Ircam documentation

- Research reports
- Musical works
- Software

PatchWork

Reference

Third Edition, April 1996

IRCAM Centre Georges Pompidou

This manual was written by Mikhail Malt,
in collaboration with Curtis Roads,
and produced under the editorial responsibility of
Marc Battier - Marketing Office, Ircam.

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

Version 3.0 of the documentation, April 1996.
This documentation corresponds to version 2.5.1 of the software.

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 Groups

The use of this program and its documentation is strictly reserved for members of the IRCAM Software Users Groups. For further information, contact:

Marketing Office

IRCAM

1, Place Stravinsky

F-75004 Paris

France

Telephone (1) 44 78 49 62

Fax (1) 42 77 29 47

Electronic mail: bousac@ircam.fr

Please send suggestions and comments on this documentation to

Electronic mail: ircam-doc@ircam.fr

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

Vous trouverez dans ce manuel de référence la description des modules qui composent PatchWork (jusqu'à la version 2.5). Les modules sont regroupées en 16 sections:

1. **Data**
2. **Arithmetic**
3. **Num-series**
4. **Function**
5. **Control**
6. **List**
7. **Set**
8. **Combinatorial**
9. **Abstract**
10. **Breakpoint function**
11. **Extern**
12. **Multidimensional**
13. **Edit**
14. **Conversion and approximation**
15. **MIDI**
16. **Multidimensional music**

La documentation de PatchWork est complétée par un manuel d'introduction, un tutorial et une série de manuels sur les différentes librairies qui proposent des approches diversifiées à la composition assistée par ordinateur. Un document à parution périodique, « PatchWork Newsletter », rend compte de l'état de l'environnement PatchWork, des numéros des dernières versions et donne des informations sur les améliorations récentes apportées au logiciel et aux librairies.

Introduction

This reference manual documents the core non-library modules forming the PatchWork toolkit. The modules are grouped here according to function, as they are in the Kernel and Music menus:

1. **Data**
2. **Arithmetic**
3. **Num-series**
4. **Function**
5. **Control**
6. **List**
7. **Set**
8. **Combinatorial**
9. **Abstract**
10. **Breakpoint function**
11. **Extern**
12. **Multidimensional**
13. **Edit**
14. **Conversion and approximation**
15. **MIDI**
16. **Multidimensional music**

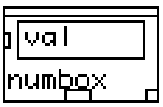
To see an example of each module in a patch, select the module in an open window and type t.

Conventions

In this manual, the Macintosh keyboard **command** key is identified with the clove symbol.

Data Modules

numbox



Syntax

(pw::numbox val)

Input

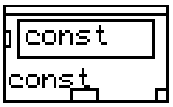
val	An integer or float

Output

returns	*val*

The **numbox** module accepts integers and floating-point values. It returns the value it receives on its input. This module is useful for controlling many inputs that must have the same value.

const



Syntax

(pw::const const)

Input

const	Any data type (integer, float, string, list, etc.)

Output

returns	The data contained in the module

This module controls numerical values or lists (on many levels). It accepts either numerical values, symbols, or mixed values. It returns a list without evaluating it, and it can also be useful for control of many inputs that must have the same list.

evconst

Syntax

(pw::evconst const)

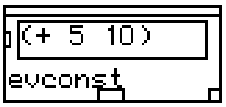
Input

const	Any data type (integer, float, string, list, etc.)

Output

returns	The evaluation of the contents of the module

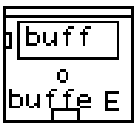
This module controls numerical values or lists (on many levels). It accepts either numerical values, symbols, or mixed values. Contrary to const, evconst returns the evaluation of its input. Specifically, this module behaves like the Lisp expression (eval *const*), where *const* is the input value of the module. For example,



will return

? PW->15

buffer



Syntax

(c-patch-buffer::buffer buff)

Input

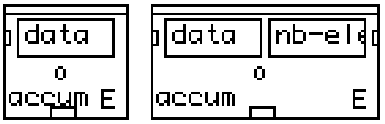
buff	Any data type (integer, float, string, list, etc.)

Output

returns	If the module is unlocked, it returns *buff* (if not connected) or evaluate the patch to which it is connected. If the module is locked, it returns the last value.

The **buffer** module stores the results of patch calculations connected to its input. It has two states: open (indicated by a small *o* on the module) and closed (indicated by *x*). The user can switch between these two states by clicking on the *o* or the *x*. When the module is open, it behaves exactly like the module **const**. When it is closed it returns the last value evaluated. It is advisable to close the module immediately after evaluation to avoid recalculating the input.

accum



Syntax

(c-patch-buffer::accum data &optional nb-elems)

Input

data	Any data type (integer, float, string, list, etc.)

Optional Input

nb-elems	Fixnum > 0

Output

returns	If the module is locked it returns a list of all values of *data* received before the module was locked.

The **accum** module accumulates the results of calculations of a patch that is connected to its input. It has two states: open (indicated by a small *o* on the module) and closed (indicated by *x*). The user can switch between these two states by clicking on the *o* or the *x*. When the module is open, it accumulates in a list the result of each evaluation of the patch connected to its input, or the value (in the form of a list) at its input. When it is closed it returns the accumulated list. The module is reinitialized by a change of state from closed to open. **accum** takes a list of maximum length 400 elements, which is the default value. This value can be modified by the opening of the optional input to the **accum** module, by clicking on the *E* found on the right. When the list reaches its maximum value, the resulting list begins to wraparound in a circular fashion, writing over old values.

lst-ed

Syntax

(pw::lst-ed list)

Input

list	A list (at many levels)

Output

returns	Returns *list*

This module is an editor for graphic tables. To open the editor window associated with this module, click twice within the module (but not on the input window!). The module has two types of functionalities, depending on whether it is locked or not. If it is not locked, it accepts the input data and sends it to the output . You can also edit the same data in the editor window. If it is locked, it accepts no input from outside the module, but you can edit data in the editor window. To obtain more information, type *h* with the module open. To close it type return.

Upon opening, the editor presents a small two-line, two-column table. The commands to edit the table are the following:

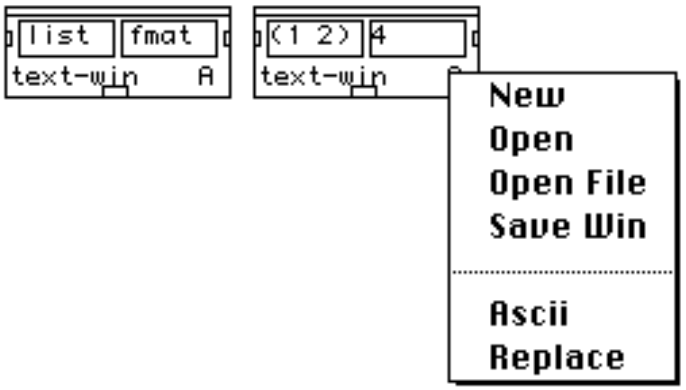
- To edit each cell, click twice on the cell, type the desired values and then hit return.
- To add cells (lines or columns), it is necessary to first select a cell (i.e., position the cursor on the cell and click once).

After selecting the cell one has access to the following commands:

- To add a cell before the current cell: hit '->'
- To add a cell after the current cell: hit '<-'
- To add a cell above: hit Up-arrow
- To add a cell below: hit Down-arrow
- To cut a cell, select it and hit "BACK-SPACE" ('<-' above the return key)
- To add a column in front: SHIFT '->'
- To add a column in back: SHIFT '<-'
- To add a line above: SHIFT Up-arrow
- To add a line below: SHIFT Down-arrow

The addition of cells, lines, or columns causes a cell to be opened, which must be edited. Type the values desired and then type Return immediately. The evaluation of this module returns a list of lists where each sublist corresponds to a column. The first elements of each column are the list headings. Entering a list of lists in the module formats the table anew. It is possible to edit either numbers or symbols but it is not possible to edit parentheses! Note: It is possible to save the module, independently of the patch, by choosing the **Save** option in the front menu. To open the front menu, move the cursor to the *A* and click once.

text-win



Syntax

(Pw::text-win data fmat)

Input

data	A list
fmat	A fixnum > 0

Output

returns	The content of the **text-win**

This module lets one create and communicate with a Lisp text window. The new window is created by choosing **New** in the front menu (click on *A* at right to open the menu). The new window appears and makes PatchWork switch to Lisp.

To return to PatchWork click on the PatchWork window, select PW in the Apps menu or type Command-1. It is possible to write in this window, either directly (returning to Lisp) or by entering data by the input at the left of the module. The connection of a patch at the left input *list* and the evaluation of the module **text-win** also makes a window and switches to Lisp.

To save data, evaluate the **text-win** module.

The front menu presents six options:

New	creates a new window and links it to the **text-win** module
Open	opens (selects) the window linked to the module.
Open-file	opens a Macintosh dialog box for retrieving a text window to be linked to the module.
S**ave-win**	lets one save the current window in a file,
Ascii *or* Lisp	an option that selects the format of the data written on the window.

When this module is selected the default format is Lisp expressions. In this case the third option is **Ascii**, which lets one change the format of the data. If the **ASCII** option is chosen, the third option is Lisp, which makes it possible to return to the format of Lisp expressions.

Replace *or* **Add**	an option that selects whether adding or replacing data on the linked window. When this module is selected the default option is **Add** , this meaning that input data is added to the end of the window. In this case the option is **Replace**. If the **Replace**

option is chosen ,'input data replaces all previous contents of the window, and in this case the option is **Add** .

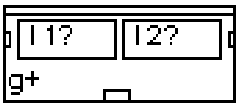
The second entry of the module *fmat* (at the right) determines how many elements will be written per line in the corresponding text window. For further information; type *h* with the selected module open.

Arithmetic Modules

All modules that start with the letter *g* act on *trees,* that is, on lists of lists on many levels.

The inputs of arithmetic modules can be either simple arguments or lists, on many levels. When the inputs are lists, the principle of the shortest list obtains. That is, for two lists of different sizes and levels, the module takes the shortest list.

g+



Syntax

(epw::g+ l1? l2?)

Input

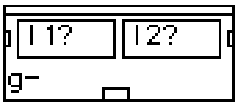
l1?	float or fixnum or list
l2?	float or fixnum or list

Output

returns	The sum of *l1?* and *l2?*

Sums two of numbers or trees.

g-



Syntax

(epw::g- l1? l2?)

Input

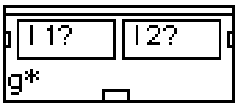
l1?	Float or fixnum or list
l2?	Float or fixnum or list

Output

returns	The difference of l1? and l2?

Reckons the difference of two numbers or trees.

g*



Syntax

(epw::g* *l1? l2?*)

Input

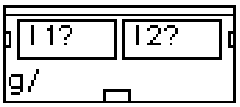
l1?	Float or fixnum or list
l2?	Float or fixnum or list

Output

returns	The product of *l1?* and *l2?*

Determines the product of two numbers or trees.

g/



Syntax

(epw::g/ l1? l2?)

Input

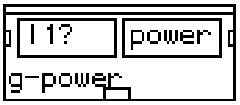
l1?	Float or fixnum or list
l2?	Float or fixnum or list

Output

returns	The quotient of *l1?* by *l2?*

Obtains the quotient of two numbers or trees.

g-power



Syntax

(epw::g-power l1? power)

Input

l1?	Float or fixnum or list
power	Float or fixnum or list

Output

returns	*l1?* to the power of *power*

Calculates *l1?* taken to the power of *power.*

g-exp

Syntax

(epw::g-exp l ?)

Input

l?	Float or fixnum or list

Output

returns	Exponential of *l?*

Computes the exponential of a number or a tree.

g-log

Syntax

(epw::g-log l ?)

Input

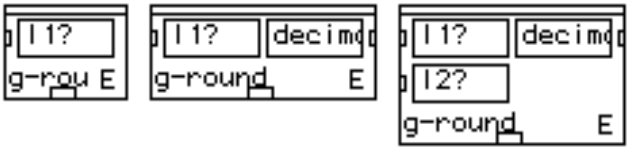
l?	Float or fixnum or list

Output

returns	The log of *l ?*

Calculates the natural logarithm of a number or a tree.

g-round



Syntax

(epw::g-round l1? &optional decimals l2?)

Input

l1?	Float or fixnum or list

Optional Input

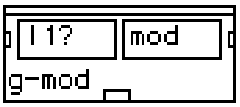
decimals	An integer or float
l2?	Float or fixnum or list

Output

returns	*l1?* with *decimal* decimal places.

Rounds a number or tree. This module allows many operations, since it is extendible. (See the letter *E* on the module.) The input *decimals* sets the choice of number of decimal places to round to. *l2?* specifies division of this rounded number by a second before rounding.

g-mod



Syntax

(epw::g-mod l1? mod)

Input

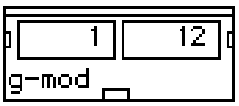
l1?	Float or fixnum or list
mod	Float or fixnum or list

Output

returns	The remainder of an integer division between two numbers *11?* and *mod*

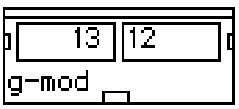
Calculate the number that is congruent modulo *mod* to *l1?*, or the remainder of an integer division (Euclidean division between two numbers *l1?* and *mod*.)

For example



will return

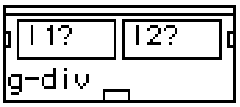
? PW->1 , and



will return

? PW->1.

g-div



Syntax

(epw::g-div l1? l2?)

Input

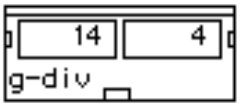
l1?	Float or fixnum or list
l2?	Float or fixnum or list

Output

returns	Integer division of *l1?* by *l2?*

Divides two numbers or trees, according to Euclidean division (integer division).

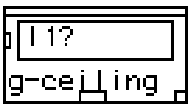
For example



will return

? PW->3.

g-ceiling



Syntax

(epw::g-ceiling l1?)

Input

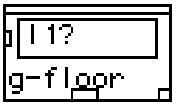
l1?	Float or fixnum or list

Output

returns	*l1?* rounded to the larger integer

Approximates a number or tree to the nearest larger integer .

g-floor



Syntax

(epw::g-floor l1?)

Input

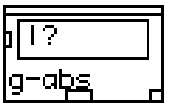
l1?	Float or fixnum or list

Output

returns	*l1?* rounded to the smaller integer

Performs truncation of number or tree, rounded to the smaller integer.

g-abs



Syntax

(epw::g-abs l ?)

Input

l?	Float or fixnum or list

Output

returns	The absolute value of *l?*

Calculates the absolute value of a number or a tree.

g-min

Syntax

(epw::g-min list)

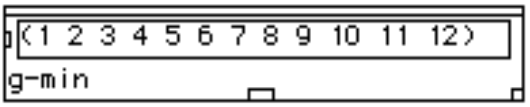
Input

list A list

Output

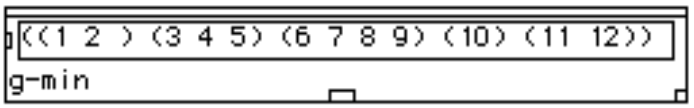
returns The minimum value(s) of the leaves of each deepest level subtree of *list.*

Returns a tree of the minimum value(s) of the leaves of each deepest level subtree. Trees must be well-formed: the children of a node must be either all leaves or all non leaves. For example,



will return

? PW->1, and



will return

? PW->(1 3 6 10 11)

g-max

Syntax

(epw::g-max list)

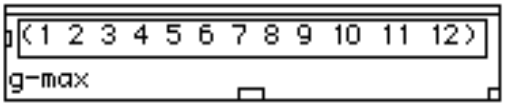
Input

list A list

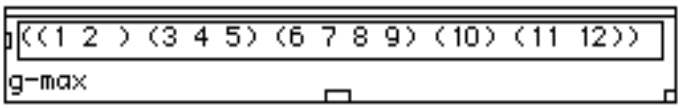
Output

returns The maximum value(s) of the leaves of each deepest level subtree of *list.*

Returns a tree of the maximum value(s) of the leaves of each deepest level subtree. Trees must be well-formed: The children of a node must be either all leaves or all nonleaves. For example



will return
? PW->12, and



will return
? PW->(2 5 9 10 12)

g-random

Syntax

(epw::g-random low high)

Input

low	Fixnum or float or list
high	Fixnum or float or list

Output

returns	Returns a random value between *low* and *high* inclusive.

Calculates a random value between *low* and *high* inclusive. *Low* and *high* can be trees.

g-average

Syntax

(epw::g-average xs weight)

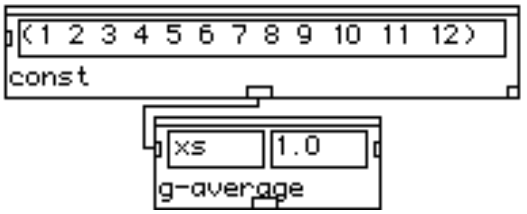
Input

xs	A list
weight	A number (fix or float) or a list

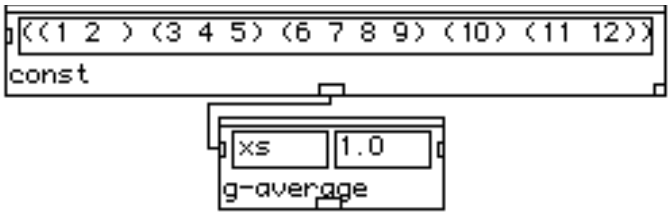
Output

returns	Average value of *xs,* weighted by linear *weights*

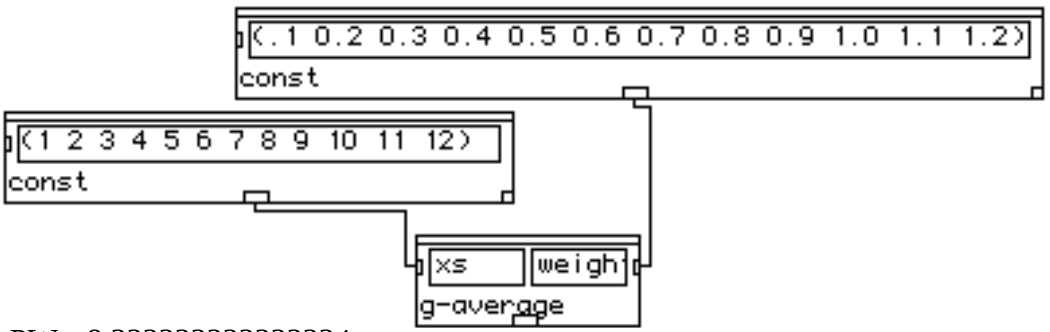
Determines the average value of *xs,* weighted by linear *weights* or 1., *xs* and *weights* can be trees. Trees must be well-formed. That is, the children of a node must be either all leaves or all nonleaves.



will return ? PW->6.5



will return ? PW->(1.5 4.0 7.5 10.0 11.5) , and



will return ? PW->8.333333333333334.

