal-out (self c-gen) &rest args) (declare (ignore args)) 1)
(defmethod sample-grid ((self c-gen) grid) (declare (ignore grid)) 1)
(defmethod sample-time ((self c-gen) times) (declare (ignore times)) 1)
(defmethod sample-auto ((self c-gen)) 1)
```

These methods are also defined for numbers (subclass of t) and lists (subclass of cons) Any new methods should at least always be defined for at list t, cons and c-gen. Usually a specific action is needed for the subclasses of c-gen, such as f.ex. **signal-out**, that has been defined differently for **c-curve**, **c-signal**, **c-bpf**. Take care that the class **c-break-point-function** from PatchWork has been redefined here to be a subclass of **c-gen**, to incorporate it in both the GenUtils method system, and to maintain its link with the **c-multi-function**, the display module of PatchWork. At the heart of the method system is the method **do-any**, which is the only test performed to differentiate between types. For the type *list* it goes as follows:

```
(defmethod do-any ((self cons) method &rest args)
(if (every #'numberp self) (apply method self args)
        (mapcar #'(lambda(obj) (apply #'do-any obj method args)) self)))
```

In words it reads, if "I" am a true number-list then apply action on arguments, else reapply **do-any** on each one of the parameters in arguments. This is simply a recursion, that will eat a tree until it reach recognisable objects. It takes advantages of the property that methods can be "applied" to arguments, just as with ordinary Lisp-functions. So following many of the core modules has been defined using **do-any** *obj method args*. This is how proper objects to their proper version of method is "recognized".

Many modules has been defined to be easely incorporated into further Lisp-code:

```
(make-bpf-class tlist vlist &optional logs)
```

for more, see files `:function:bpf-tools & :shapes:tranfer-xbpfs`.

```
(make-bezier-class xlist ylist plist)
(make-hermite-class xlist ylist plist)
(make-blend-class xlist ylist plist)
(make-matrix-class xlist ylist matrix)
```

(**make-spline-class** xlist ylist)
for more, see file `:curves:class-curves`.

(**make-signal-class** sign-type func-type amp-pars frq-pars pha-pars)
for more, see file `:shapes:class-shapes`.

All classes has methods to eveluate their function, please refer to source code.
For the **bpf-class**, see patch-value in file `:function:bpf-tools`.
For the **bezier-class**, see eval-bezier in file `:curves:blend-curve`s.
For the **hermite-class**, see eval-hermite in file `:idem`.
For the **blend-class**, see eval-blend in file `:idem`.
For the **matrix-class**, see eval-matrix in file `:idem`.
For the **spline-class**, see eval-spline in file `:curves:spline-curves`.
For the **signal-class**, see signal-out in file `:shapes:class-shapes`.

Note that the patch-value mechanism of PatchWork has not been maintained throughout the GenUtils library, except for the **BPF**, which is a redefinition and for compatibillity kept so. The technique of indirect invocation (or "ghost" modules) used allover in PatchWork, where originally adopted to aid graphical programming. They will not though evaluate correctly, if just included into ordinary Lisp code (returns a nil, unless one calls their corresponding patch-value method), and therefore it has been decided to keep GenUtils modules as close to any other Lisp function (defined using **defun**) as possible. GenUtils modules can then be used directly as is in other code.

Why Objects? Choice and comparison of programming styles

In order to demonstrate the the reason why GenUtils is using CLOS, one could do so by showing the difference between classic sequential and object-oriented programming. As an example, a very central function called **make-xbpf** has been implemented in both ways. It should make from (almost) anything as input the number of **BPF**(s) needed, reflecting the tree-structure of the input in the output, that is, respecting levels of parenthesis. In GenUtils one must differentiate numbers, lists, double-lists, objetcs, list-of-objects.

1- In Sequential style

A number of testing function must be made to check type of input at any level.

```
(defun xpred (list pred &optional (depth 0))
"recursive function, that test if item of list at parenthesis depth is satisfying pred"
(if (and (> depth 0) (listp list))
        (xpred (car list) pred (1- depth))
        (funcall pred list)))

(defun bpfp (anything)
"bpf-predicate, returns t if anything is of type bpf, otherwise nil"
(eq 'c-break-point-function (type-of anything)))

(defun nor (flag1 flag2)
"returns t if neither flag1 nor flag2 is true, otherwise nil"
(not (or flag1 flag2)))

(defun make-dummy-bpf (vlist)
"make bpf from vlist (y-points) with auto-setup of tlist (x-points"
(make-break-point-function (arithm-ser 1 1 (length vlist)) vlist))
```

Here is the sequential version, typically with a intervowen serie of conditional tests:

```
( defun make-xbpf (anything)
```

"If *anything* is a number, a single or a double list, and not already a bpf or a xbpf, then make bpf(s) accordingly. When *anything* is a list, supposes it to be of equal types"

```
(if (nor (xpred anything 'bpfp 0) ;if-not a bpf or a multi-bpf
        (xpred anything 'bpfp 1))
        (if (and (xpred anything 'numberp 2);then is it a list of lists?
                (xpred anything 'listp 1))
                (mapcar #'make-dummy-bpf anything)
                (if (and (xpred anything 'numberp 1) ;if-not, is it a simple list?
                        (xpred anything 'listp 0))
                        (make-dummy-bpf anything)
                        (if (xpred anything 'numberp 0) ;if-not, is just a number?
                                (make-dummy-bpf (list anything)) anything))) anything))
```

Though the code fills up some space it is fast in execution, but hard to alter, because of the hard logic of the predicates. A further generalisation would be to let it accept functions as arguments (in the same manner as below).

2- In Object-Oriented style:

As mentioned earlier the method **do-any** has been defined for the existing class **cons** (subtype of list), which is the only test performed to differentiate between types:

```
( defmethod do-any ((self cons) method &rest args)
```

"Apply *method* on *self* and optionally *args*, if *self* is a true number list, otherwise recursivly reapply **do-any** to each item of *self* with *method* & *args*.*Self* does not suppose its elements to be of equal types."

```
(if (every #'numberp self) (apply method self args)
        (mapcar #'(lambda(obj) (apply #'do-any obj method args)) self)))

(defunp make-xbpf ((anything symbol (:value "()"))) all-types

"auto make bpf(s) according to type of anything"

