


- Research reports
- Musical works
- Software

PatchWork

Tutorial

Second English Edition, October 1996

IRCAM  Centre Georges Pompidou

© 1996, Ircam. All rights reserved.

This manual may not be copied, in whole or in part,
without written consent of Ircam.

This manual was written by Jacopo Baboni Schilingi under the supervision of Andrew Gerzso, and was produced under the editorial responsibility of Marc Battier, Marketing Office, Ircam.

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

Second edition of the documentation, October 1996.
This documentation corresponds to version 2.5 or higher of PatchWork.

Apple Macintosh is a trademark of Apple Computer, Inc.
PatchWork is a trademark of Ircam.

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

IRCAM Users' group

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

Département de la Valorisation

Ircam

Place Stravinsky, F-75004 Paris

Tel. (1) 44 78 49 62

Fax (1) 42 77 29 47

E-mail: bousac@ircam.fr

Send comments or suggestions to the editor:

E-mail: bam@ircam.fr

Mail: Marc Battier,

Ircam, Département de la Valorisation

Place Stravinsky, F-75004 Paris

Contents

Résumé	6	Tutorial 28	109
Getting started	7	Tutorial 29	114
What is PatchWork?	7	Tutorial 30	118
Structure of folders of PatchWork	9	Tutorial 31	120
Invoking the on-line documentation	10	Tutorial 32	123
Programming with PatchWork	13	Tutorial 33	127
Launching PatchWork	13	Tutorial 34	130
The Listener window	14	Tutorial 35	133
The PatchWork window	16	Tutorial 36	135
Programming with modules	18	Tutorial 37	138
How to invoke a module	19	Tutorial 38	141
How to edit a module	21	Tutorial 39	144
How to connect modules	24	Tutorial 40	150
How to evaluate	27		
Structure of data	29	Index	156
Modules for representing data and music	33		
Tutorials	34		
Opening and modifying tutorials	34		
Tutorial 1	36		
Tutorial 2	39		
Tutorial 3	41		
Tutorial 4	43		
Tutorial 5	45		
Tutorial 6	47		
Tutorial 7	49		
Tutorial 8	52		
Tutorial 9	54		
Tutorial 10	56		
Tutorial 11	58		
Tutorial 12	62		
Tutorial 13	65		
Tutorial 14	67		
Tutorial 15	70		
Tutorial 16	72		
Tutorial 17	75		
Tutorial 18	77		
Tutorial 19	79		
Tutorial 20	82		
Tutorial 21	85		
Tutorial 22	91		
Tutorial 23	95		
Tutorial 24	99		
Tutorial 25	102		
Tutorial 26	104		
Tutorial 27	106		



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.

Résumé

Ce manuel présente une série d'exemples commentés, conçus pour faciliter la prise en main de PatchWork. Les quarante exemples sont organisés en ordre progressif. Le patch est représenté à l'écran, et il est décrit au moyen d'un organigramme qui figure le flot de données et de commandes.

Vous trouverez les fichiers des patches sur votre CD-ROM ; chacun porte le nom « tutorial.pw », précédé du numéro du tutorial, de 1 à 40 ; ex. : 01-tutorial.pw. Il est conseillé de travailler sur une copie, surtout si vous comptez modifier le patch afin de mieux l'expérimenter.

Getting started

What is PatchWork?

PatchWork is an interactive graphical environment for music composition, sound synthesis and Midi processing. It provides tools for the composer for generating and manipulating musical data with a wide range of musical applications. These tools are capable of representing, transforming, editing, exporting, importing and playing musical material during the creation of musical compositions.

PatchWork is entirely implemented in Common Lisp for the Macintosh. It works by using data modules and musical modules. What we call a “Patch” is a visual arrangement of PatchWork modules (that resemble little boxes) in a window. That is why PatchWork is called a graphical programming environment.

Modules are boxes that contain inputs and outputs which can be connected to each other.