Num-Series Modules

arithm-ser

Syntax

(epw::arithm-ser begin step end)

Input

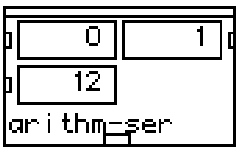
begin	An integer or float
step	An integer or float
end	An integer or float

Output

returns	An arithmetic series

Returns a list of numbers starting from *begin* to *end* with increment *step.* For example:

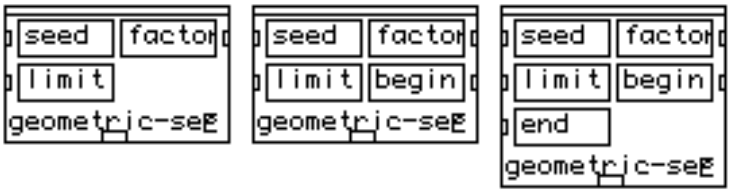
? (epw::arithm-ser 0 1 12)



returns

PW->(0 1 2 3 4 5 6 7 8 9 10 11 12)

geometric-ser



Syntax

(epw::geometric-ser seed factor limit &optional begin end)

Input

seed	An integer or float
factor	An integer or float
limit	fixnum

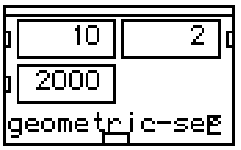
Optional Input

begin	An integer or float
end	An integer or float

Output

returns	A geometric series

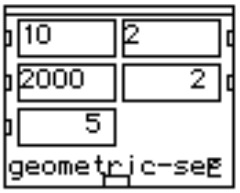
The **geometric-ser** module returns a geometric series of numbers in which the first element is *seed* and the multiplicative coefficient is *factor.* The *limit* parameter is the limit of this list. It is also possible to specify two parameters begin and end which delimit the calculation of the series. For example:



will return

? PW->(10 20 40 80 160 320 640 1280)

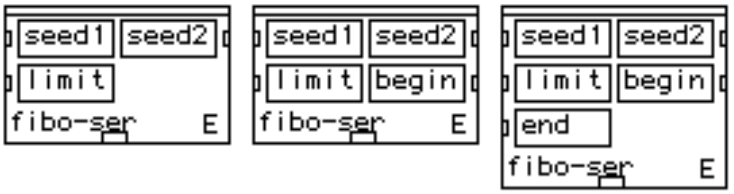
and if one sets begin to 2 and end to 5



one obtains

? PW->(40 80 160 320)

fibo-ser



Syntax

(epw::fibo-ser seed1 seed2 limit &optional begin end)

Input

seed1	An integer or float
seed2	An integer or float
limit	A fixnum

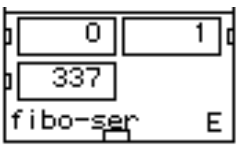
Optional Input

begin	An integer or float
end	An integer or float

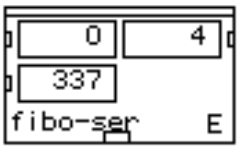
Output

returns	A Fibonacci series

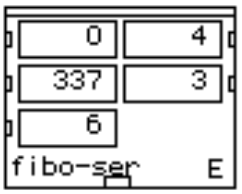
Returns a list of numbers in the Fibonacci series where the first element is *seed* and the additive factor for the two first steps is *seed2.* The *limit* parameter is the limit of this list. It is also possible to specify two parameters *begin* and *end* which delimit the calculation of the series. For example:



returns ? PW->(0 1 2 3 5 8 13 21 34 55 89 144 233),



returns ? PW->(0 4 8 12 20 32 52 84 136 220), and



returns ? PW->(12 20 32 52).

g-scaling

Syntax

(epw:: g-scaling vals? minout maxout &optional minin maxin)

Input

vals?	A fixnum or float or list
minout	An integer or float
maxout	An integer or float

Optional **Inputs**

minin	An integer or float
maxin	An integer or float

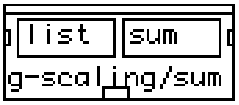
Output

returns	The list *vals?* rescaled

Replaces all the *vals?* between the minimum value of the list and the maximum value of the list, by the values proportionally placed between *minout* and *maxout.* If the list in question is a part of a larger list, or *vals?* is a variable that takes a value within a known interval, one can specify the minimum and maximum values by opening two optional windows by double-clicking on 'E' at the right of the module.

For lists of lists, trees must be well-formed. That is, the children of a node must be either all leaves or all non-leaves. Sub-lists will be scaled.

g-scaling/sum



Syntax

(epw:: g-scaling/sum list sum)

Input

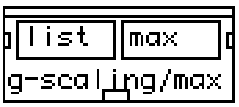
list	A list
sum	A fixnum list

Output

returns	The list *list* rescaled in function of *sum*

Scales *list* (which may be a tree) so that its sum becomes *sum.* Trees must be well-formed. The children of a node must be either all leaves or all nonleaves. When *sum* is a list, *list* should contain as many sub-lists as there elements in *sum*..

g-scaling/max



Syntax

(epw:: g-scaling/max list sum)

Input

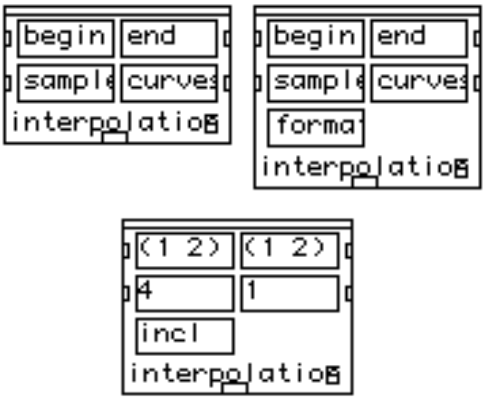
list	A list
max	A fixnum list

Output

returns	The list *list* rescaled in function of *max*

Scales *list* (may be a tree) so that its maximum value becomes *max.* Trees must be well-formed. The children of a node must be either all leaves or all nonleaves. When *max* is a list, *list* should contain as many sub-lists as there elements in *max*..

interpolation



Syntax

(epw:: interpolation begin end samples curves &optional format)

Input

begin	A list
end	A list
samples	A fixnum
curves	A fixnum or float or list

Optional Input

format	Menu

Output

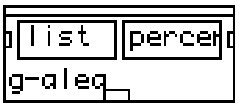
returns	A list of interpolations between *begin* and *end*

Interpolates two lists of the same length. (If the lists are not the same length, the operation produces only the number of terms equal to the shorter list.) *begin* and *end,* in *samples* steps (i.e., *samples* is the number of steps). *curve* is an optional value that selects the type of interpolation:

1 = straight line,
< 1 = convex
> 1 = concave

If *format* is 'incl' the two extremes are included in the output . If format is 'excl' they are excluded.

g-alea



Syntax

(epw:: g-alea list percent)

Input

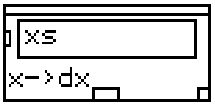
list	Fixnum or float or list
percent	Fixnum or float or list

Output

returns	A *list* with *percent* percent of noise

Apply a uniform random function to the leaves of the tree *list* of a depth according to a *percent* indicated.

x->dx



Syntax

(epw:: x->dx xs)

Input

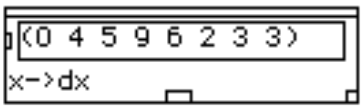
xs	A list

Output

returns	A list of the intervals between the contiguous values of *xs*

Returns the list of the intervals between the contiguous values of a list *xs.*

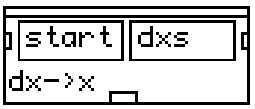
For example



will return

? PW->(4 1 4 -3 -4 1 0)

dx->x



Syntax

(epw:: dx->x start dxs)

Input

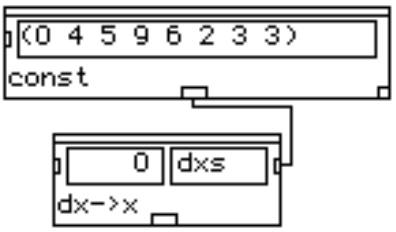
start	A list
dxs	A list

Output

returns	An accumulated list

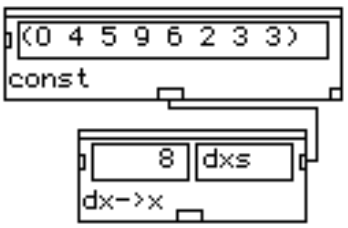
Constructs a list of numbers from *start* with the consecutive intervals of *dxs.*

For example



will return

? PW->(0 0 4 9 18 24 26 29 32), and

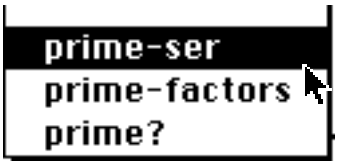


will return

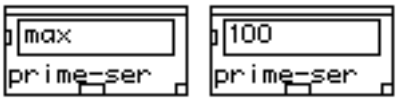
? PW->(8 8 12 17 26 32 34 37 40)

New modules in Kernel>Num-series

Three modules have been added by version 2.5 in the **Kernel>Num-series** .



prime-ser



Syntax

(epw::prime-ser max)

Input

max fixnum 0

Output

returns list

Returns a list of prime numbers between 0 and max.

Pw->(1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97)

Prime-factors

Syntax

(epw::prime-factors number)

Input

number	fixnum	0

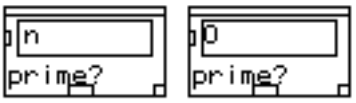
Output

returns	list

Factories number in prime factors. Ex: number=500

? PW->((2 2) (5 3)) which means $2^2 * 5^3$.

Prime?



Syntax

(epw::prime? n)

Input

n	fixnum	0

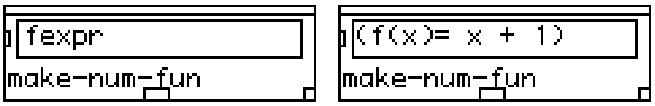
Output

returns	A boolean value "t" or "nil"

Checks if n is prime. If yes, then "t", else "nil".

Function Modules

make-num-fun



Syntax

(CLPF-Util::make-num-fun fexpr)

Input

fexpr A list

Output

returns A Lisp function object

Creates a Lisp function object from the expression *fexpr,* which is basically an infix expression. When *fexpr* begins with something like (*f* (*x*)= ...), the formal arguments are taken from the given list, otherwise they are taken from the body of *fexpr* and collected in the order they appear in it. Local variables are automatically handled. **make-num-fun** has two different possible syntaxes.

1. The standard Lisp syntax, that is: (*f* (*x*) = (- (* *x x*) *x*)))

2. C language syntax, that is: (*f* (*x*)= (*x* * *x*) - *x*).

Note that the only difference between this notation and standard C notation is that a space must be put between operators. The variable name definition at the beginning of the function:

(*f* (*x*) = ...)

is optional. If it is not included by the user, the program figures out which variables are involved.

The function definition can support more than 1 variable. For instance:

(f(x y) = (+ (* 4 x) (* 3 y)))

sample-fun

Syntax

(epw::sample-fun fun xmin step xmax)

Input

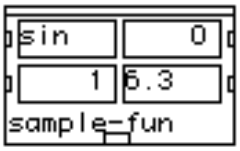
fun	A Lisp function object
xmin	An integer or float
step	An integer or float
xmax	An integer or float

Output

returns	A list of values of *fun* beetwen *xmin* to *xmax* with step *step*

Returns the list of values of *fun* from *xmin* to *xmax* with step *step.*

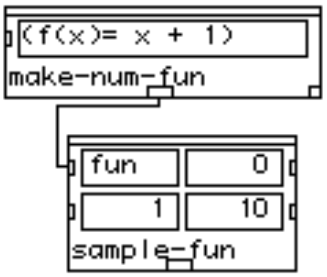
For example:



will return

? PW->(0.0 0.8414709848078965 0.9092974268256817 0.1411200080598672 0.7568024953079282 -0.9589242746631385 -0.27941549819892586)

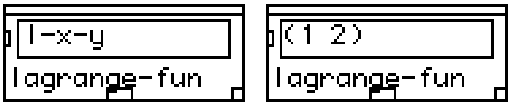
and



will return

? PW->(1 2 3 4 5 6 7 8 9 10 11)

lagrange



Syntax

(epw::lagrange l-x-y)

Input

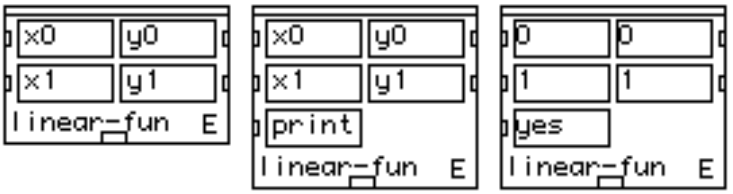
l-x-y	A list

Output

returns	A Lisp function object

Returns a Lagrange polynomial (a Lisp function object) defined by the points of list *l-x-y*.

linear-fun



Syntax

(epw::linear-fun x0 y0 x1 y1 &optional print)

Input

x0	An integer or float
y0	An integer or float
x1	An integer or float
y1	An integer or float

Optional Input

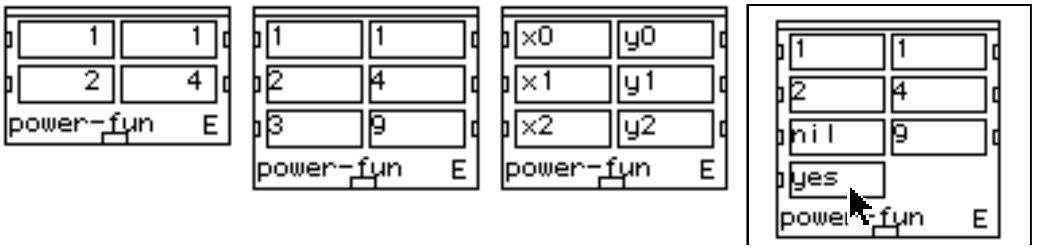
print	A menu, "yes" (print), "no" (not print)

Output

returns	A Lisp function object

Calculate the parameters of the equation $y = a x + b$ as a function of the two points (*x0,y0*) (*x1,y1*). The optional parameter *print* lets one print the function.

power-fun



Syntax

(epw:: power-fun x0 y0 x1 y1 &optional x2 y2)

Input

x0	An integer or float
y0	An integer or float
x1	An integer or float
y1	An integer or float

Optional Input

x2	An integer or float
y2	An integer or float
print	A menu, "yes" (print), "no" (not print), for the corresponding function
yes	Allows the display of the corresponding function

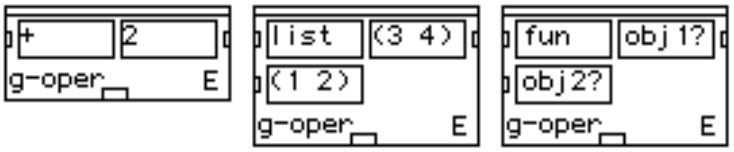
Output

returns	A Lisp function object

Calculate the parameters of the equation $y = a\,x^b + c$ or $y = a\,x^b$ as a function of the points (*x0,y0*) (*x1,y1*) and (optional) (*x2,y2*) and create the corresponding function, either $y = ax^b$ (for two pairs of points) or $y = a\,x^b + c$ (for three pairs of points).

The optional parameter print lets one print the function.

g-oper



Syntax

(epw::g-oper fun obj1? &optional obj2?)

Input

fun	A Lisp function (see **make-num-fun**)
obj1?	A fixnum float list

Optional Input

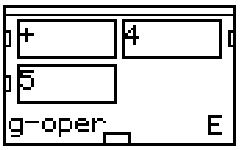
obj2?	A fixnum float list

Output

returns	The function *fun* applied to *obj1?* and *obj2?*

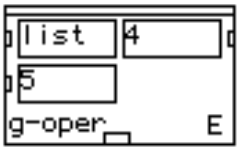
Applies *fun* to leaves of trees of *obj1?* and (optionally) *obj2?*. *fun* may be a Lisp function (**list**, +, *, **cons**, etc.) or a function object created by the **make-num-fun** box.