(do-any anything #'make-dummy-bpf))
```

So here the recognition of types is left to the system, only by defining one method. In combination with the technique of indirect invocation that applies to both functions and methods, the amount of code has been drastically reduced, and in addition one can now go on writing simple functions like **make-dummy-bpf**, without having to worry further on correct types (as it should be by the way!).

5 Description of Splines

Splines as they are used in the GenUtils Library are polynomial functions, adopted for their abillity to find a "pleasant" trajectory between any points.

Generally they might do so in 2 ways:

- either "approximating" the original points, though not touching them, as f.ex. in *bspline* of **make-matrix**.
- or by "interpolating" the original points, forced to touch them as f.ex. in *catmull* of **make-matrix**.

There also exist a "mixed" category, that touches only start/end points and approximates the ones in between; as f.ex. in*hermite* of **make-matrix**.

Here follows an explanation of the principles and nature of each one of them, in the order they are found in the menu **curves**.

Bezier Polynomial Function from the make-bezier **module**

Reference: John Vince "3D Computer Animation" (Addison-Wesley).

This is a 2 point interpolation function with 1 or 2 control points, so as to pull or push interpolation curve to/from source/target:

$$f(t, N) = (1-t)^N * x[0] + t^N * x[1] + Nt(1-t)^{N-1} * p[0] + Nt^{N-1}(1-t) * p[1]$$

where	t	= 0 ≤ time ≤ 1(normalized time)
	N	= degree related to number of control points
	cp equals N = 2, 2 cp equals N = 3	
	x[0]	= returns source value
	x[1]	= returns target value
	p[0]	= returns first control value
	p[1]	= returns second control value

The function is implemented so that it takes both x, y coordinates into account, where the nature of the 2 polynomial degrees can be described as follows:

- Quadratic : with 1 control point as a pull/push interpolation or envelope.
- Cubic : with 2 control points to determine the slope at segment start/end.

The idea of usage

It should be relativly easy to use for making "bended" trajectories between 2 data-points only (called a segment). It does not work well for splicing more segments because it cannot trace abrubt changes smoothly. Because control points can be given as x, y coordinates directly, the calculation of controls as binding points should be straigthforward (as offsets). Also note that control points are not restricted to the x, y range of the origines.

The formula exist in an elaborate and more general version called: **make-blend** (see later)

Parametric Polynomial Function from the make-hermite **module**

Reference: John Vince "3D Computer Animation" (Addison-Wesley).

This is a 2 point interpolation function with 1 or 2 control points, that uses parameter matrices of third-degree (cubic) polynomial, so as to speedup/slowdown departure/arrival of an interpolation curve between 2 points.

It is a special version of the more general **make-matrix**, please refer to this.

The idea of usage

Meant relativly easy to shape a trajectory between 2 points only, which, is a very commonly need, as for instance when going smoothly from one chord to another, or any other situation with only 2 states. For the splicing of more than 2 states, it is recommended to use make-matrix instead, which is esp. made for that purpose.

Cubic Spline Function from the make-spline module

Reference: Robert Sedgewick "Algorithms" (Addison-Wesley 1988).

This is a method to connect adjacent points by a "pleasing" smooth curve.

The model is based on a third-degree (cubic) polynomial defined as:

$$S[i](X) = a[i]x^3 + b[i]x^2 + c[i]x + d[i],$$
with i = 1 to N-1, in the interval x[i] to x[i+1]

The spline is represented as a system of simultanous equations in a matrice with N-2 unknowns to be solved by Gaussian elimination. For reasons of saving data memory a "tridiagonal" matrice is used.

The program flows as follows:

- first : the x, y data-points are split into a matrice representation,
- second : a forward elemination phase removes variables under the diagonal,
- third : a backward substitution phase replace the unknowns by their answer,
- fourth : the evaluation of the spline curve with the found coefficients.

The idea of usage

To make a seemingly knotless trajectory between arbitrary data-points, eg. the interconnection of two MIDI-controllers like pan/volume. It is also usefull for splicing not-so-well-fitting data-blocks together, or the frequency/amplitude variation of a slowly transforming sound, and not the least for smooth functional fluctuation of a tempo/rhythm.

Lisp implementation note

For clarity of code and for security in indexing the **loop for**... statement is here extensivly used, though it migth hurt the eye of some Lisp puritans. That is, the function is implemented in a canonical manner, and at present it is not known whether **recursion** or **mapcar** could speed it up.

Parametric Polynomial Functions from the make-matrix module

Reference: John Vince "3D Computer Animation" (Addison-Wesley).

This method interpolate adjacent points by a "smooth" curve.

The model uses parameter matrices of third-degree (cubic) polynomial:

$$f(t) = [t^3\ t^2\ t\ 1] \begin{bmatrix} -T & 2-T & T-2T \\ 2T & 3-T & 3-2T & -T \\ -T & 0 & T & 0 \\ 0 & T & 0 & 0 \end{bmatrix} \begin{bmatrix} y0 \\ y1 \\ y2 \\ y3 \end{bmatrix}$$

where t = 0 ≤ time ≤ 1 (normalized time)
T = tension parameter (eg. catmull = 0.5)
x[i] = succesive y"s indexed by x[i, N-1]

Setting up the middle matrix means to parametresize the polynomial, so as to facilitate the estimation of correct/ good slope-parameters.

The nature of the different matrices can be described as follows:

- hermite : different kind of "bended" slopes between two values,
- parabol : more lively version that has a "peak/valley" tendency,
- bspline : an "approximating" spline tracing a path towards the points,
- catmull : an "interpolating" spline tracing a smooth curve through the points.

The idea of usage

To easily make a certain "profile" specified with a minimum of data, aiming at different degrees of "smoothness" / "jaggedness". Through putting in identical x-values, one can obtain abrupt changes. Also if y-list is 2 points longer than x-list these will be taken as constant slopes, used for every segment, whereas equal-length xy-list will use a succesive sliding set of 4 y-values to determine slope-character. In general hermite is best with constant slopes, while the others can have constant/variable slopes as wished.

It is well fitted for making envelopes with an elastic character.

Polynomial Segment Function from the make-blend module

Reference: John Vince "3D Computer Animation" (Addison-Wesley).
This is a method to interpolate adjacent points by a "hill/valley" curve.
The model is based on a fouth-degree (quartic) polynomial defined as:

$$f(t, d, N) = (1-t)^d * x[i] + t^d * x[i+1] + \sum[0, N] t(1-t)^N * A + (N+N/2)t^{N(1-t)} * B$$

where	t	= 0 ≤ time ≤ 1 (normalized time)
	d	= polynomial degree (can be any float >= 0)
	N	= polynomial degree - 1 (must be abs int [0, 3])
	x[i]	= return y source value at x[low]
	x[i+1]	= return y target value at x[high]
	A	= source slope control value
	B	= target slope control value
	∑[0, N]	= sum integers from 0 to N
	(N+N/2)	= shift degrees in pairs

As a function it can smoothly change from one polynomial degree to another,
by correctly adjusting d & N, while A & B tells the slope at segment start/end.
The nature of the different polynomial degrees can be described as follows:

- Unique : usually a lin|exp interpolation disregarding A & B,
- Quadratic : a "squared" interpolation using A only to tell acceleration,
- Cubic : a "cubed" interpolation using both A & B to peak|flatten curve,
- Quartic : a curve using both A & B to jag extremes in quasi-periods.

The idea of usage

To make "bended" trajectories between abitrary data-points, note necessarily aiming at being "smooth" throughout the segments. Though d, N, A, B migth be difficult to estimate, N, A, B can optionally be left to default calculation:

N = abs-integer(log10(1/d)), A = y1, B = y2.

Also note the mirroring-effect, when A & B are specified to be negative. It is not known wether there exists a straigth forward normalisation, so meanwhile please "transpose" curve to positive domain using f.ex. **units**.

It is imaginable to use it for envelopes, rhythms, impulses, filters, etc.

6 Formulas used in GenUtils

Interpolation Functions

A classic unit scaler that maps x in the range [x1, x2] to [y1, y2]

f(x) = (y2-y1) / (x2-x1) * (x-x1) + y1 , where

x1 : low boundary of x

x2 : high boundary of x

y1 : low boundary of y

y2 : high boundary of y

x : any number

A non-standard pin scaler that maps x in the range [0, pin] to [0, 1/2] & [pin, 1] to [1/2, 1]

f(x) = unit(mod(x, 1), min(f, p), max(f, p), min(f, 1/2), max(f, 1/2)), where

p : pin point position [0, 1]

f : flag 1 if $p \leq x$

0 otherwise

x : unit time [0, 1]

Trigonometric Functions

A classic sine oscillation of f periods with A amplitude

f(x) = A * sin ($2\pi f$x - 2πø), where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A classic cosine oscillation of f periods with A amplitude

f(x) = A * cos ($2\pi f$x - 2πø), where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A classic hanning window of f periods with A amplitude

f(x) = A * -0.5cos($2\pi f$x-2πø)+0.5, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A classic hamming window of f periods with A amplitude

f(x) = A * -0.46cos($2\pi f$x-2πø)+0.54, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A classic general window of f periods with A amplitude

$f(x) = A * (ß-1)\cos(2\pi f x)+ß$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ß : beta [0, 1]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

Linear Functions

A classic sawtooth oscillation of f periods with A amplitude

f(x) = A * mod(fx-ø, 1), where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

x : unit time [0, 1]

A classic square oscillation of f periods with A amplitude

f(x) = A * mod (truncate(fx+ø+0.5), 0.5), 2), where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

x : unit time [0, 1]

A non-standard triangular oscillation of f periods with A amplitude

f(x) = A * (1-|2mod(fx-ø, 1)-1|), where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

x : unit time [0, 1]

A non-standard rectangular oscillation of f periods with A amplitude

f(x) = A * $(1-|2f\text{x}-1|)^{2e}$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