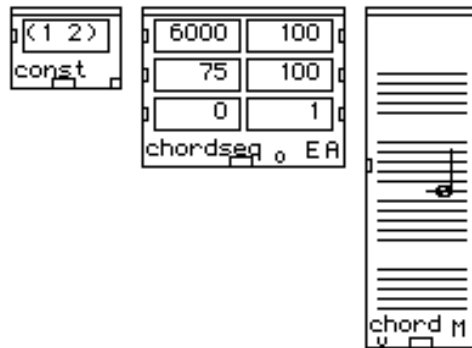


Figure 1

The module is the basic unit of PatchWork. Its graphical representation is a rectangular box and every box is associated with a specific function¹. In other words, a module is a graphic representation of a specific Common Lisp function. PatchWork modules have various types: general Lisp Function (sum, difference, quotient, product...), computation

1. See chapter 4 of your Introduction manual.

(logic, set operations...), conversion between numeric and symbolic values, generation of musical structures (interpolation of patterns, change of melodies, creation of series and spectra...) and graphical editing of curves and normal musical notation.

An ensemble of connected boxes is a patch. A graph of the connection between these modules defines the specific way of editing musical data.

A PatchWork control interface with synthesis programs such as Csound and Chant adds supplementary tools for controlling sound synthesis.

Since PatchWork is an open environment, it provides composers with the possibility of developing personal graphical applications that we call libraries. A library is an ensemble of Common Lisp functions programmed directly by the composers and it appears like a Macintosh menu. Every library can be designed to give to PatchWork a personal touch, according to our own aesthetic and programming tastes.

Structure of folders of PatchWork

To start with PatchWork it is necessary to have properly made the software installation on your hard disk. The folder structure of PatchWork must be like the one you will find described in the PatchWork Newsletter.

Here, as an example, is the folder structure for PatchWork version 2.6.3. Your installation of PatchWork may have a different structure, depending on the version of the software you are working with.

```
graph TD; Root["PatchWork 2.6.3 ppc"] --> 40Tutorials["40-Tutorials"]; Root --> BuildImage["Build-image"]; Root --> Documentation["Documentation"]; Root --> Images["Images"]; Root --> PWcode["P'W'-code"]; Root --> PWinits["P'W'-inits"]; Root --> PWmodifs["P'W'-modifs.lisp"]; Root --> UserLibrary["user-library"]; Root --> ReadMe["ReadMe PatchWork 2.6.3ppc"];
```

Figure 2

Invoking the on-line documentation

PatchWork already has different manuals (mainly the Introduction and Reference manuals, available on your CD-ROM) that provide complete documentation with all kinds of information you need for using the program.

Moreover, PatchWork has an on-line help and an on-line documentation. Don't forget to use them whenever you need to. To make the documentation window of a box to appear, select the desired box you need information about with mouse. When you have selected the box, type "d" on the keyboard.

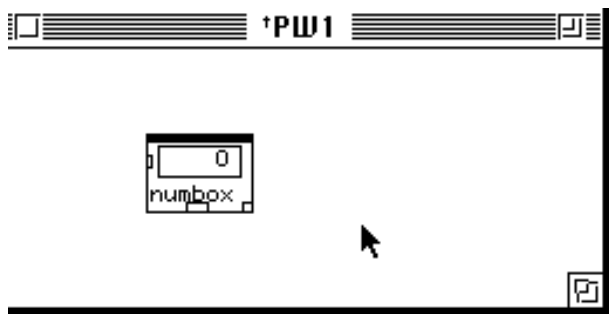


Figure 3

The following dialog box will appear if you have not installed the Common Lisp on your computer.

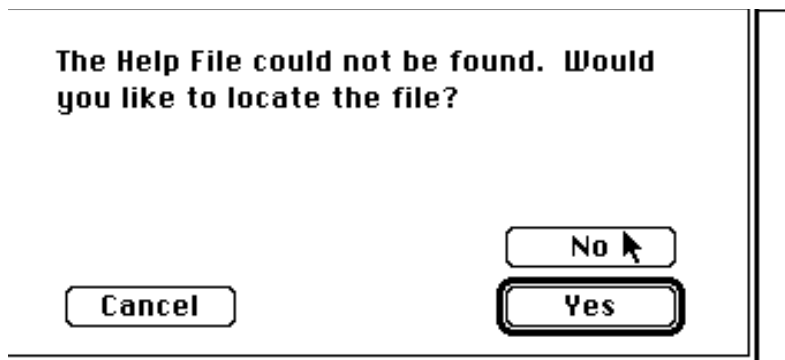


Figure 4

In this case, you should respond “NO” to the question to access the PatchWork on-line documentation.