For example:



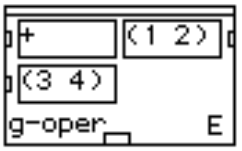
will return

? PW->9



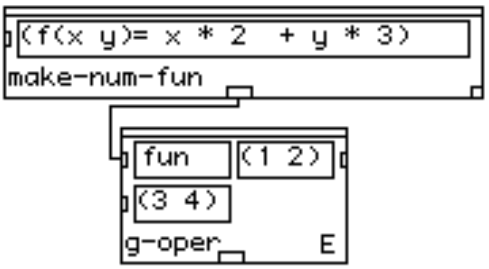
will return

? PW->(4 5)



will return

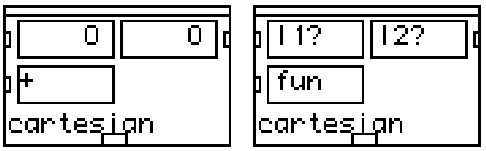
? PW->(4 6), and



will return

? PW->(11 16).

cartesian



Syntax

(epw::cartesian l 1? l 2? fun)

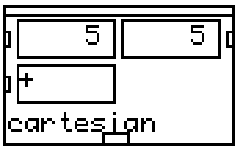
Input

l1?	Fixnum or float or list
l2?	Fixnum or float or list
fun	A Lisp function (see **make-num-fun**)

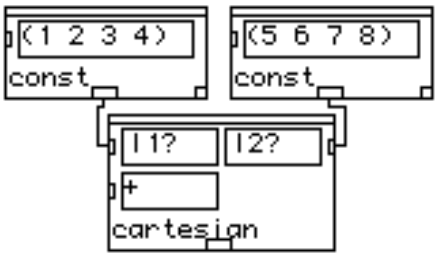
Output

returns	A Cartesian product between *l1?* and *l2?*

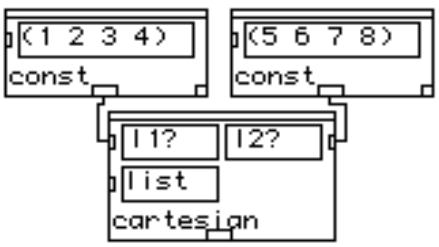
Applies the function *fun* to elements of *l1?* and *l2?* considered as matrices. Like **g-oper** *fun* may be a Lisp function (**list**, +, *, **cons**, etc.) or a function object created by the **make-num-fun** box .The result is a cartesian product of *l1?* by *l2?*.



will return ? PW->((10))

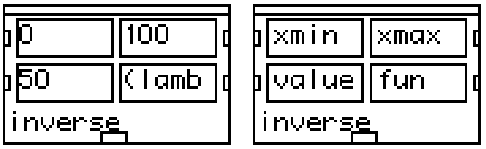


will return ? PW->((6 7 8 9) (7 8 9 10) (8 9 10 11) (9 10 11 12)) and



will return ? PW-> (((1 5) (1 6) (1 7) (1 8)) ((2 5) (2 6) (2 7) (2 8)) ((3 5) (3 6) (3 7) (3 8)) ((4 5) (4 6) (4 7) (4 8)))

inverse



Syntax

(epw::inverse xmin xmax value fun)

Input

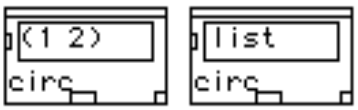
xmin	An integer or float
xmax	An integer or float
value	An integer or float
fun	A Lisp function (see **make-num-fun**)

Output

returns	A value of *x* such that *fun*(*x*)= *value.*

Binary searches *x* in the interval [*xmin,xmax*] , such that *fun*(*x*)= *value. fun* must be either increasing or decreasing in the interval.

circ



Input

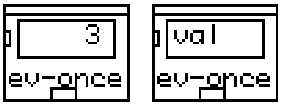
list	A list

Output

returns	The next element of *list*

Circular buffer that accepts lists as input. The module is reinitialized at each mouse click on its output; this returns the first element of the *list.* Indirect evaluation of **circ** (select the module and type 'v') causes the list to circulate around the buffer.

ev-once



Input

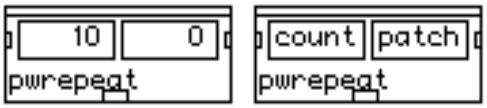
val	Any data type (integer, float, string, list, etc.)

Output

returns	*val*

This module assures that all the modules that are connected to its output receive the same values.

pwrepeat



Input

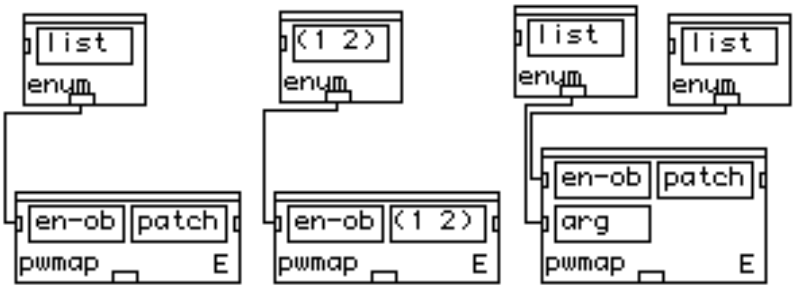
count	A fixnum
patch	Any data type (integer, float, string, list, etc.)

Output

returns	A list containing *count* repetitions of *patch*

The pwrepeat box lets one evaluate the input *patch count* times and to collect data as a list. The first input *count* tells how many times the second input *patch* is evaluated.

pwmap



Input of enum

list A list

Output of enum

returns The next element of *list*

Input of pwmap

en-ob The output of **enum**

patch A patch

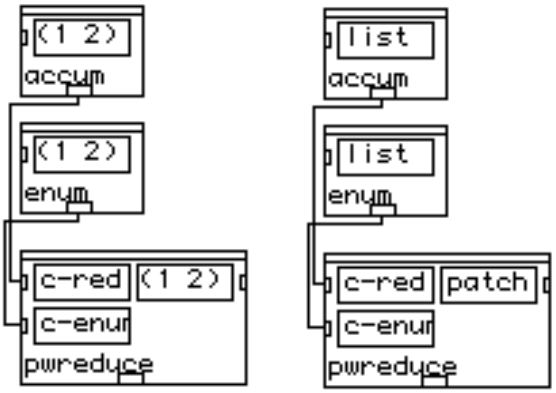
arg The output of **enum**

Output of pwmap

returns A list containing the evaluation of *patch* for each value of the list in enum**.**

This group of modules creates a list starting with the evaluation of a patch that takes into account all the elements of a list connected to the module **enum**. The output of the patch must be connected to the input *patch* of **pwmap**. Since this module is extensible, it is possible to control many lists, by opening the inputs *arg* to which is connected a module **enum**. In the case of lists ofvarious sizes, **pwmap** will select the shortest one.

pwreduce



Input of enum

list	A list

Output of enum

returns	The next element of *list*

Input of accum

list	A list

Output of accum

returns	The last computed result

Input of pwreduce

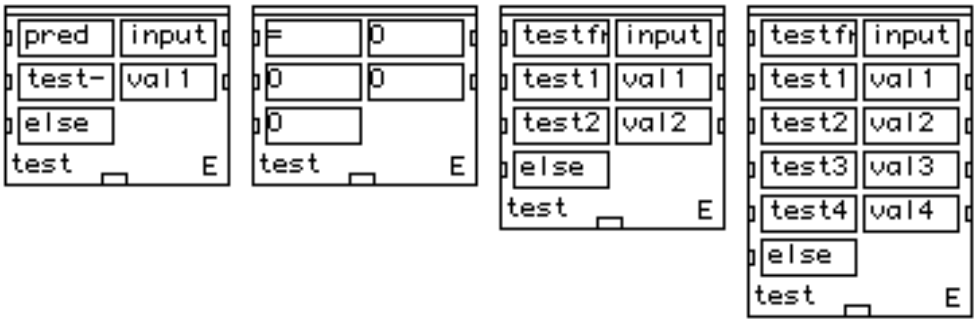
c-red	The output of **accum**
c-enum	The output of **enum**
patch	A patch

Output of pwreduce

returns	The last evaluation of *patch*

The **pwreduce** module applies a function to a list of elements. The list is entered in **enum** and the function is defined by patch. **accum** gives an initial value for the function, and serves to accumulate the results of the function for each step in the loop. In other words, pwreduce repeatedly applies a function - patch - to each of a list of elements - **enum** - and puts the results of each successive evaluation in accum. The output is the last computed result.

test



Input

testf	A predicate (=, >, <, etc.) or any function which takes two arguments
input	Any data type that is accepted by the predicate in question; can also be a list.
est1	Any data type accepted by the predicate in question
val	A patch
else	A patch

Output

returns	The evaluation of *val1* if *testf* (or *pred*) does not return nil, or the evaluation of *else* if *testf* (or *pred*) returns nil, or a list of the preceding if *input* is a list

The module **test** applies a test function *pred* (or *testf* if other inputs are open) using input and test1 as arguments. If the test succeeds val1 is evaluated, otherwise else is evaluated. This module can be extended to include multiple cases. In this case input is compared with test1, then test2, test3, etc.; as soon as the test function succeeds, the corresponding val patch will be evaluated. For example if test1 and test2 return nil, but test3 returns a true, val3 is evaluated. In this case test4 and test5 will never be considered. If all tests fail, then the else patch is evaluated. If input is a list, a list is returned with the results of applying the module's result to each element of the input list.

trigger

Input

patch1	Any data type
patch2	Any data type

Optional Input

patch	Any data type

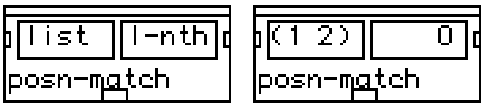
Output

returns	nil

This extensible module launches the evaluation of many patches in sequence. The sequence of evaluation is equal to the sequence of the inputs.

List Modules

posn-match



Syntax

(epw::posn-match list l-nth)

Input

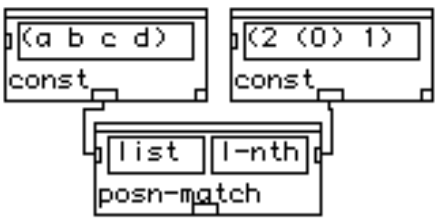
list:	A list
l-nth:	A number or list of numbers

Output

returns	*l-nth* with its values replaced by the corresponding elements of *list.*

l-nth: accepts a number or a list of numbers. Returns a copy of *l-nth* where each number is replaced by the corresponding element in the list *l.*

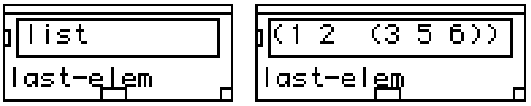
For example, if *list* is (a b c d) and *l-nth* is (2 (0) 1))



returns

? PW->(c (a) b) , where the list returned has the same structure as *l-nth*.

last-elem



Syntax

(epw::last-elem list)

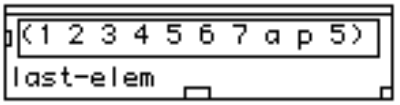
Input

list	A list of anything

Output

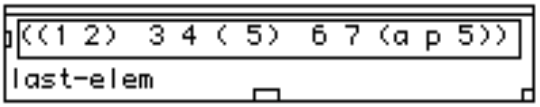
returns	The last element of *list* as an atom

Returns the last element of *list.*



will return

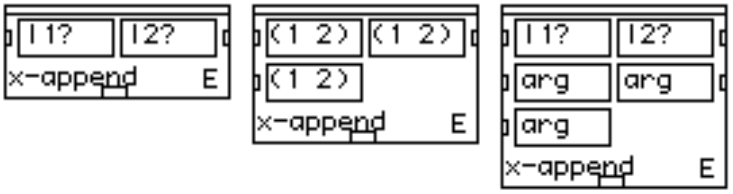
? PW->5, and



will return

? PW->(a p 5)

x-append



Syntax

(epw::x-append l1? l2? &rest arg)

Input

l1?	Any data type
l2?	Any data type

Optional Input

arg	Any data type

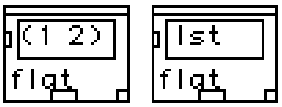
Output

returns	A set of lists merged

Appends lists or atoms together to form a new list. This box can be extended.

This modules removes one level of parentheses to the lists that it is forming.

flat



Syntax

(epw::flat list)

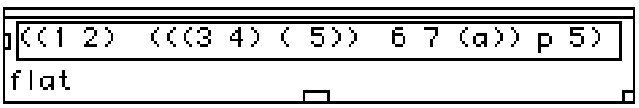
Input

lst	A list of any data types

Output

returns	A copy of *list* with all elements (including those which were in embedded lists) on the same level

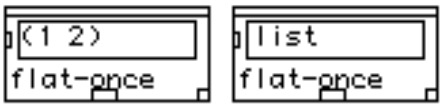
Takes off every parenthesis. There should be no dotted pair.



will return

? PW->(1 2 3 4 5 6 7 a p 5)

flat-once



Syntax

(epw::flat-once list)

Input

list	A list of any data types.

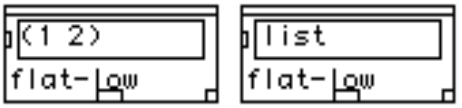
Output

returns	A copy of *list* less the first level of parenthesis

Flattens the first level of a list of lists.

For example:
'(((1 2 3) (4 5 6)) ((7 8 9) (10 11 12)))
becomes:
((1 2 3) (4 5 6) (7 8 9) (10 11 12)).

flat-low



Syntax

(epw::flat-low list)

Input

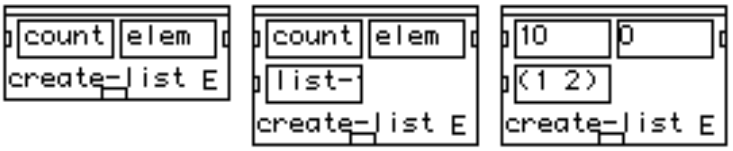
list A list of any data types.

Output

returns A copy of *list* less the lowest level of parenthesis

Flattens lowest level sublists.
For example:
'(((1 2 3) (4 5 6)) ((7 8 9) (10 11 12)))
becomes:
((1 2 3 4 5 6) (7 8 9 10 11 12)) .

create-list



Syntax

(epw::create-list count elem &optional list-fill)

Input

count	An integer.
elem	An integer, float, string, or list.

Optional Input

list-fill	A list

Output

returns	A list of *count* instances of *elem*

Returns a list of length *count* filled with *elem* (if no list-fill, i.e. if the box is not extended) or duplicates the elements of *list-fill* until its length equals *count* (if *list-fill*, i.e. if the box is extended).

will return

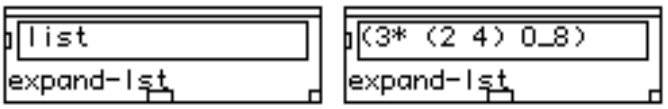
? PW->((a 2) (a 2) (a 2) (a 2) (a 2))

and

will return

? PW->(a a a 2 2)

expand-lst



Syntax

(epw::expand-list list)

Input

list	A list of commands

Output

returns	An expanded list

Expands a list by one (or both) of the following:

1. Repeating each item *number* times following a pattern of the form: *number**

2. Creating a sequence of numbers going from *n* to *m* by steps of *k*, indicated by the pattern *n_msk*. A step of 1 can be omitted.

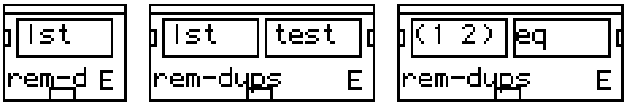
For example the list (3* (2 4) 0-8) returns

(2 4 2 4 2 4 0 1 2 3 4 5 6 7 8),

and the list (2* (a z 2*(4 12) (1-5)) 0_16s2) returns

(a z 4 12 4 12 (1 2 3 4 5) a z 4 12 4 12 (1 2 3 4 5) 0 2 4 6 8 10 12 14 16).

rem-dups



Syntax

(epw::rem-dups lst &rest test)

Input

lst A list

Optional Input

test A logic predicate

Output

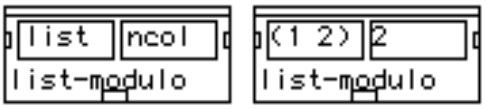
returns *lst* without repetitions

The **rem-dups** module removes repetitions of elements in *lst,* according to *test* (if the second input is open by clicking on 'E'). *test* must be commutative.

For example, the list (this this is my list list) returns (this is my list).

Note that the last occurrence of a repeated element in a list is preserved; thus, the list: (1 2 3 1 4) returns (2 3 1 4). Returns a copy of *lst.*

list-modulo



Syntax

(epw::list-modulo list ncol)

Input

list	A list.
ncol	A positive integer

Output

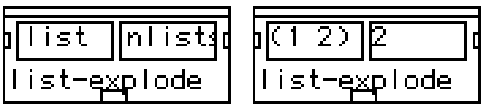
returns	A list of lists containing elements modulo *ncol* according to their position in the *list.*

The **list-modulo** module groups elements of a list that occur at regular intervals, and returns these groups as lists. *ncol* defines the interval between group members.

For example, if we take the list (1 2 3 4 5 6 7 8 9) and give 2 for ncol, the result is ((1 3 5 7 9) (2 4 6 8)). In other words, every second element starting with the first, and then every second element starting with the second.

If the number of *ncol* exceeds the number of elements in the list, the remaining lists are returned as nil. In effect, list-modulo divides *list* into *ncol* sublists containing elements modulo *ncol* according to their position in the list.

list-explode



Syntax

(epw::list-explode list nlist)

Input

list	A list.
nlist	a positive integer

Output

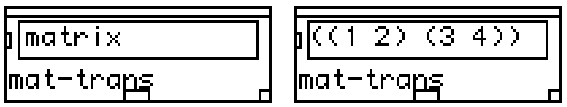
returns	*list* subdivided into *nlist* sublists of consecutives elements

The **list-explode** module divides a list into *nlist* sublists of consecutives elements.

For example, if list is (1 2 3 4 5 6 7 8 9), and ncol is 2, the result is ((1 2 3 4 5) (6 7 8 9)), if list is (1 2 3 4 5 6 7 8 9), and ncol is 5, the result is: ((1 2) (3 4) (5 6) (7 8) (9)).