e : exponential factor [-inf, +inf]

x : unit time [0, 1]

Impulse and Conditional Functions

A classic conditional impulse oscillation of *f* periods with A amplitude

f(x) = {A if -1.0E-100 < mod(*f*x-ø, 1) < *f*/r, where

{0 otherwise

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

r : resolution [100]

x : unit time [0, 1]

A classic exponential impulse oscillation of *f* periods with A amplitude

$f(x) = A * (10^d)^{(e*d*\mathrm{mod}(fx, -1))}$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

e : decay factor [-inf, +inf]

d : decimals [0, 8]

x : unit time [0, 1]

A general sinc oscillation of *f* periods with A amplitude

$$f(x) = A * \frac{\sin(2\pi fx-2\pi ø)^e}{x}$$

where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

e : decay factor [2]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A general cosc oscillation of *f* periods with A amplitude

$$f(x) = A * \frac{\cos(2\pi f x - 2\pi ø)^e}{x}$$

where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

e : decay factor [2]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A non-standard quasi-random impulse oscillation of *f* periods with A amplitude

$f(x) = A * ((10^d)^{e*d*\mathrm{mod}(f x, -1)})^{\mathrm{ran}(\mathrm{mod}(f(1-x-ø), 1))}$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ø : phase [0, 1]

e : decay factor [-inf, +inf]

d : decimals [0, 8]

x : unit time [0, 1]

Transient Functions (adapted from S. Tempelaars)

A simple sine oscillation (no phase) of f periods with A amplitude

$f(x) = A * \sin(2\pi f x)$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A simple sine oscillation (no phase) of f periods with A amplitude

$f(x) = A * \cos(2\pi f x)$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

An exponential decay sine oscillation of f periods with A amplitude

$f(x) = A * \sin(2\pi f x) * e^{-\pi ß x}$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ß : decay factor [-inf, +inf]

2π : two π [6.283185307179586…] (unit circle)

$-\pi$: minus π [-3.141592653589793…]

x : unit time [0, 1]

An exponential decay cosine oscillation of f periods with A amplitude

$f(x) = A * \cos(2\pi f x) * e^{-\pi ß x}$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ß : decay factor [-inf, +inf]

2π : two π [6.283185307179586…] (unit circle)

$-\pi$: minus π [-3.141592653589793…]

x : unit time [0, 1]

A attack/decay sine oscillation of f periods with A amplitude

$f(x) = Ax * \sin(2\pi fx) * e^{-\pi ßx}$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ß : decay factor [-inf, +inf]

2π : two π [6.283185307179586…] (unit circle)

-π : minus π [-3.141592653589793…]

x : unit time [0, 1]

A attack/decay cosine oscillation of f periods with A amplitude

$f(x) = Ax * \cos(2\pi fx) * e^{-\pi ßx}$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ß : decay factor [-inf, +inf]

2π : two π [6.283185307179586…] (unit circle)

-π : minus π [-3.141592653589793…]

x : unit time [0, 1]

A beat sine oscillation (no phase) of f periods with A amplitude

$f(x) = A * \sin(2\pi fx) * \sin(2\pi ßx)$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ß : beat frequency in Hz

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A beat cosine oscillation (no phase) of f periods with A amplitude

$f(x) = A * \cos(2\pi fx) * \cos(2\pi ßx)$, where

A : amplitude [-inf, +inf]

f : frequency in Hz

ß : beat frequency in Hz

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A transient sine+cosine oscillation of f periods with A amplitude

$f(x) = A * (\sin(2\pi f x) + \cos(2\pi\Omega x) * e^{-\pi ß x})$, where

A : amplitude [-inf, +inf]

f : carrier frequency in Hz

Ω : transient frequency in Hz

ß : decay factor [-inf, +inf]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

A transient cosine+sine oscillation of f periods with A amplitude

$f(x) = A * (\cos(2\pi f x) + \sin(2\pi\Omega x) * e^{-\pi ß x})$, where

A : amplitude [-inf, +inf]

f : carrier frequency in Hz

Ω : transient frequency in Hz

ß : decay factor [-inf, +inf]

2π : two π [6.283185307179586…] (unit circle)

x : unit time [0, 1]

Distribution Functions (adapted from M. Malt)

A uniform distribution in the unit-range

f() = ran(1.0),

such that $0 \leq y \leq 1$ which may touch boundaries

A uniform distribution in the any-range

f() = ran(low, high),

such that low < y < high which must NOT touch boundaries

A linear distribution in the range of from zero to deviation D

f(D) = D * 1-sqrt(ran(1.0)), where

D: magnitude of deviation

An exponential distribution in the range of from zero to deviation D

$$f(D) = D^* \frac{\log(ran(low, high))}{-10}$$

where

D : magnitude of deviation

low : lower inhibitance limit

high : higher inhibitance limit

A triangular distribution in the range of from zero to deviation D

$$f(D) = D * \frac{ran(1.0)+ran(1.0)}{2}$$

where

D : magnitude of deviation

An arc sine distribution in the range of from zero to deviation D

$f(D) = D * \sin(\pi * ran(1.0)/2)^2$, where

D : magnitude of deviation

π : π [3.141592653589793…]

A hyperbolic cosine distribution in the range of from zero to deviation D

f(D) = D * log(tan(sin(π*ran(low, high)/2))), where

D : magnitude of deviation

π : π [3.141592653589793…]

low : lower inhibitance limit

high : higher inhibitance limit

A variant on gaussian distribution in the range of from zero to deviation D