You can also select the PatchWork on-line documentation by using the PWoper menu as shown in the following figure.¹

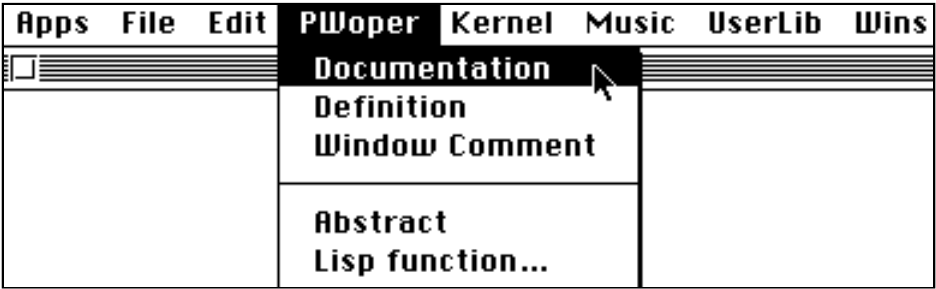


Figure 5

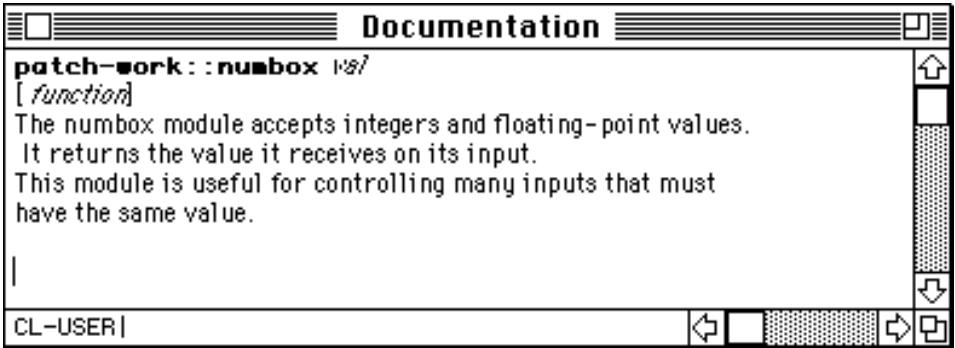


Figure 6

1. See also chapter 4 of your Introduction manual.

Every box of PatchWork has a proper Tutorial Patch on-line already done in which you can find explanations concerning the selected box. To invoke an on-line Tutorial select the box you want to explained¹ and then type "t" on the keyboard. A Tutorial patch will appear with some practical examples of how to use the selected box (it's usually in French).