If the number of divisions exceeds the number of elements in the list, the remaining divisions are returned as the last element of the list..

mat-trans



Syntax

(epw::mat-trans matrix)

Input

matrix	A list of lists.

Output

returns	*Matrix* with rows and columns interchanged

mat-trans transposes a matrix. That is, it interchanges rows and columns.

Thus for example, (mat-trans '((1 2) (5 7))) returns the list ((1 5) (2 7)), or if *matrix* is ((1 2) (3 4) (5 6) (7 8) (9 10)) **mat-trans** returns ((1 3 5 7 9) (2 4 6 8 10)). **mat-trans** behaves as if the sublists of matrix were arranged vertically. Then a sublist is constructed for each column resulting from this arrangement.

The result is the list of all these sublists.

list-filter

Syntax

(epw::list-filter test val list)

Input

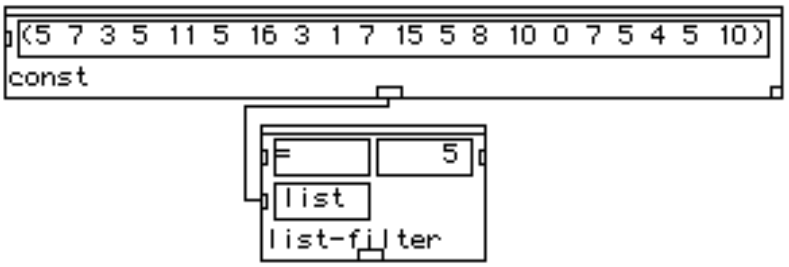
test	A predicate
val	Any data type
list	A list

Output

returns	*list,* without all instances of *val* according to a predicate *test*

The **list-filter** module removes elements from a *list* according to a predicate *test.* If the predicate is **eq**, all instances of *val* are removed from the list, regardless of their level.

If, for example, the predicate is **>**, all elements of list which are greater than *val* are removed. Note that *val* can be a string, but only if the predicate *test* can handle a string.



will return ? PW->(7 3 11 16 3 1 7 15 8 10 0 7 4 10) ,
with *test.* ">" **list-filter** will return ? PW->(5 3 5 5 3 1 5 0 5 4 5), and
with *test.* "<" **list-filter** will return ? PW->(5 7 5 11 5 16 7 15 5 8 10 7 5 5 10)

table-filter

Syntax

(epw::table-filter test val list numcol)

Input

test	A predicate
val	Any data type
list	A list of lists
numcol	An integer or a float

Output

returns	*list,* without any instances of *val* according to a predicate *test*

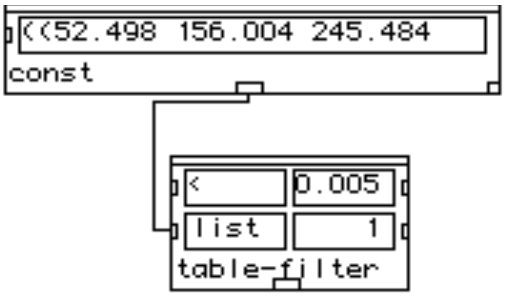
Choose from each of the sublists of *list* the elements for which the order number corresponds to each element of the sublist numcol that satisfies the condition *test val.*

Let's take for instance a spectral analysis from a marimba note.

((52.498 156.004 245.484 263.016 329.442 363.212 383.406 439.426 601.696 612.632 679.306 719.217 740.243 755.378 827.99 860.439 889.979 1048.307 1242.274 1316.117)
(0.004 0.002 0.01 0.494 0.009 0.002 0.004 0.002 0.005 0.112 0.004 0.001 0.002 0.0 0.001 0.002 0.001 0.267 0.001 0.003)
(1.686 0.429 1.296 1.958 1.635 0.265 1.132 1.379 2.61 12.189 2.714 1.058 1.942 0.436 2.269 0.934 1.309 5.909 2.084 1.993))

where the first sub-list represents frequencies, the second linear amplitudes and the third bandwidths.

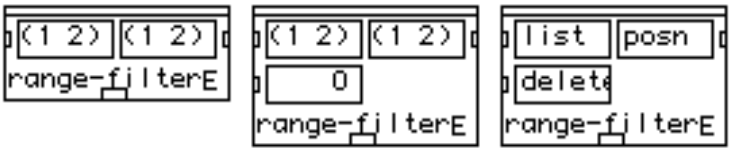
It is possible to filter this analysis result using the partials amplitudes, the analysis being stored in a const module.



When partials have amplitudes (numcol=1, i.e. second sub-list) smaller (test = <) than 0.005 (val=0.005), they will be erased. The result will be:

? PW->((245.484 263.016 329.442 601.696 612.632 1048.307) (0.01 0.494 0.009 0.005 0.112 0.267) (1.296 1.958 1.635 2.61 12.189 5.909))

range-filter



Syntax

(epw::range-filter list posn &rest delete)

Input

list	A list
posn	A list of positions

Optional Input

delete	1 (one) or 0 (zero)

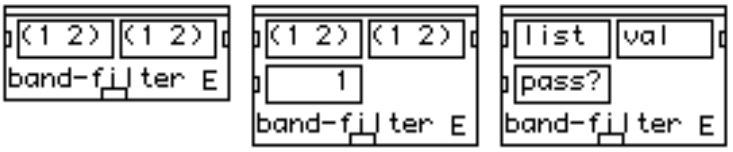
Output

returns	*list,* filtered by *posn* according with the argument *delete.*

The **range-filter** module selects from a list *list* all elements falling inside a range of given positions *posn.* The range of positions *posn* is given either as a list of two numbers or as a list of lists of two numbers. Each pair of numbers define an interval. If *delete* (the optional argument) is zero (the default) any element in list falling inside one of these intervals is selected. If *delete* is one, elements in *list* not falling inside one of those intervals is selected. Intervals are defined by position inside list.

For example, if *list* is (4 7 2 3 1 8 5) and *posn* is ((4 5) (0 2)), **range-filter** returns (4 7 2 1 8). On the other hand (if the third input is open), if *list* is (4 7 2 3 1 8 5), *posn* is ((4 5) (0 2)) and *delete* is 1, **range-filter** returns (3 5). The argument list can be a list of lists. In this case the described behaviour applies to each sublist.

band-filter



Syntax

(epw::band-filter list val &rest pass?)

Input

list	A list
val	A list of values

Optional Input

pass?	1 (one) or 0 (zero)

Output

returns	*list,* filtered by *val* according with the argument *pass?*

The **band-filter** passes or rejects all elements from *list* that fall inside a band of specified values of *val.* The range of values *val* is given either as a list of two numbers or as a list of lists of two numbers. Each pair of numbers define an interval. If *pass?* (the optional argument) is one (the default) only the element in list falling inside one of these intervals (of *val*) is selected. If *delete* is zero, elements in *list* not falling inside one of those intervals is selected. Intervals are defined by values inside list.

For example, if *list* is (2 4 6 8 10 12 14 16 18 20) and *val* is ((1 2 3) (7 9)), **band-filter** returns (2 8), (the default is one). On the other hand (if the third input is open), if *list* is (2 4 6 8 10 12 14 16 18 20), *val* is ((1 2 3) (7 9)) and *pass?* is 0 (zero), **band-filter** returns (4 6 10 12 14 16 18 20). The argument list can be a list of lists. In this case the described behavior applies to each sublist.

New modules in Kernel>list

Five modules have been addedby version 2.5 to **Kernel->list**.

interlock

Syntax

(patch-work::last-n list n)

Input

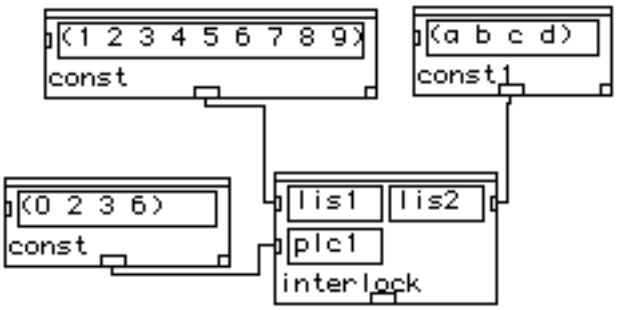
lis1	A list
lis2	A list
plc1	A list

Output

returns	*list*

Interleaving of *lis1* in *lis2* before elements placed at location *plc1*.

Example:



where

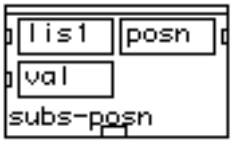
lis1 = (1 2 3 4 5 6 7 8 9)

lis2 = (a b c d)

plc1 = (0 2 3 6)

? PW->(a 1 2 b 3 c 4 5 6 d 7 8 9)

subs-posn



Syntax

(patch-work::subs-posn lis1 posn val)

Input

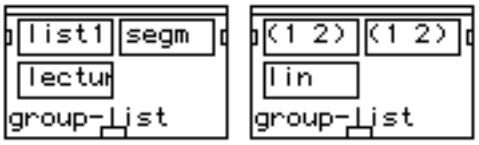
lis1	A list
posn	A list
val	A list

Output

returns	*list*

Substitution of all the elements of *lis1* which are located at posn by successive elements from list *val*.

group-list



Syntax

(patch-work::group-list list1 segm lecture)

Input

list1	A list
segm	A list or number
lecture	A menu

Output

returns	*list*

This module groups elements of list *list1* as sub-lists with lengths equal to elements of *segm*.

Example:

where

list1= '(0 1 2 3 4 5 6 7 8 9 10)

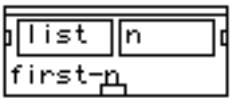
segm= '(2 4 3 2)

(group-list '(0 1 2 3 4 5 6 7 8 9 10) '(2 4 3 2))

PW->((0 1) (2 3 4 5) (6 7 8) (9 10)).

It is possible to choose linear or circular scanning of list1 through menu lecture.

first-n



Syntax

(patch-work::first-n list n)

Input

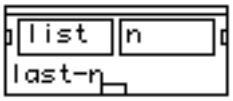
list	A list
n	fixnum 0

Output

returns	*list*

Returns the n first elements of list *list*.

last-n



Syntax

(patch-work::last-n list n)

Input

list	A list
n	fixnum 0

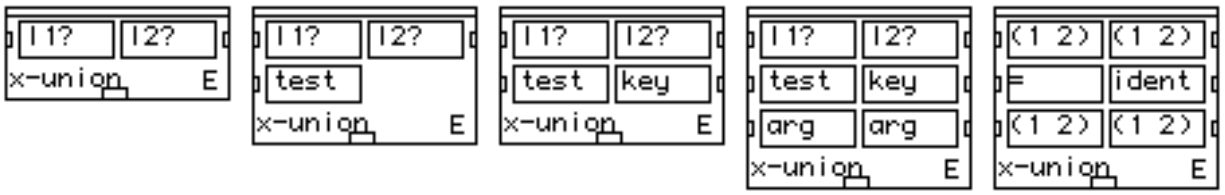
Output

returns	*list*

Returns the n last elements of list *list*.

Set Operations

x-union



Syntax

(epw::x-union l1? l2? &optional test key &rest args)

Input

l1?	Any data type
l2?	Any data type

Optional Input

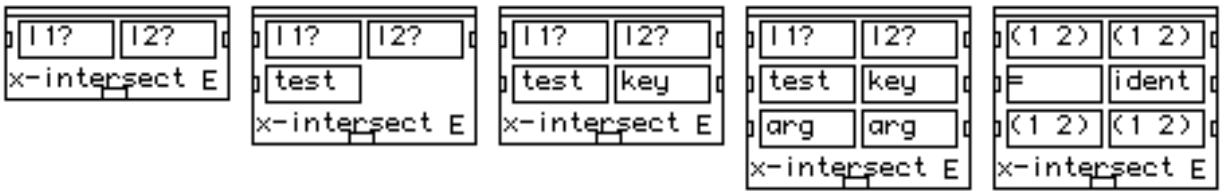
test	A predicate
key	A function
args	Other lists

Output

returns	A set of lists (*l1?, l2?* and *args*) merged

This box merges a set of lists, *l1?* and *l2?,* or elements into a single list, with no repetitions. If the optional *test* argument is added (remember that this module is an extended box) , the lists can be compared according to any predicate. Only elements in *l1?* that return true when compared with all the elements in *l2?* (according to the predicate), are returned in the result list. If the *key* argument (remember, this module is an extended box) is included, its function is evaluated using each of *l1?* elements as input, and the lists are then compared according to the test on the results of the function. Additional lists (click on 'E') can be compared using *arg .*

x-intersect



Syntax

(epw::x-intersect l1? l2? &optional test key &rest arg))

Input

l1?	Any data type
l2?	Any data type

Optional Input

test	A predicate
key	A function or method
arg	Other lists

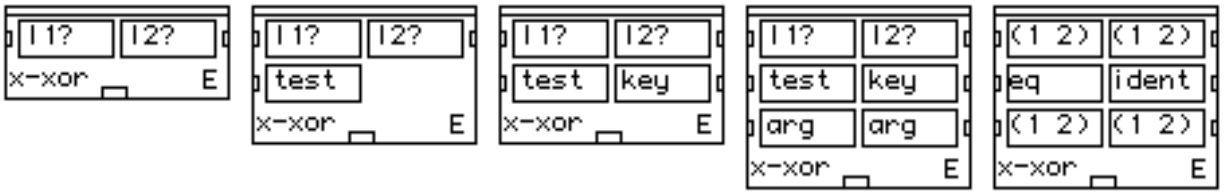
Output

returns	An intersection of lists (*l1?, l2?* and *args*)

This box returns a list of elements which are common to both *l1?* and *l2?.* If the optional *test* argument is added (remember that this module is an extended box) , the lists can be compared according to any predicate. Only elements in *l1?* which return true when compared with at least one element in *l2?* (according to the predicate), are returned in the result list. If the *key* argument (remember, this module is an extended box) is included, its function is evaluated using each of *l1?* elements as input, and the lists are then compared according to the test on the results of the function. Additional lists (click on 'E') can be compared using *arg.*

Beware that this operation is not commutative. For example:
(epw::x-intersect '(1 2 4 5 4) '(2 4)) will return -> (2 4 4)
(epw::x-intersect '(2 4) (1 2 4 5 4)) will return -> (2 4)

x-xor



Syntax

(x-xor l1? l2? &optional test key &rest arg))

Input

l1?	Any data type
l2?	Any data type

Optional Input

test	A predicate
key	A function or method
arg	Any data type

Output

returns	A list with the result of the XOR operation between *l1?, l2?* and *args*

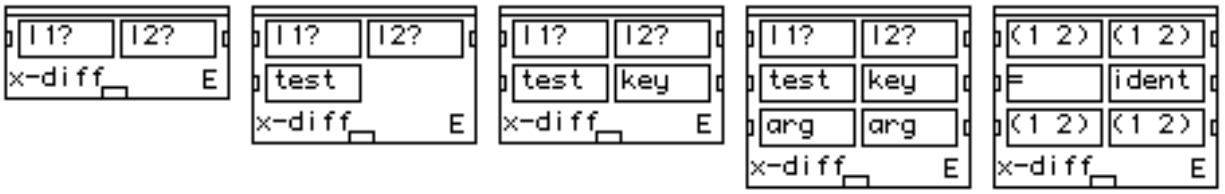
This box compares lists for elements that are present in either one or the other list (but not in both), and then returns them a list. If the optional *test* argument is added (remember that this module is an extended box), the lists can be compared according to any predicate. Only elements in *l1?* that return true when compared with all the elements in *l2?* (according to the predicate), are returned in the result list. If the *key* argument (remember, this module is an extended box) is included, its function is evaluated using each of *l1?* elements as input, and the lists are then compared according to the test on the results of the function. Additional lists (click on 'E') can be compared using *arg .*

Beware that this operation is not commutative. For example:

(epw::x-xor '(1 2 4 5 4 2 1) '(2 4 7)) will return -> (1 5 1 7)

(epw::x-xor '(2 4 7) (1 2 4 5 4 2 1)) will return -> (7 1 5 1)

x-diff



Syntax

(epw::x-diff l1? l2? &optional test key &rest arg))

Input

l1?	Any data type
l2?	Any data type

Optional Input

test	A predicate
key	A function or method
arg	Any data type

Output

returns	A list with all elements present in *I1?* and *args* but not in *I2?*

This box compares *I1?* to *I2?* and then returns all elements present in *I1?* but not in *I2?,* as a list. If the optional test argument is added (remember that this module is an extended box) , the lists can be compared according to any predicate. Only elements in *I1?* that return true when compared with all the elements in I2? (according to the predicate), will be returned in the result list. If the *key* argument (remember, this module is an extended box) is included, its function is evaluated using each of I1? elements as input, and the lists are then compared according to the test on the results of the function. Additional lists (click on 'E') can be compared using *arg.*

Beware that this operation is not commutative. For example:
(epw::x-dif '(1 2 4 5 4 2 1) '(2 4 7)) will return -> (1 5 1)
(epw::x-dif '(2 4 7) (1 2 4 5 4 2 1)) will return -> (7)

included?

Syntax

(epw::included? l1? l2? &optional test)

Input

l1?	Any data type
l2?	Any data type

Optional Input

test	A predicate

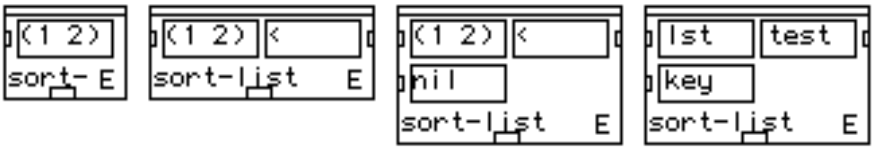
Output

returns	True if all of the elements of *l1?* are also elements of *l2?*. Otherwise, it returns nil.