$$f(D) = D * \frac{\sum[1, 12]ran(1.0)-6}{6.5535}$$

where

D : magnitude of deviation

A variant on beta distribution in the range of from zero to deviation a, b

f(a, b) = {f(a, b) if 1 < (y1+y2), where

$y1 = ran(1.0)^{1/a}$

$y2 = ran(1.0)^{1/b}$

{y1/(y1+y2) otherwise

a : magnitude of deviation

b : magnitude of reduction

7 Documentation of menus in Signal Modules

Here is a file of parameters, their names and appropriate value ranges, for the functions found in the scrolling menus of the modules under **shapes**. All functions take 3 parameters, the 2 first of which are always the same in a module, the third being the "joker-parameter", changing its functions according to choice of menu.

In the charts below, the first column indicates the menu choice, followed by the three parameters:

1st input (amps), 2nd input (frqs), 3rd input (pars).

signals

	amps = magnitude	frqs = periods	pars = see below
sinus	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = phase (0.0)
cosinus	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = phase (0.0)
power	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[-inf, +inf] = slope (1.0)
slope	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[-inf, +inf] = slope (1.0)
phasor	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = phase (0.0)
triangle	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = phase (0.0)
square	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= phase (0.0)
impulse	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= decay (1.0)
window	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= edges (0.5)

impulses

	amps = magnitude	frqs = periods	pars = see below
dirac	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = phase (0.0)
impulse	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = decay (1.0)
sinc	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = phase (0.0)
cosc	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = phase (0.0)
netron	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0] = decay (1.0)

transients

	amps = magnitude	frqs = periods	pars = see below
e*sin	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= decay (1.0)
e*cos	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= decay (1.0)
b+sin	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= beats (1.0)
b+cos	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= beats (1.0)
atsin	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= decay (1.0)
atcos	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= decay (1.0)
sinos	[-inf, +inf] (1.0),	[0.0, +inf] (1/2),	[0.0, 1.0]= decay (1.0)
sonis	[-inf, +inf] (1.0),	[0.0, +inf] (1/2),	[0.0, 1.0]= decay (1.0)

envelopes

	amps = magnitude	frqs = periods	pars = see below
hanning	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= phase (0.0)
hamming	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= phase (0.0)
window	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= alpha (0.5)
rectangle	[-inf, +inf] (1.0),	[0.0, +inf] (1.0),	[0.0, 1.0]= width (0.5)

pin-points

	amps = magnitude	pin = position	pars = see below
sinus	[-inf, +inf] (1.0),	[0.0, +inf] (0.5),	[0.0, 1.0]= phase (0.0)
cosinus	[-inf, +inf] (1.0),	[0.0, +inf] (0.5),	[0.0, 1.0]= phase (0.0)
triangle	[-inf, +inf] (1.0),	[0.0, +inf] (0.5),	[0.0, 1.0]= phase (0.0)
square	[-inf, +inf] (1.0),	[0.0, +inf] (0.5),	[0.0, 1.0]= phase (0.0)
hanning	[-inf, +inf] (1.0),	[0.0, +inf] (0.5),	[0.0, 1.0]= phase (0.0)
hamming	[-inf, +inf] (1.0),	[0.0, +inf] (0.5),	[0.0, 1.0]= phase (0.0)
window	[-inf, +inf] (1.0),	[0.0, +inf] (0.5),	[0.0, 1.0]= alpha (0.5)
rectangle	[-inf, +inf] (1.0),	[0.0, +inf] (0.5),	[0.0, 1.0]= width (0.5)

segments

	amps = low limit	pin = high limit	pars = see below
sinus	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= phase (0.0)
cosinus	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= phase (0.0)
power	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= slope (1.0)
slope	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= slope (1.0)
phasor	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= phase (0.0)
triangle	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= phase (0.0)
square	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= phase (0.0)
impulse	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= decay (1.0)
window	[-inf, +inf] (0.0),	[-inf, +inf] = (1.0),	[0.0, 1.0]= alpha (0.5)

8 References

F.R. Moore, "ELEMENTS OF COMPUTER MUSIC", (Prentice Hall, Englewood Cliffs, New Jersey, 1990)

F.R. Moore, "AN INTRODUCTION TO THE MATHEMATICS OF DSP", in Digital Audio Signal Processing: an anthology (W. Kaufmann, Los Altos, California, 1985)

J.O. Smith, "INTRODUCTION TO DIGITAL FILTER THEORY", in Digital Audio Signal Processing: an anthology (W. Kaufmann, Los Altos, California, 1985)

S. Tempelaars, "SIGNAL PROCESSING IN SPEECH & MUSIC", (Koninklijk Conservatorium, Den Haag, Holland, 1988)

G.L. Steel Jr, "COMMON LISP, the language" (2nd edition) especially ch. 2 on TYPES & 28 on CLOS (Digital Equipment Cooperation, USA, 1990)

R. Sedgewick, "ALGORITHMS" , Addison-Wesley, 1988. (See chapter on cubic spline)

J. Vince, "3D COMPUTER ANIMATION", description of other splines and film animation techniques (Addison-Wesley)

Index

O

P

Q

R

S

T

U

X