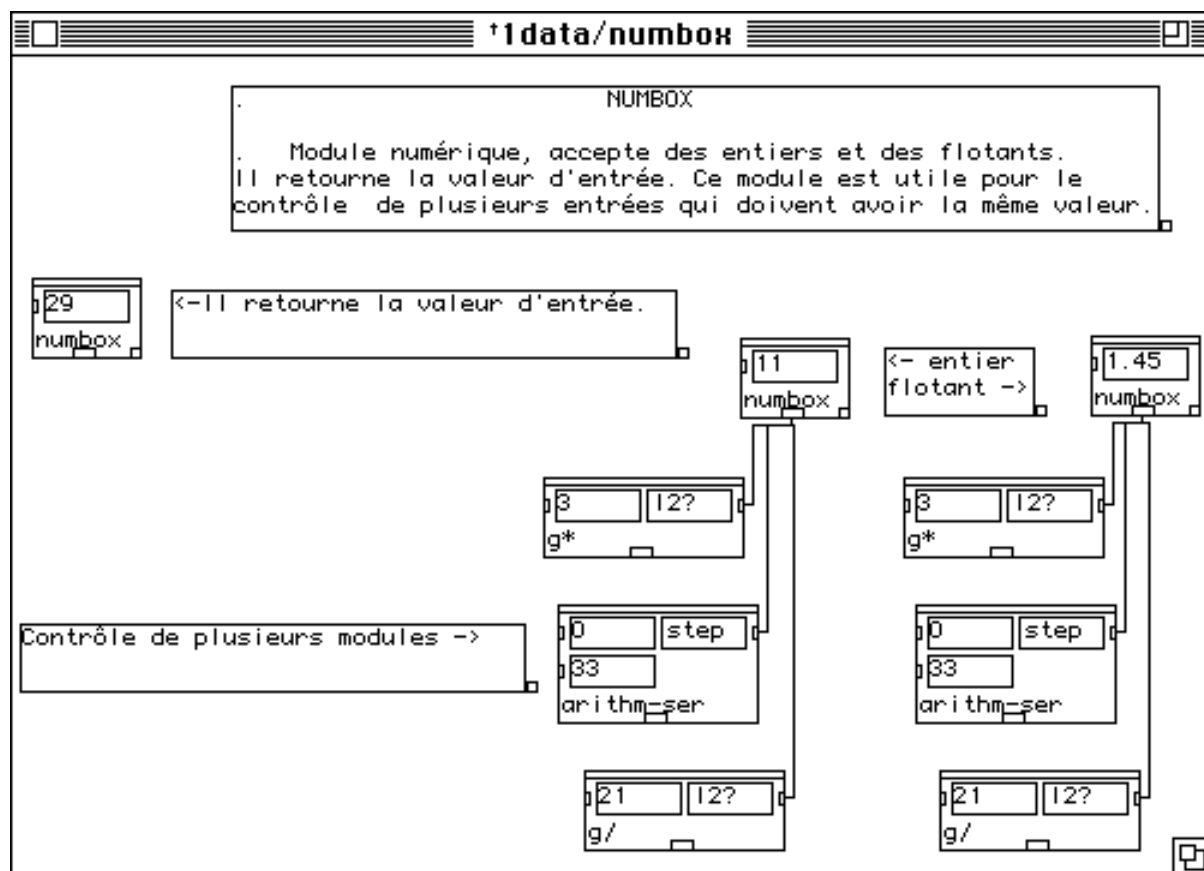


Figure 7

1. See figure number 3.

Programming with PatchWork

Lauching PatchWork

To launch PatchWork, double-click on the Macintosh icon of PW-Music.image in your “Images” folder. This is similar to any Macintosh application.



Figure 8

Once the PatchWork application has started, you will see a new window. This window is the Common Lisp interface. It is called the Listener.

It is very important not to start by clicking on a PatchWork file before having properly loaded the PatchWork application. If you click on a file the patch will not be opened with the PatchWork graphical interface but in the text form of the Common Lisp language. That is why we ask you to follow this chapter step by step.

The Listener window

This is the place where PatchWork prints out messages to the user. When you start up the PW-Music.image, PatchWork prints in the Listener window a list of files that have been loaded as in the window shown in the following example.

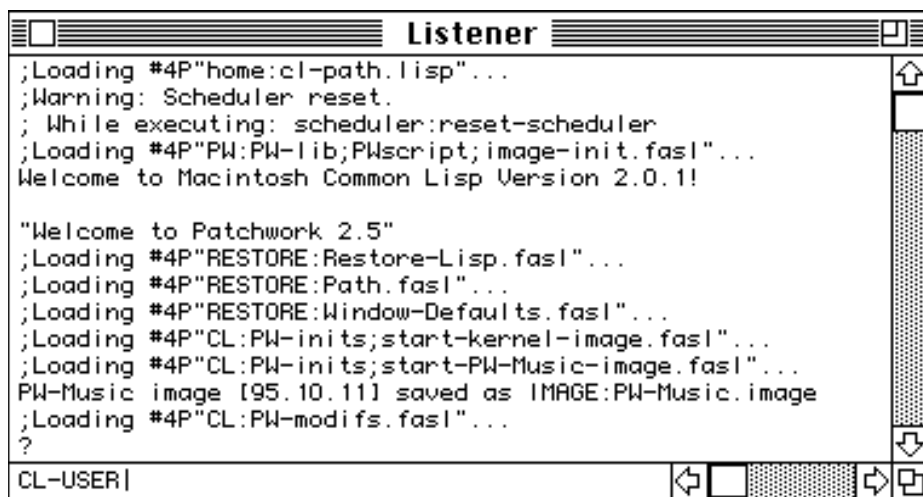


Figure 9

It is in this window that PatchWork will print out any error messages, which are the results of any operations or any warnings while you are editing or running a patch.