This box compares two lists, returning true if all the elements in the first are also elements of the second. If the optional *test* argument is added (remember that this module is an extended box), the lists are compared globally according to any predicate. For example, if the predicate is >, the module returns true if all elements in the first list (*l1?*) are greater than at least one element in the second .

Combinatorial Modules

sort-list



Syntax

(epw::sort-list list &optional test key)

Input

lst	A list of numbers or lists

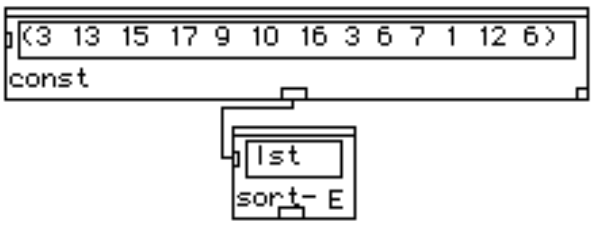
Optional Input

test	A predicate (<, >, =)
key	A function or method (see **make-num-fun**)

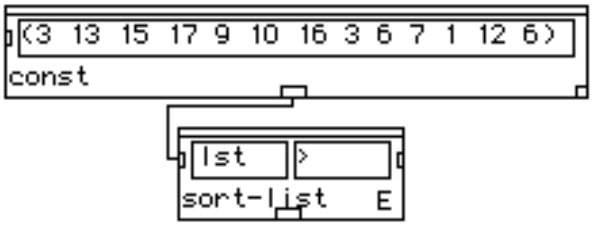
Output

returns	The same list with its contents rearranged

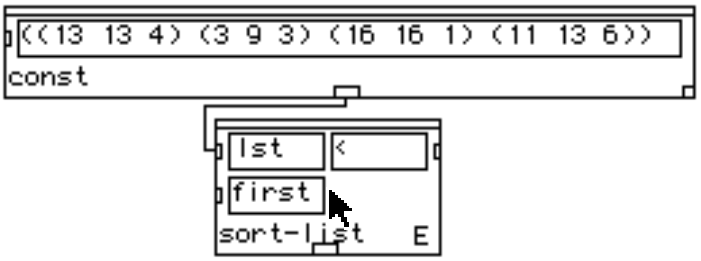
This module sorts a list. By default, the order of the sort is ascending, but since the module is extensible, you can open a second entry *test* to set the choice of order. If *test* is '>' the order is ascending, '<' indicates descending, and '=' keeps the order the same. One can also open a third input *key* for a function. The function *key* evaluates each element of the list *lst* and the result is then sorted according to the parameter *test.*



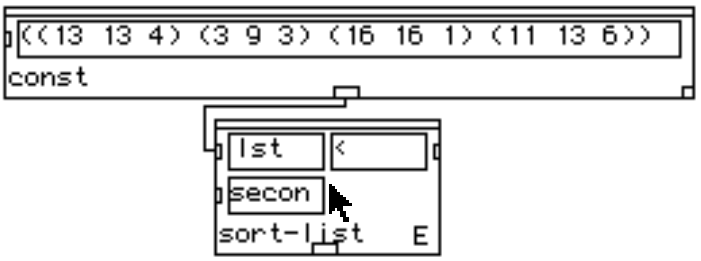
will return
? PW->(1 3 3 6 6 7 9 10 12 13 15 16 17),



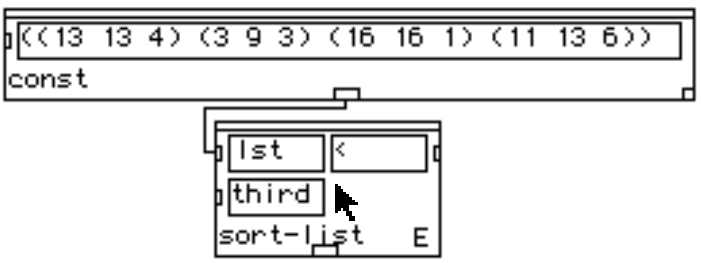
will return
? PW->(17 16 15 13 12 10 9 7 6 6 3 3 1),



will return
? PW->((3 9 3) (11 13 6) (13 13 4) (16 16 1)),

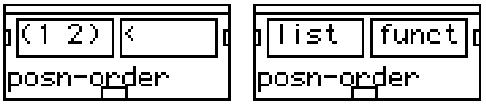


will return
? PW->((3 9 3) (13 13 4) (11 13 6) (16 16 1)), and



will return
? PW->((16 16 1) (3 9 3) (13 13 4) (11 13 6))

posn-order



Syntax

(epw::posn-order list funct)

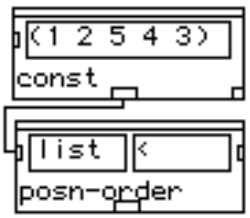
Input

list	A list of numbers
funct	A predicate (<, >, =)

Output

returns	A list. See description

The **posn-order** module returns a list of rows of the list *list,* ordered according to the function *funct.* It is possible to change *funct* and obtain the rows according to other ordering principles. For example



returns
? PW->(0 1 4 3 2)

permut-circ

Syntax

(epw::permut-circ *list &optional nth*)

Input

list	A list.

Optional Input

nth	An integer (optional argument)

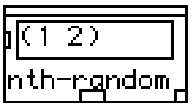
Output

returns	A copy of *list* with its elements shifted

The **permut-circ** module returns a circular permutation ;of a *list* starting from its *nth* element, (nth is the argument of the second optional input) (which defaults to 1) , (*nth* = 0 means the first element of *list, nth* = 1 means the second element of *list,* and so on).

For example, if *list* is (1 2 3 4 5 6 7 8 9 10), the **permut-circ** module returns (2 3 4 5 6 7 8 9 10 1), (the default is one). On the other hand (if the second input is open, *nth*), if *list* is (1 2 3 4 5 6 7 8 9 10), and *nth* is 3 (zero) , **permut-circ** returns (4 5 6 7 8 9 10 1 2 3).

nth-random



Syntax

(epw::nth-random *list*)

Input

list	A list.

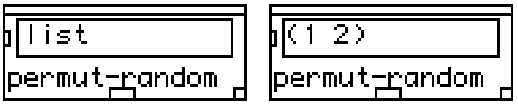
Output

returns	An element, randomly chosen, from *list*

The **nth-random** module returns a random element from its input *list.*

For example, the list (1 2 3 foo bar) might return the value 3 at the first evaluation; the next time a value was requested it might return the string "foo; " and the next time it perhaps returns 2, and so on.

permut-random



Syntax

(epw::permut-random *list*)

Input

list	A list.

Output

returns	A list, with the ordering of its elements randomly rearranged

Returns a random permutation of *list.*

Abstract Modules

absin

Input

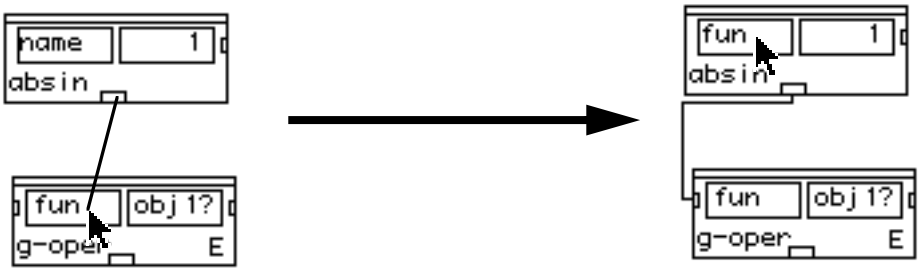
name	A name
order	An integer or float

Output

returns	nil

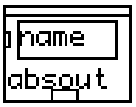
An abstraction (or a subpatch) must have one and only one **absout** box, but it can have as many **absin** boxes as one might want (including none). **Absin**s specify the input boxes of an abstraction-box. They are sorted by the value given by the second input box. All **absin** boxes, in an abstraction must have different names and different numbers.

Please remember that when patching an **absin** module to another module, the *name* input of **absin** takes the same name as the variable linked to the patched window.



It is advised to give a name to the absin module *after* having made the patch connection.

absout



Input

name	A name (and a patch)

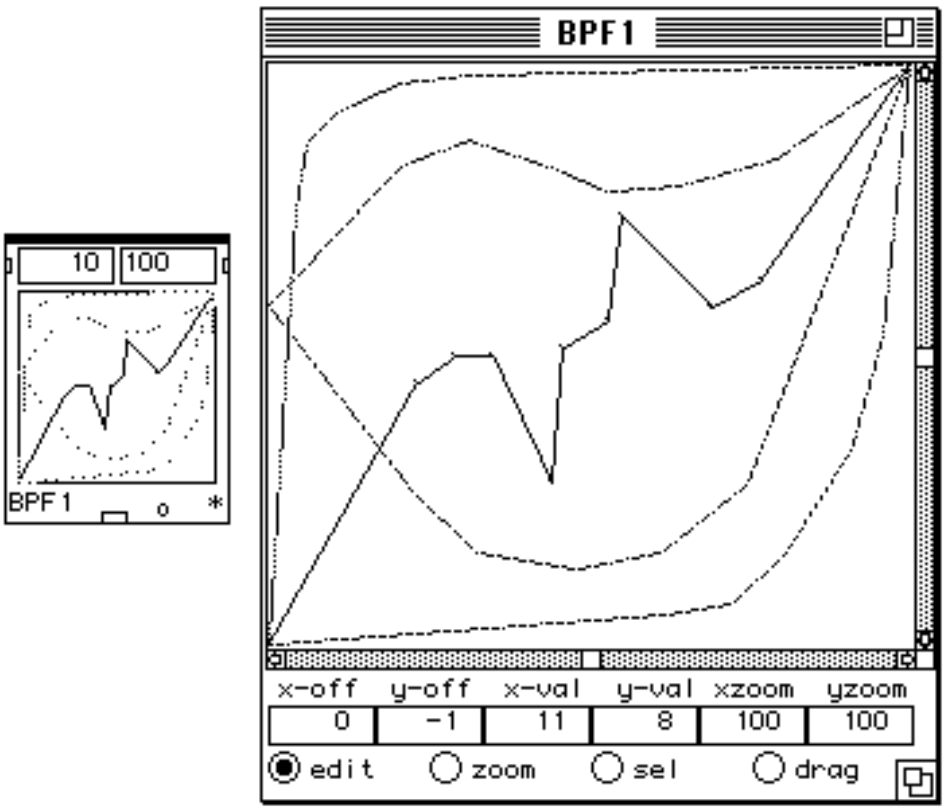
Output

returns	This module is evaluated when the abstraction output is evaluated

An abstraction (or a subpatch) must have one and only one absout box, but it can have as many **absin** boxes as you desire (including none). **absout** specifies the output of an abstraction-box. You can edit the name of the abstraction-box and the abstraction window by editing the input-box of the **absout** box.

Breakpoint function (BPF) Modules

multi-bpf



Input

tlist	A list (or a list of lists) of time values (in ticks).
vl/bpfs	A list (or a list of lists) of values .

Output

returns	A pointer to the module, or a list of break-point objects or a list of 'y' or a list of *x*

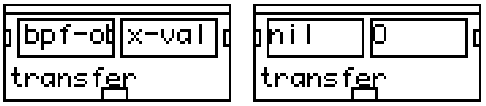
The **multi-bpf** module can be used to create and edit simultaneous breakpoint functions an once, create and edit coordinate pairs (x,y), display a series of coordinate pairs, either as a BPF or as a series of points, save and load **multi-bpf** modules to and from a library and retrieve data concerning the points contained in the **multi-bpf** module. A function editor can be opened by selecting this box and typing *o* from the keyboard. The BPF can

be changed from PatchWork by connecting a value-list to the vl/bpfs (or a list of lists-values) input box and option-clicking its output box. If there is no connection in the first input box *tlist* then the points have a constant time-difference.If the first input box *tlist* is connected, then the input should be a list of ascending timepoints. One can change the representation of the module **BPF** (in segments by default) to a representation in points by selecting *flip-mode* in the front menu. Click on the *A* of the module to open the front menu. For more information, type *h* with the window of the module open.

BPF Editor Keyboard Commands

H	opens the Window Help file, displaying commands
Return	selects the current PW window, hiding the BPF editor
Enter	selects the current PW window, hiding the BPF editor
R	renames the BPF window
Backspace	deletes the selected point
f	rescales the BPF so that the function fills the editor window
K	removes all point except the first
+	zoom out
-	zoom in
g	show/hide the grid
->	time-stretch the selected points
<-	time-contract the selected points
up-arrow	stretch the values of the selected points
down-arrow	contract the values of the selected points
tab	change to another edit mode following the sequence (edit - zoom - sel - drag ...)
a	add another BPF to the editor. Note that the addition of a new BPF is always done by duplicating the current (i.e. selected) BPF
s	select one BPF
d	deletes the selected BPF

transfer



Syntax

(pw::transfer *bpf-ob x-val*)

Input

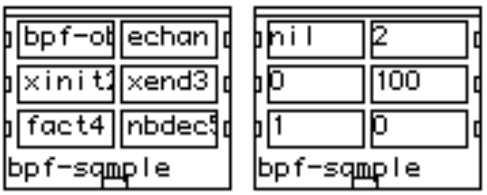
bpf-ob	The output of a **multiple-bpf** module in mode "bpf-object"
x-val	A number or a list of numbers

Output

returns	The *y* value (or list of y values) which corresponds to *x-val* for the connected **multiple-bpf**

The input **multiple-bpf** should always be connected with a **multiple-bpf** box (in mode 'bpf-object'). Behaves like a transfer function when fed with values to the second input box *x-val*. returns the y value (or list of y values) which corresponds to *x-val* (or list of x values) for the connected **multiple-bpf.**

bpf-sample



Syntax

(pw::bpf-sample bpf-ob echant xinit xend fact nbdec)

Input

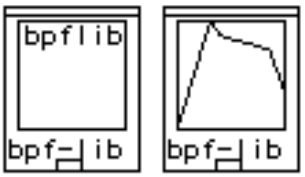
bpf-ob	The output of a **multiple-bpf** module in mode "bpf-object"
echant	A number
xinit	An integer
xend	An integer
fact	An integer or a float
nbdec	An integer

Output

returns	A sampled list in *bpf0*

The **bpf-sample** module creates a list starting by sampling a breakpoint function table. *bpf-ob* is the input to the table, *echant* is the number of samples desired, *xinit* et *xend* delimit the sampling interval. The *fact* variable is a multiplicative coefficient for scaling the data, and *nbdec* is the number of decimals desired in the output.

bpf-lib



Output

returns A breakpoint function

Breakpoint functions can be stored in a library. There is only one current library, structured as a circular list. The menu item **add to lib** (in the BPF menu, when the window of **multiple-bpf** module is open) adds the current BPF to the library and **reset lib** resets the library to one item: a ramp. The menu items **next BPF from lib_** and **prev BPF from lib_** allow browsing in the library.

Extern Modules

in

Input

foo	A name

Output

returns	The evaluation of the patch that is connected to the **out** module with the same name.

The **in** module receives remote messages from **out** modules that share the same name. See an example in the reference of **out** module.

out

Input

foo	A name (and a patch)

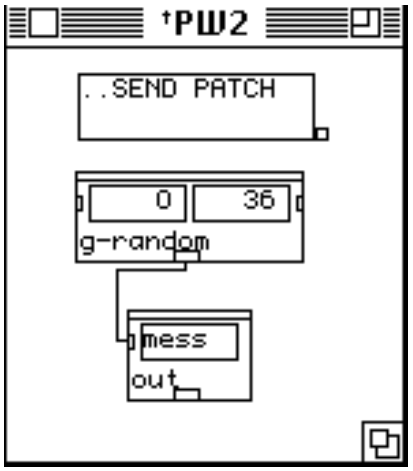
Output

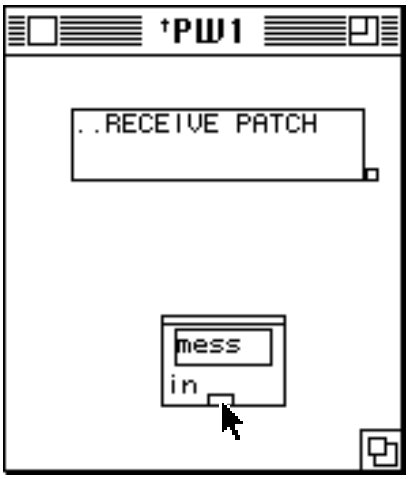
returns	The evaluation of the patch to which it is connected

The **out** module allows one to pass the evaluation of a patch to a remote destination (i.e., to another patch or to another window). **out** modules are assigned a name. **in** modules with the same name receive the results of the patch evaluation.

For example:

It is possible to have to windows that communicate with each other. In the first window (PW2), a patch can generate data that will be transmitted through the **out** module and received in a second window (PW1) by module **in**. Notice that the two modules (**in** and **out**) have identical names.

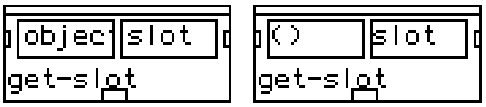




Evaluation of the in module in window PW1 will return a random number (uniform distribution) between 0 and 36.

Warning : it is advised to always load new modules to assign pairs of variables; also, avoid duplicating or changing the names of the already used **in** and **out** modules.

get-slot



Syntax

(pw::get-slot *object slots*)

Input

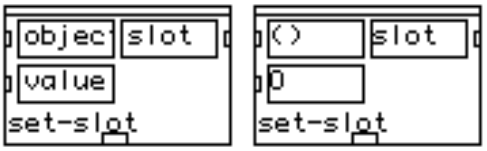
object	A chord object, a breakpoint function, a collector or a group thereof.
slots	A slot (data field) for the object(s) concerned

Output

returns	The current value(s) for the requested slot(s) of object

Inspects an object slot. The first input must be an object-instance (such as chord objects or breakpoint functions), or a list of object-instances. The second input is a slot-name. Returns the corresponding value(s) of the chosen slot. Evaluating the **get-slot** module with the name 'slot in the *slot* input, the module returns the list of the valid name slots of the object in the first input *object.*

set-slot



Syntax

(set-slot object slots value)

Input

object	A chord object, a break-point function, a collector or a group thereof.
slots	A slot (data field) for the object(s) concerned
value	A value or list of values

Output

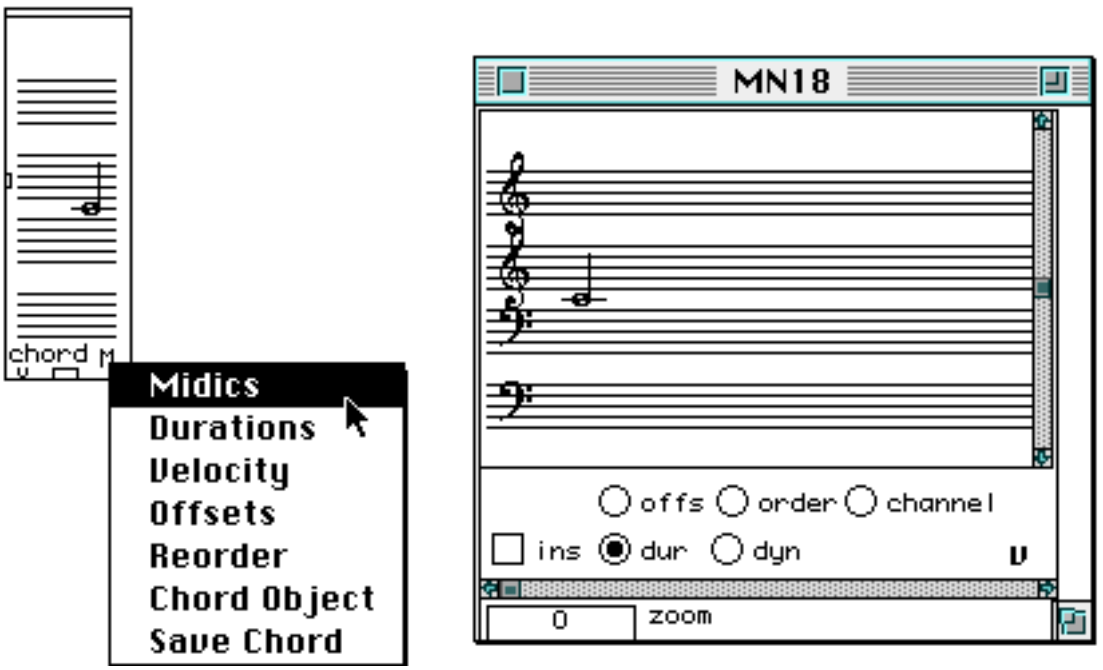
returns	The object(s) to which the module is connected

An object slot modification. The first input must be an object-instance or list of object-instances. The second input is either a slot-name or a list of slot names. The *value* input is the corresponding new value or list of new values. Returns the modified object(s).

Warning: This operation is potentially dangerous. You should know what you are doing when changing object slot's values.

Edit modules

chord



Input

midics	A list of midicents or a chord object

Output

returns	A list or a chord object

The module chord is a constructor module for chords. Its input can be a list of midicents or a chord object. In the first case, a chord object having notes with the given pitches is created (or modified, if a chord for the module exists already). In the second case, the given chord object is copied into the module. A chord module has a popUp menu linked to the letter just to the right of its output box. The output of the **chord** module depends on the menu item chosen. The items in this menu are as follows:

midics	Output is the list of *midic* slots of the notes.
Durations	Output is the list of *dur* slots of the notes.
Velocity	Output is the list of *vel* slots of the notes.
offsets	Output is the list of *offset-time* slots.

Reorder	Output is the list of *midic* slots, reordered according to the values for field *order*
Chord Object	Output is the whole chord object.
Save Chord	Output does not change. The module is saved in a file.

The letter to the right of the chord module's output box indicates the current output option.

An editor for the chord object is entered either by selecting the module and typing the letter *o*, or by double-clicking on the module's name. Type *h* with the music notation editor opened for more information. See the Introduction manual for more information.

Chord Editor Window Keyboard Commands

The chord editor window understands certain keyboard commands :

H	Opens the Window Help file with list of commands
Return	Selects the current PW window, hiding the **chord** editor
Enter	Selects the current PW window, hiding the **chord** editor
R	Rename the editor window for the **chord** module
p	Plays the chord
t	Time view (If the chord editor window is not open *t* means: Display tutorial help patch.)
n	Normal view
e	Typing *e* (edit) after selecting notes opens a small dialog box at the bottom of the MN window, where you can enter a value; typing Return puts that value in the corresponding field (midic, dur; chan, time-offset, dyn) of all selected notes. The field chosen is that of the currently active button (offs, channel, dur, dyn) or midic, if no other is active.
up-arrow	Transpose upwards the selected note (up-arrow = 1/4 tone, shift + up-arrow = 1/2 tone, control + up-arrow = 1 octave)
down-arrow	Transpose downwards the selected note (down-arrow = 1/4 tone, shift + down-arrow = 1/2 tone, control + down-arrow = 1 octave)
Backspace	Delete the selected note or notes
Tab	Select edit mode.

mk-note

Syntax

(pw::mk-note midic dur vel chan &optional m-ins)

Input

midic	A midicent.
dur	A positive integer.
vel	Integer between 0 and 127.
chan	Integer between 1 and 16

Optional input

m-ins	An object representing an instrument.

Output

returns	A note object

mk-note (make note) is the note object constructor module.
Each entry corresponds to the value of the named slot of the object. The *midic* argument, for instance, is a midicent value (with a default value of 6000) that is stored in the midic slot of the constructed object, *dur* is the duration of the note, in hundredths of a second, *vel* is the velocity or "dynamic" of the note, between 1 and 127 (default is 100); *chan* is a MIDI channel number (default is 1). The optional *m-ins* input, if given, should be connected to a patch that outputs a PatchWork instrument. This could be, for example, a structured abstraction, that is, a window with a subpatch on it. Any argument not supplied to **mk-note** takes its default value.

mk-chord

Syntax

(pw::mk-chord midics durs offs dyns &optional channs ords comment object)

Input

midics	Midicent(s)
durs	Positive integer(s).
offs	Integer(s).
dyns	Integer(s) between 0 and 127

Optional input

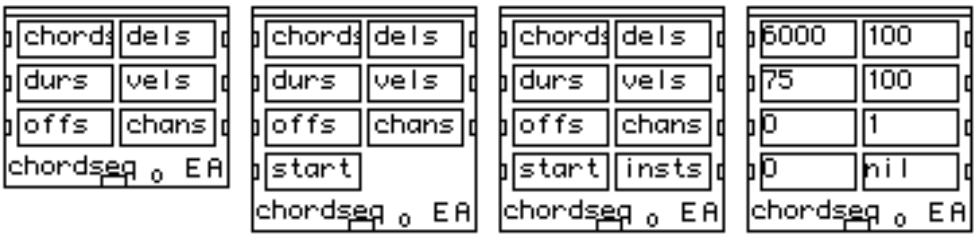
channs	Integer(s) between 1 and 16.
ords	Integer(s).
comment	String(s).
object	Note object(s) or a chord object.

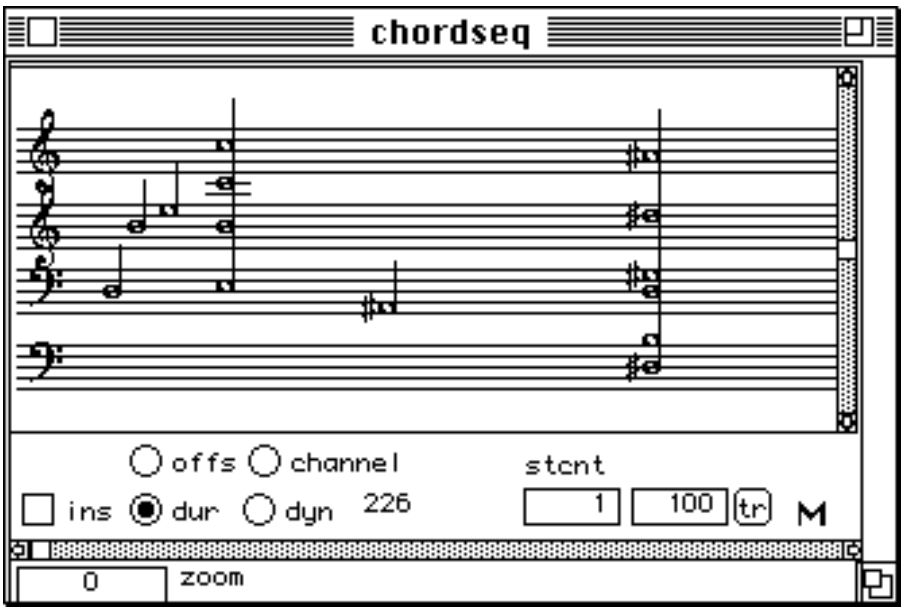
Output

returns	A chord object

The **mk-chord** module is a chord constructor object. Each input is either an element or a list of elements. The number of notes in the constructed chord is equal to the length of the longest list of values supplied for the inputs. Each note of the chord is constructed by taking the next element of each list supplied to the inputs and operating on them exactly as for the **make-note** object described above. If a list is exhausted, the default value of the note object slot it names is taken for each successive note until all lists are exhausted. If an input is supplied to the *object* argument it must be either a note object, a list of note objects or a chord object. In this case the behavior of the module is as described before, except that the values of the slots of the supplied objects (either note or chord) have precedence over the default values. That is, if one of the lists in the inputs is exhausted but note objects remain in the list of note objects supplied in *object*, then those remaining note objects will not be modified in the slot corresponding to the input that was exhausted. Similarly, if a chord object is supplied in *object* , then those note objects in the chord's note list (that is, the list linked to the chord's *notes* slot) which have not yet been processed will not be modified in the slot corresponding to an exhausted list.

chordseq





Input

chords — List, list of lists, chord object, list of chord objects, midics, sequence of chords. Can receive a list of midicents. If the list is simple, it is interpreted as a melodic sequence. If the list has two levels of structure, each sub-list is interpreted as a chord.

dels — A positive integer. A list of temporal intervals, in hundredths of a second, which determine the gap between the attack of one chord and that of the next.

durs — A positive integer. A list of durations, in hundredths of a second (or list of lists)

vels — Integer between 0 and 127. A list of MIDI velocities (or list of lists)

offs — Integer (positive or negative). A list of offsets, in hundredths of a second (or list of lists)

chans — Integer between 1 and 16. A list of MIDI channels (or list of lists)

Optional input

starts — An integer or a float. A number that determines at what moment the sequence starts.

insts — Positive integer (used for PW-Chant synthesis). Input for synthesis instruments.

Output

returns — A chord sequence object

The chordseq module constructs sequences, i.e. *chord sequence objects.*. It builds a chord-line (an object with a 'chords' slot containing a list of chords). The input *chords* can be either a list (or list of lists) of midics or note objects, a chord object (or list of chord objects), or a single midic or note object.

The *dels* input controls the spacing of the chords (in 100ths of a second). The *durs* input sets the duration of the chord (in 100ths of a second), the *vels* input controls the dynamic of the chord, *chans* determines the MIDI channel number of the chord, *offs* input controls the offset time of each note of the chord, *start* is the start point of the sequence (in 100ths of a second), and the *insts* input lets one connect an instrument to chord. A popup menu is linked to the letter *A* just to the right of the output box. This menu can save the module (and its collected chord sequence) into a file.

An editor for the **chordseq** object is entered either by selecting the module and typing the letter *o*, or by double-clicking on the module's name. Click *h* with the music-notation-editor opened for more information. The module can be locked (to avoid new evaluations of the patch that is under 'chord') to keep its contents by clicking on the small *o* in the lower right of the module. The *o* indicates that the module is open. See the Introduction manual for more information.

Chordseq Keyboard Commands

The chordseq editor window understands certain keyboard commands :

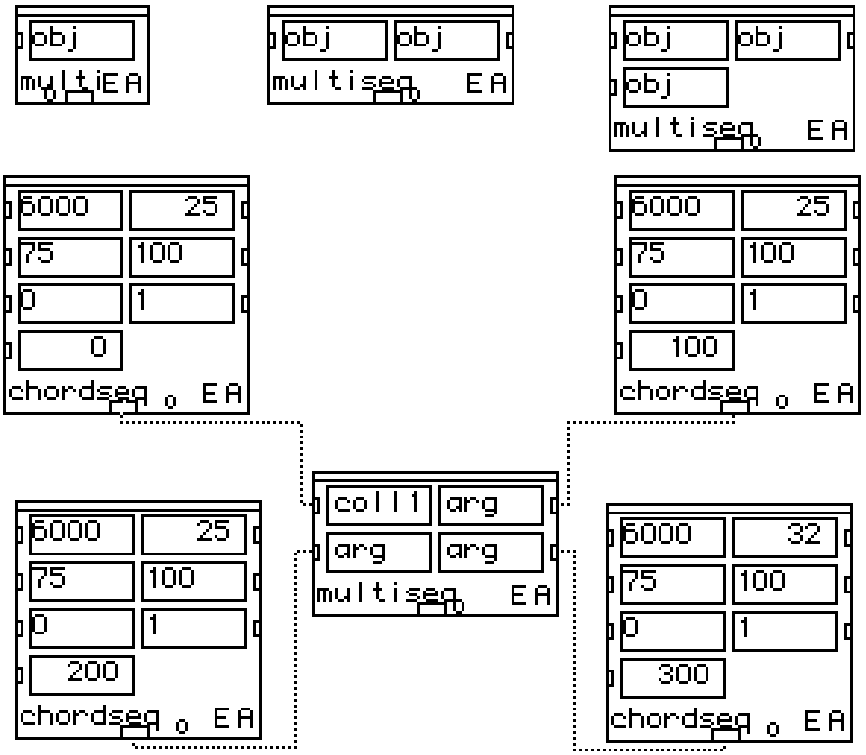
H	Opens the Window Help menu with list of commands
Return	Selects the current PW window, hiding the **chordseq** editor
Enter	Selects the current PW window, hiding the **chordseq** editor
R	Renames the editor window for the **chordseq** module
p	Plays the entire sequence or the selected chords
o	Opens the selected chords

A lock protects the contents of a **chord** module (whether edited or calculated) or to avoid reevaluation.

The **chordseq** module can be extended.

A pulldown menu saves the module along with its current contents. See also **Save with MN** menu option.

multiseq



Input

obj A chord sequence object.

Optional input

obj A chord sequence object.

Output

returns A list of chord sequence objects.

A **multiseq** (multiple sequence) box represents polyphonic data. It takes one or more chord sequence objects as input and returns them in a list. This module works with the associated music notation editor, which displays as many systems as chord sequence inputs have been defined for the module. A pulldown menu is linked to the letter *A* just to the right of the output box. It offers the option of saving the module with all its chord sequences into a file. The **multiseq** module is state-preserving. It only changes its output if there is a module connected to one of its inputs (or if changes are made by hand in the editor, of course).

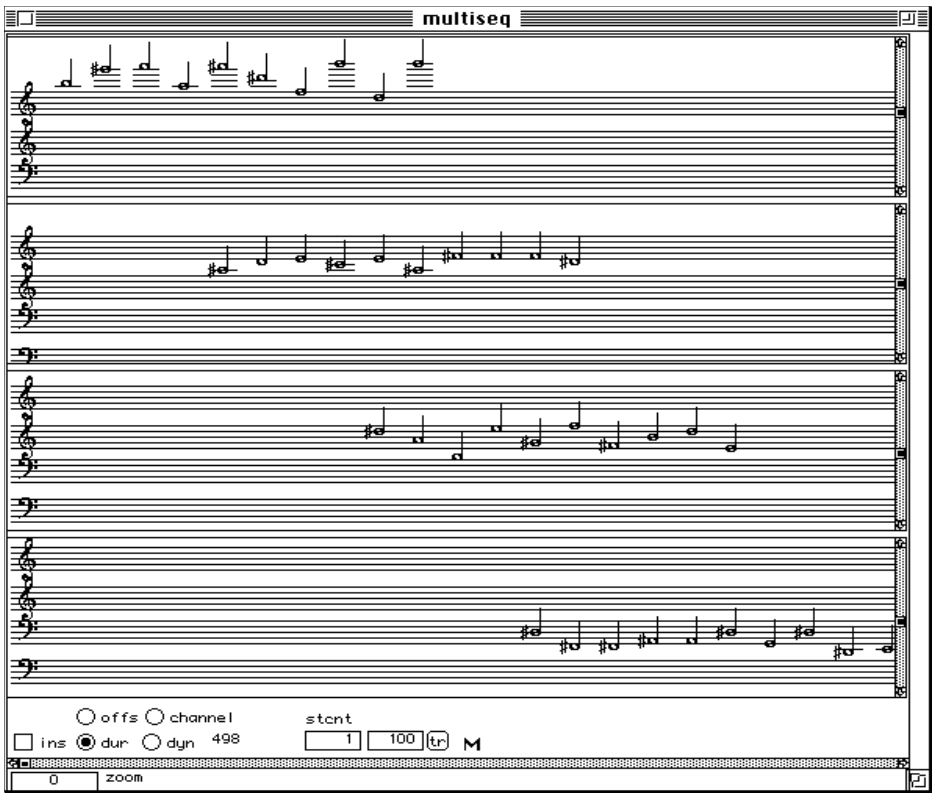
Keyboard Commands for multiseq Editor

The **multiseq** editor window understands certain keyboard commands :

H	Opens the Window Help menu with list of commands
Return	Selects the current PW window, hiding the **multiseq** editor
Enter	Selects the current PW window, hiding the **multiseq** editor

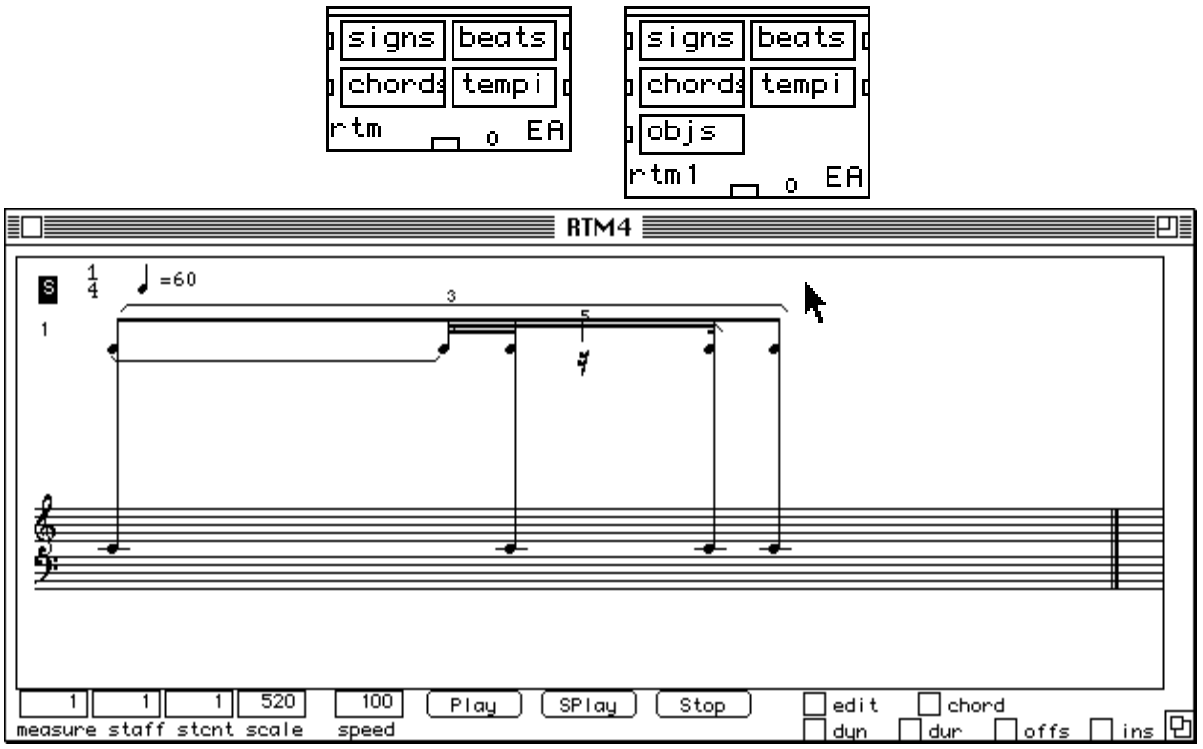
R	Renames the editor window for the **multiseq** module
p	Plays the entire sequence or the selected chords
o	Opens the selected chords

This box can be opened by selecting it and pressing *o* from the keyboard or extended by option-clicking bottom-right.



Type *h* with the music notation editor opened for more information. See the Introduction manual for more information.

rtm



Input

signs	A list of measures. For example: ((3 4) (2 8) (4 16)).
beats	A list of the subdivisions of the basic pulse. For example: (1/4 1/3 2/5). (For more explanation, see the section on RTM inputs.)
chords	A list of pitches, note-objects, chord-objects, measure objects, notes or chords given in midicents and lists of whole number values given in the form of objects. Sublists are considered as chords.
tempi	A list or an atom. A list of tempos. For example: (60 78 120 90).

Optional Input

objs	A list of measure objects. This entry connects a **quantify** module to an **rtm** module

Output

returns	A PatchWork measure-line object.

The **rtm** module is a rhythm editor that makes a measure object out of the input rhythm. The input *signs* is a list of time signatures, *beats* is a list of beat divisions in expand list notation. For example '(3*(1/4) /(1 2 1) 2//4 2*(1/5 1/7 1/8)) takes:

three times: four 16th notes,

one time: an 8th, a 16th and an 8th note,

one time four 8th notes and

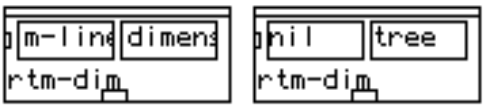
two times: five 16th notes (in a beat) seven 16th notes (in a beat)
and eight 32ths notes (in a beat).

The duration of the notes in the chord are automatically taken to be the full value of their rhythmic notation, so a quarter-note at a tempo of 60 lasts one full second (staccato and legato articulations are not taken into account.) The duration of notes can be changed by hand in the editor, but not from outside the module. The *chords* input is a list (or list of lists) of midics, and *tempi* is a list of tempos in expand list notation (see **expand-list**). The module can be locked (to avoid new evaluations of the patch that is under 'chord') to keep its contents by clicking on the small *o* in the lower right of the module. The *o* indicates that the module is open. An editor for the **rtm** object is entered either by selecting the module and typing the letter *o*, or by double-clicking on the module's name. Click *h* with the rhythm editor opened for more information. See the Introduction manual for more information.

RTM Editor Keyboard Commands

H	With rtm module selected, opens the Window Help with list of editing commands
Return	Selects the current PW window and quits the RTM editor
Enter	Selects the current PW window and quits the RTM editor
R	Renames the RTM editor window
Backspace	Delete the measure(s) or beat(s) selected
H	(Home) With the editor open, goes back to the first measure
K	Delete all the measures in all lines
L	Advances the window to the last measure
+	Advances the window to the next measure
-	Returns the window to the previous measure
->	Advances the window to the next page
<-	Returns the window to the previous page
p	Plays all staves within the selected measure
P	Play only the selected measure
e	Toggle in and out of the edit mode
k	Empty the MIDI buffer
S	Select all the edit buttons
U	Unselect all selected edit buttons
D	Redraw the window

rtm-dim



Syntax

(pw::rtm-dim m-lines dimension)

Input

m-lines	A PatchWork measure line object
dimension	A slot name (tree, delay, durs, sign, tempo, or chords)

Output

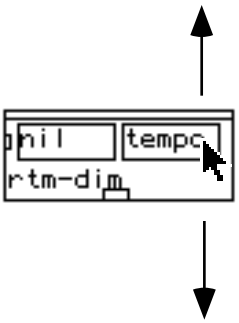
returns	A list values of type *dimension*

Takes the output of an rtm module and filters a specified parameter from this output. The *m-lines* input can be a measure-line object or a list of them. The dimension (or parameter) can be: tree, delay, sign, tempo, durs or chords.

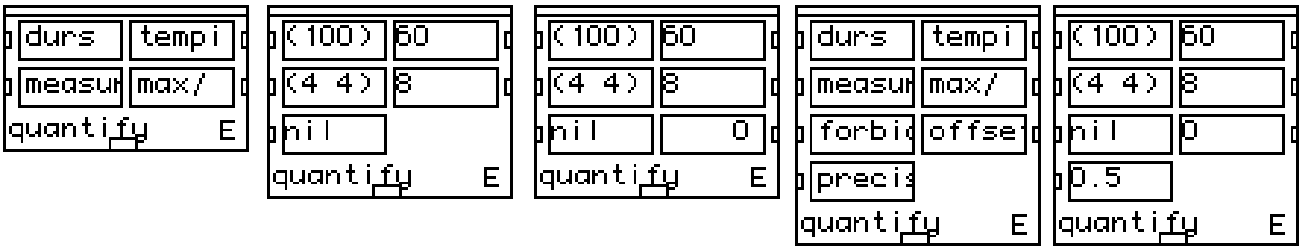
The corresponding output is:

tree	The list of rhythm beat trees for each measure-line.
delay	The list of delays for each measure-line
dur	The list of beat durations for each measure-line
sign	The list of measure signatures for each measure-line
tempo	The list of measure tempi for each measure-line.

If you click and hold down on the parameter field of the **rtm-dim** module, it turns into a slider. As you move the mouse up and down, the parameter names change.



quantify



Syntax

(pw::quantify durs tempi measures max/ &optional forbid offset autom)

Input

durs	A list
tempi	A fixnum , a float or a list
measures	A list
max/	A fixnum or a float

Optional input

forbid	A list or a list of lists
offset	A fixnum or a float
precision	Float between 0 and 1.

Output

returns	A list of PatchWork measure-line objects

Quantizes a list of *durs* (100 = 1 sec.) into the given measure(s), with the given *tempi. max/* is the maximum unit division minus 1 that is taken to be a significant duration. For instance, if *max/* = 8, the maximum unit division will be a septuplet. A list of *forbid* forbidden unit divisions can optionally be specified.

With this variable, you can control the subdivisions of the beats, either by avoiding them or by imposing them, at a global level or at the *beat* level.

A simple list, such as (11 9 7 6), does not permit at a global level divisions by 11, 9, 7, or 6. The introduction of sub-lists at the first level indicates a control over the measures. For example, ((5 6) (7 4) () () (4 7)) indicates that in the first measure, subdivisions by 5 and by 6 are forbidden, and in the second and fifth measure, subdivisions by 7 and by 4 are forbidden. As the third and fourth sub-lists are empty lists, there are no restrictions for these measures. A second level of sub-lists will permit to control subdivisions of beats. The list (((5 4) () (3 6) ()) (() () (8 7) ()) (3 2) ()) indicates :

first measure

first beat - fourth beat : no restriction

second beat : no restriction

third beat : subdivisions by 3 and by 6 forbidden

fourth beat : no restriction

second measure

first beat - fourth beat : no restrictions

second beat : no restrictions

third beat : subdivisions by 8 and by 7 forbidden

fourth beat : no restrictions

third measure

all beats : subdivisions by 3 and by 2 forbidden

fourth measure

all beats : no restrictions

To impose subdivisions, you add a *!* at the beginning of the lists.

At a global level

(! 5) imposes a subdivision by five on the entire sequence

(! 5 7 6) imposes a subdivision by 5, by 7, or by 6 on the entire sequence.

The module will do the necessary computations et will choose one of the subdivisions in such a way that approximation errors are reduced.

The syntax is the same for all other levels:

For measures

((! 3 4) (! 5) () () ())

and for time units

(((! 5 4) () (! 3 6) ()) (() () (! 8 7) ()) (! 3 2) ()) .

Of course, it is possible to mix syntaxes at the measure level as well as at the beat level. Here is an example:

(((5 4) () (! 3 6) ()) ((! 6) () (8 7) ()) (! 3 2) (6 8)),

In this example, some measures and time units have impositions of subdivisions, where in others, we have restrictions of subdivisions.

The output is a list of measure-objects that can be entered directly into the optional input objs of the rtm module.

poly-rtm

Syntax

(pw::poly-rtm rtmcol &rest rtmcn)

Input

rtmcol	A list of PatchWork measure-line objects

Optional Input

arg	A PatchWork measure-line object

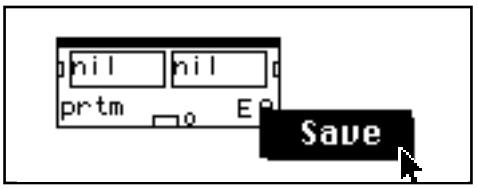
Output

returns	List of PatchWork measure-line objects

The inputs of the **poly-rtm** box should be connected only with **RTM** modules. If the out-box of a **poly-rtm** box is double-clicked, then all the measure-line objects are taken inside the **poly-rtm** box. The module can be locked (to avoid new evaluations of the patch that is under 'chord') to keep its contents by clicking on the small *o* in the lower right of the module. An *o* indicates that the module is open.

You can enter an editor for the **poly-rtm** object either by selecting the module and typing the letter *o*, or by double-clicking on the module's name. Click *h* with the **poly-rtm** editor opened for more information.

As of PatchWork version 2.5, the module **poly-rtm** can save musical data.



Keyboard Commands for poly-rtm Editor

H	With **poly-rtm** module selected, opens the Window Help with list of editing commands
Return	Selects the current PW window and quits the editor
Enter	Selects the current PW window and quits the editor
R	Renames the editor window
H	(Home) With the editor open, goes back to the first measure
L	Advances the window to the last measure
+	Advances the window to the next measure
-	Returns the window to the previous measure

- >	Advances the window to the next page
<-	Returns the window to the previous page
p	Plays all staves within the selected measure
P	Play only the selected measure
e	Toggle in and out of the edit mode
Backspace	Delete the measure(s) or beat(s) selected
K	Delete all the measures in all lines
k	Empty the MIDI buffer
S	Select all the edit buttons
U	Unselect all selected edit buttons
D	Redraw the window

f->mc

Syntax

(epw::f->mc freqs &optional approx ref-m)

Input

freqs	An integer, float, or list

Optional input

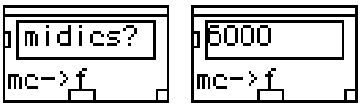
approx	An integer, or float
ref-m	A midicent

Output

returns	Midicent value(s)

Converts *freqs* to midicents. It takes a frequency (Hz) or list of frequencies and returns corresponding midicent values. The optional *approx* argument lets one limit returned values to a given approximation (see **approx-m**). When *approx* = 1, the optional argument *ref-m* in midicents specifies the frequency resolution of the approximation. A value of 100 specifies semitone resolution, 50 specifies quartertone resolution, and so on.

mc->f



Syntax

(epw::mc->f midics)

Input

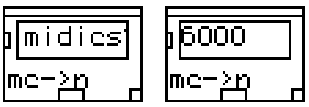
midics?	An integer or list

Output

returns	Frequency(ies)

Converts a midicent pitches *midicents* to frequencies (Hz).

mc->n



Syntax

(epw::mc->n midics)

Input

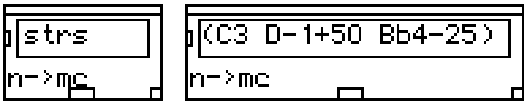
midics	An integer or list.

Output

returns	Symbolic (Ascii) note names

The **mc->n** module takes a midicent value *midics* or a list, and returns corresponding ASCII note names. Symbolic note names follow standard notation with middle C (midicent 6000) being C3. Semitones are labeled with a '#' or a 'b.' Quartertone flats are labeled with a '_', and quartertone sharps ;with a '+'. Thus, C3 a quartertone sharp (midicent 6050), would be labeled 'C+3'. Gradations smaller than a quartertone are expressed as the closest quartertone + or - the remaining cent value (i.e., midicent 8176 would be expressed as Bb4-24).

n->mc



Syntax

(epw::n->mc strs)

Input

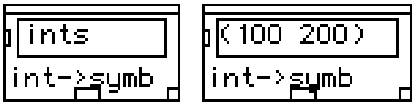
strs	An Ascii character string (see description) or list thereof.

Output

returns	Midicent(s)

n->mc takes a symbolic *strs* string or list, and returns corresponding midicent values. Symbolic note names follow standard notation with middle C (midicent 6000) being C3. Semitones are labeled with a '#' for sharp or 'b' for flat. Quartertone flats are labeled with a '_', and quartertone sharps with a '+'. Thus, C3 a quartertone sharp (midicent 6050), would be labeled 'C+3'. Gradations smaller than a quartertone are expressed as the closest quartertone + or - the remaining cent value (i.e., midicent 8176 would be expressed as Bb4-24).

int->symb



Syntax

(epw::int->symb ints)

Input

ints	An integer or list

Output

returns	Symbolic interval name(s). (See below.)

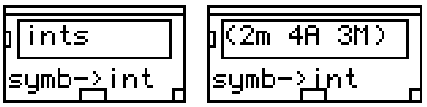
int->symb takes an interval expressed in midicents, and returns a symbolic interval name. Intervals are labeled as follows:

1 = unison	2m = minor second
2M = major second	3m = minor third
3M = major third	4 = perfect fourth
4A = tritone	5 = perfect fifth
6m = minor sixth	6M = major sixth
7m = minor seventh	7M = major seventh

All intervals larger than an octave are expressed by adding or subtracting an octave displacement after the simple interval name; for example, a major tenth becomes 3M+1, etc.

Note: for the time being, the program has a strange way of expressing downward intervals: it labels the interval as its inversion, and then transposes downwards as necessary. Thus, a major third down (-400 in midicents), returns 6m-1.

symb->int



Syntax

(epw::symb->int ints)

Input

ints	An Ascii string or list thereof

Output

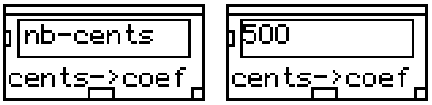
returns	Midicent(s)

The **symb->int** module takes a symbolic interval name*ints* , and returns an interval expressed in midicents. Intervals are labeled as follows:

1 = unison	2m = minor second
2M = Major second	3m = minor third
3M = Major third	4 = perfect fourth
4A = tritone	5 = perfect fifth
6m = minor sixth	6M = Major sixth
7m = minor seventh	7M = Major seventh

All intervals larger than an octave are expressed by adding or subtracting an octave displacement after the simple interval name; for example, a major tenth becomes 3M+1, etc. Note: for the time being, PatchWork has a strange way of expressing downward intervals: it labels the interval as its inversion, and then transposes downwards as necessary. Thus, a Major third down 6m-1, returns -400 in midicents .

cents->coef



Syntax

(epw::cents->coef nb-cents)

Input

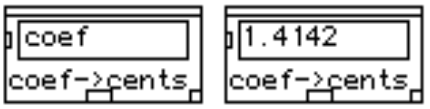
nb-cents	An integer

Output

returns	A float (see description)

cents->coef takes an interval *nb-cents* expressed in midicents and returns the ratio between two frequencies separated by that interval; that is, the value: (freq + *nb-cents*) / freq.

coef->cents



Syntax

(epw::coef->cents coef)

Input

coef	A float.

Output

returns	A midicent

coef->cents takes a *coef f1/f2* and returns the interval, expressed in midicents, between *f1* and *f2.*

approx-m

Syntax

(epw::approx-m midics? approx &optional ref-midic)

Input

midics?	An integer or list
approx	An integer or float

Optional inputs

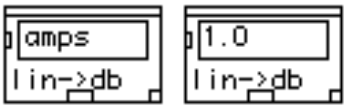
ref-midic	An integer

Output

returns	midicent(s)

approx-m takes a midicent value *midicents* and returns an approximation to the nearest division of the octave as defined by the user, *approx.* The value of resolution determines the resolution of approximation. An argument of 1, results in an output where all values are rounded to the nearest whole tone; 2, to the nearest semitone; 4, to the nearest quartertone; 4.5, to the nearest 4.5th of a tone, etc. When *approx* = 1, the optional argument *ref-midic* in midicents specifies the frequency resolution of the approximation. A value of 100 specifies semitone resolution, 50 specifies quartertone resolution, and so on.

lin->db



Syntax

(epw::lin->db amps)

Input

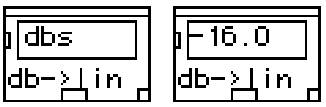
amps	An integer float or list

Output

returns	Number(s)

lin->db takes a linear amplitude number *amps* and returns the corresponding value in decibels. The input can also be a list of numbers; in this case a list of dB values is returned.

db->lin



Syntax

(epw::db->lin dbs)

Input

dbs	An integer float or list

Output

returns	Number(s)

db->lin takes a value in decibels *dbs* and returns the corresponding value in linear amplitude. The input can be a list of numbers. In this case a list of values is returned.

play/chords

Syntax

(c-pw-send-midi-note::play/chords midics vels chan durs &optional offs at-time

Input

midics	A list of midics.
vels	An integer between 0 and 127
chan	A MIDI channel.
durs	A nonnegative integer

Optional Input

offs	An integer or a list of integers
at-time	An integer or a list of integers

Output

returns	nil (and play via MIDI its contents)

The **play/chords** module formats and plays MIDI note events. If *midics* is a list, then the result is a chord. Notes are played with a channel *chan* velocity *vels* and duration *durs* as determined by the inputs. The optional input *offs* lets one assign a time offset (in 100ths of a second relative to time zero) for each note of the chord. If midics is a list of lists then **play/chords** produces a sequence of chords. The at-time determines the start time (in 100ths of a second relative to time zero) for each chord. If the second optional input *at-time* is a single value, chords are equally spaced in time by that value. A list for *at-time* gives each chord in the list its own start time. Note: if any of the argument lists is shorter than *midics,* the last value of those lists are used to play the remaining notes.

play/stop

Input

chord	A chord object
approx	Integer between 1 and 8
channel	A MIDI channel

Optional Input

dur	A non-negative integer

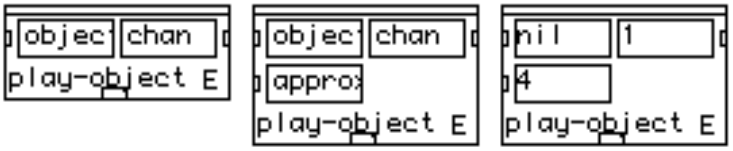
Output

returns	nil (and play via MIDI)

The play/stop module plays a *chord* object through the MIDI channel given in *channel.* The *approx* variable is the approximation value for midicents, which can be set to whole tone (*approx* = 1), semitone (*approx* = 2), quartertone (*approx* = 4, the default) or eighth-tone (approx = 8). If no duration is supplied (or if it is equal to zero) the module keeps playing until you option-click again at its output box. Otherwise it plays for the given duration *dur.* Notes with microtonal accidentals are sent to different Output

channels according to the following mapping: *channel* + 1 (eighth-tones), *channel* + 2 (quartertones) or *channel* + 3 (three-eighths tones). For example, if you set *channel* to 8, semitones are sent out channel 8, eighth-tones are sent out channel 9, and so on.

play-object



Syntax

(pw::play-object object chan &optional approx)

Input

object	A note, chord or chord sequence object
chan	A MIDI channel

Optional Input

approx	An integer between 1 and 8

Output

returns	nil

play-object plays a note, chord, or chord sequence specified in its input *object* through the MIDI channel specified in *chan*. The *approx* input is the approximation value for midicents. This can be the whole tone (approx = 1), semitone (approx = 2), quartertone (approx = 4, the default) or eighth-tone (approx = 8). The play operation takes the full duration specified in the given object. Notes with microtonal accidentals are sent to different output channels according to the following mapping: *chan* + 1 (eighth-tones), *chan* + 2 (quartertones) or *chan* + 3 (three-eighths tones). For example, if you set *chan* to 8, semitones are sent out channel 8, eighth-tones are sent out channel 9, quartertones are sent out channel 10, and so on.

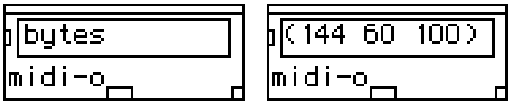
The module **play-object** has an input for MIDI channels.

The convention is :

-1 for MIDI channel 1.

- If you put 0, **play-object** will play using the MIDI channel setup of incoming objects, without changing it.

midi-o



Syntax

(pw::midi-o bytes)

Input

bytes	A list of bytes in MIDI format

Output

returns	Bytes of MIDI data

The **midi-o** module sends *bytes* out of the Macintosh serial modem port. For example, if we give the list (144 60 64) for bytes, our MIDI synthesizer (assuming it is connected) play middle-C on channel 1 with a velocity of 64. To turn off the note, we would have to give bytes the argument (144 60 0).

pgmout

Syntax

(pw::pgmout pgms chans)

Input

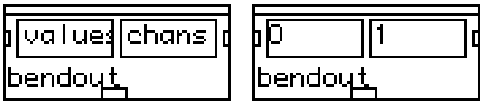
pgms	A MIDI program number or list of program numbers
chans	A MIDI channel number or list of channel numbers

Output

returns	Value of *chans*

The **pgmout** module sends a MIDI program change message out of the Macintosh serial ports. *pgms* is the program number and *chans* is the MIDI channel. Both of these can be lists. In this case a list of MIDI program change messages is sent out.

bendout



Syntax

(pw::bendout values chans)

Input

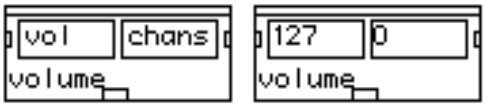
values	A number (or list thereof) between -8192 and 8190
chans	A MIDI channel (or list thereof)

Output

returns	Value of *chans*

bendout sends a MIDI pitchbend message(s) *values* in the given MIDI channel(s) *chans. values* and *chans* can be single numbers or lists. The range of pitch bend is between -8192 and 8190.

volume



Syntax

(pw::volume vol chans)

Input

vol	A number (or list thereof) between 0 and 127
chans	A MIDI channel (or list thereof)

Output

returns	Value of *chans*

volume sends a MIDI volume message(s) *values* in the given MIDI channel(s) *chans. vol* and *chans* can be single numbers or lists. The range of volume is between 0 and 127.

delay

Input

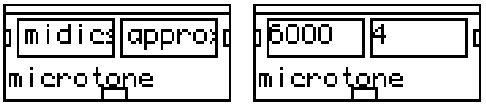
delay	An integer time value (in ticks, where 1 tick = 10 milliseconds).
patch	A patch.

Output

returns	The evaluation of *patch* after the delay

The **delay** module evaluates *patch* after the time period *delay.*

microtone



Syntax

(epw::microtone midics approx)

Input

midics	A midicent or list of midicents
approx	An integer between 1 and 8

Output

returns	A list

microtone returns a list of microinterval numbers given a list of midicent values *midics.* The microinterval numbers and their corresponding microinterval value, in *approx,* is given below:

1 = no microinterval.

2 = eighth tone

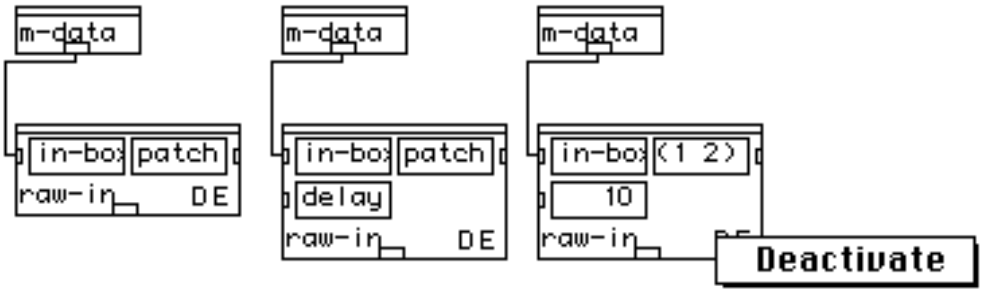
3 = quarter tone

4 = three-eighths tone

For example, the call (microtone '(6025 6200 6347 6750 6176) 8)

returns the list (2 1 3 3 4).

raw-in



Input of m-data

	No connection

Output of m-data

returns	Successive MIDI events

Input of raw-in

in-box	Always connected to the output of m-data
patch	A patch

Optional input of raw-in

delay	An integer

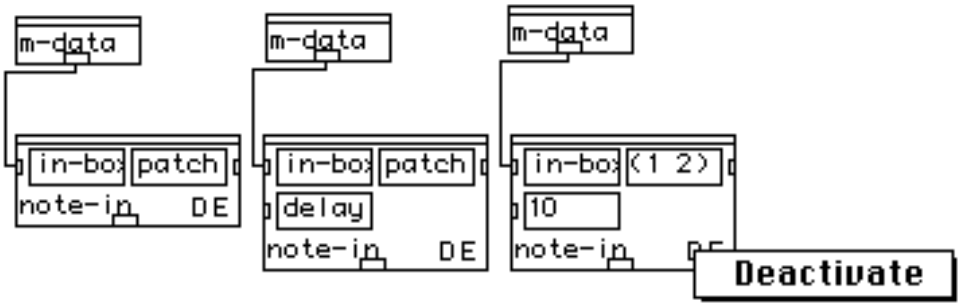
Output of raw-in

returns	nil

The modules **raw-in** and **m-data** together gather incoming MIDI data from the Macintosh serial ports. In order for them to work, there must always be a loop with **raw-in** at the bottom, **m-data** at the top, and some kind of *patch* dealing with the MIDI data in between. The optional input *delay* gives the delay time evaluation (in 100ths of a second) of the input *patch.* In order to start the loop collecting MIDI data, the **raw-in** module must be evaluated. When you are finished collecting MIDI data, select "Deactivate" in the **raw-in** menu.

Warning: If the module is not deactivated after use, patch evaluation will be very slow. It is also very dangerous because it may cause PatchWork to report endlessly: "late Task". Note that the incoming MIDI data comes out of the patch in a compressed form. To decompress the data, see the modules: **midi-opcode, midi-chan, midi-data1, midi-data2,** and **midi-status**. See the on-line documentation for more information.

note-in



Input of m-data

No connection

Output of m-data

returns — Successive MIDI events

Input of note-in

in-box — Always connected to the output of **m-data**

patch — A patch

Optional input of note-in

delay — An integer

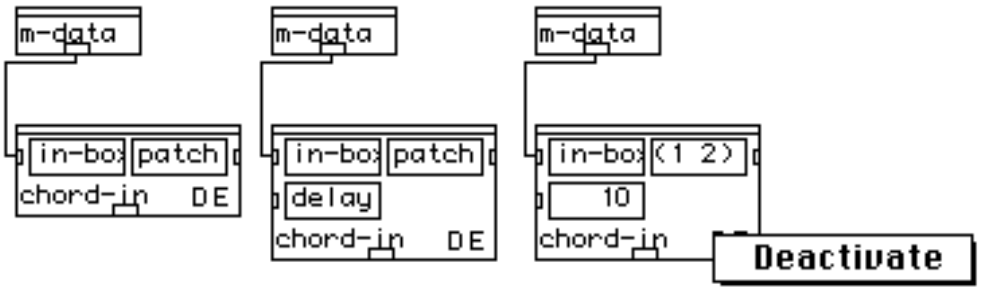
Output of note-in

returns — nil

The **note-in** and **m-data** modules are invoked simultaneously. They gather incoming MIDI data from the Macintosh serial ports. **note-in** filters out all events other than note-on messages. In order for the two modules to work, there must always be a loop with **note-in** at the bottom, **m-data** at the top, and some kind of patch dealing with the MIDI data in between. The optional input *delay* gives the delay time evaluation (in 100ths of a second) of the input patch. In order to start the loop that collects MIDI data, evaluate the **note-in** module. When you are finished collecting MIDI data, select "Deactivate" in the **note-in** menu.

Warning: If the module is not deactivated after use, patch evaluation will be very slow. It is also very dangerous because it may cause PatchWork to report endlessly: "late Task". See the on-line documentation for more information. See the on-line documentation for more information.

chord-in



Input of m-data

	No connection

Output of m-data

returns	Successive MIDI events

Input of chord-in

in-box	Always connected to the output of **m-data**
patch	A patch

Optional input of chord-in

delay	An integer

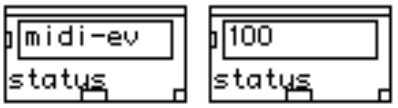
Output of chord-in

returns	nil

The chord-in and m-data modules work in a similar way as note-in. The chord-in module filters MIDI events other than note-on messages. The patch connected to *patch* is repeatedly evaluated for each new MIDI note-on event. The output of the **m-data** box is a chord object with all accumulated notes since the last box request. The optional input *delay* gives the delay time evaluation (in 100ths of a second) of the input *patch.* .

Warning: If the module is not deactivated after use, patch evaluation will be very slow. It is also very dangerous because it may cause PatchWork to report endlessly:"late Task". See the on-line documentation for more information.

status



Syntax

(c-pw-midi-in::status midi-ev)

Input

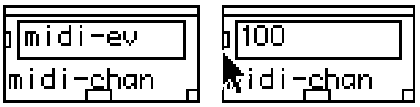
midi-ev	A number or list of compressed MIDI data

Output

returns	The status byte(s) in normal MIDI form

This module takes a compressed MIDI input, and returns the MIDI status byte.

midi-chan



Syntax

(c-pw-midi-in::midi-chan midi-ev)

Input

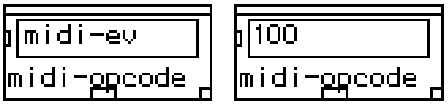
midi-ev	A number or list of compressed MIDI data

Output

returns	The channel number(s)

This module takes a compressed MIDI input, and returns the MIDI channel number.

midi-opcode



Syntax

(c-pw-midi-in::midi-opcode midi-ev)

Input

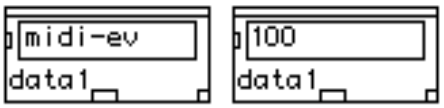
midi-ev	A number or list of compressed MIDI data

Output

returns	The first nibble of the status byte(s) converted to decimal. Note-on, for example, would return 9 (1001 converted to decimal)

This module takes a compressed MIDI input, and returns the MIDI opcode, that is, note-in, note-out, program change, etc.

data1



Syntax

(c-pw-midi-in::data1 midi-ev)

Input

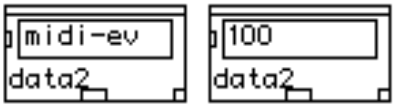
midi-ev	A number or list of compressed MIDI data

Output

returns	The first MIDI data byte (i.e., the one following the status byte), in decimal form

This module takes a compressed MIDI input, and returns the first MIDI data byte.

data2



Syntax

(c-pw-midi-in::data2 midi-ev)

Input

midi-ev	A number or list of compressed MIDI data

Output

returns	The second MIDI data byte , in decimal form

This module takes a compressed MIDI input, and returns the second MIDI data byte.

get-note-slots

Syntax

(c-get-note-slots::get-note-slots object slots)

Input

object	A note object, chord object, chord sequence object or rtm object.
slots	A slot (data field) or list of slots which contain note information (midics, offset times, channel, etc.)

Output

returns	The value(s) for the requested slots

get-note-slot is similar to **get-slot**, except that it only returns information concerning notes. This data includes: midic, dur, vel, chan, offset-time, and comm.

Note that when the module is used with a chord sequence object, the requested field is returned for each note of each chord in the object, and the slots of each chord is paired in a list with the chord's attack time.

set-note-slots

Syntax

(c-get-note-slots::set-note-slots object slots values)

Input

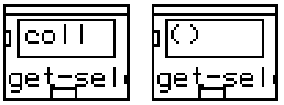
object	A note object, chord object, or chord sequence object.
slots	A slot (data field) or list of slots which contain note information (midics, offset times, channel, etc.)
values	Any data type(s)

Output

returns	the value(s) for the requested slots

The **set-note-slot** module is similar to **set-slot**, except that it only assigns information concerning notes. The slots it assigns includes: midic, dur, vel, chan, offset-time, and comm. The module changes the contents of the object for the given *slot*(s) by assigning them the given *value*(s).

get-sel



Syntax

(c-get-note-slots::get-sel coll)

Input

coll	A chord sequence

Output

returns	A list of chord objects

get-sel retrieves the list of chords previously selected in the editor. The module's input must always be connected directly with the output of a **chord-seq** module.

Index