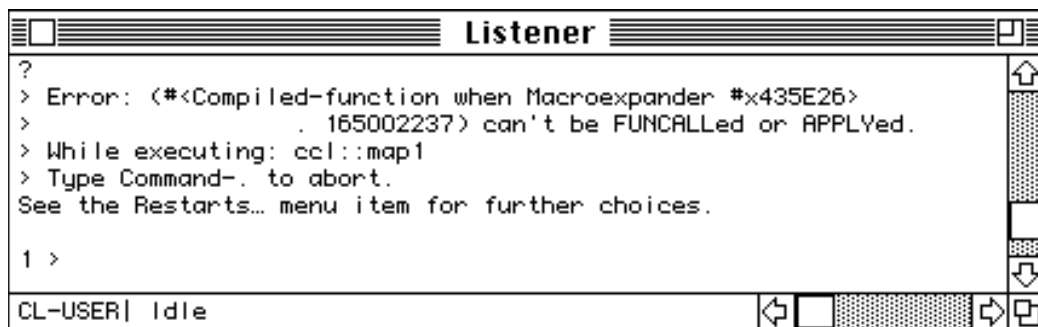


Figure 10

This window is the link between PatchWork and Macintosh Common Lisp. In other words, you can see in the listener window the the result of every step while programming your patch and whether or not your patch is working properly.

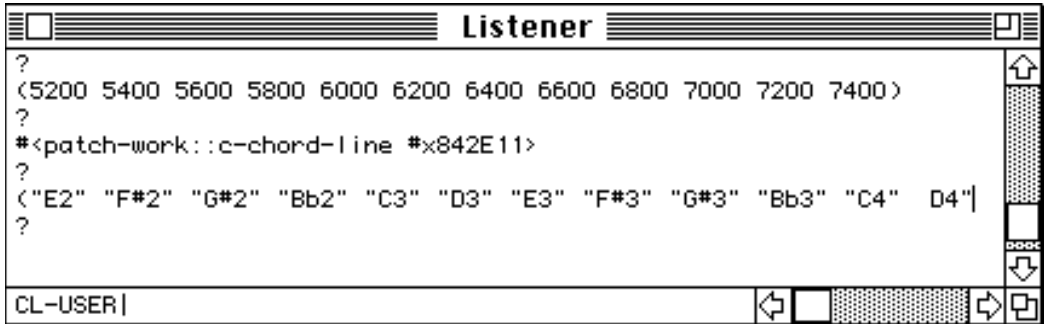


Figure 11

The PatchWork window

When you see the question mark in the Listener window it means that PatchWork is completely loaded and that you can start work with PatchWork.

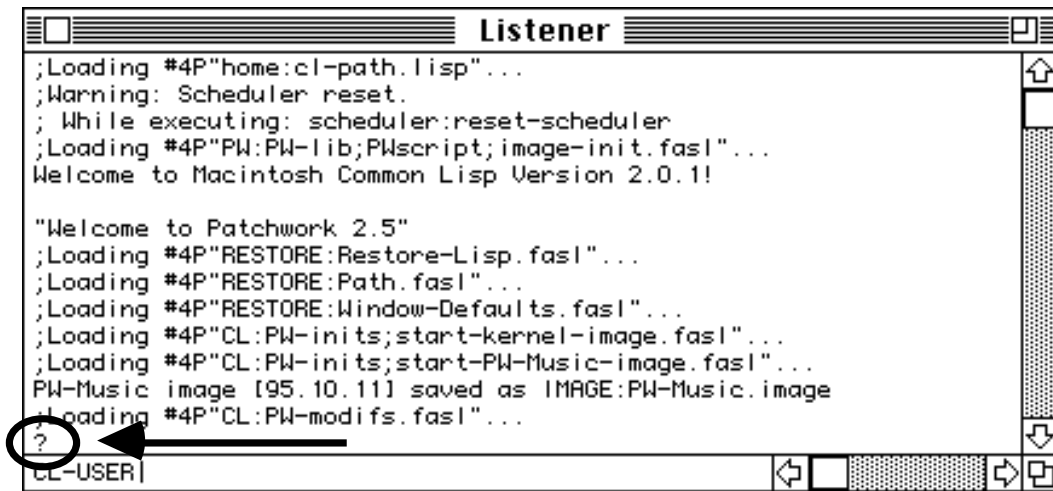


Figure 12

Once the PatchWork application is loaded, you can begin programming by selecting PW in the Apps menu bar¹.

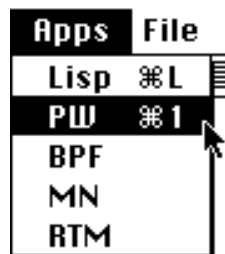


Figure 13

1. See also chapter 3 of your Introduction manual.

The menu bar Apps contains the options for moving between the Listener window and the PatchWork window. When you use the Listener window you are in the Common Lisp program and the menu bar you use is identical to the standard Common Lisp menu bar. When you have selected PW in Apps menu bar, a new empty PatchWork window will appear and you can start programming your own patch.

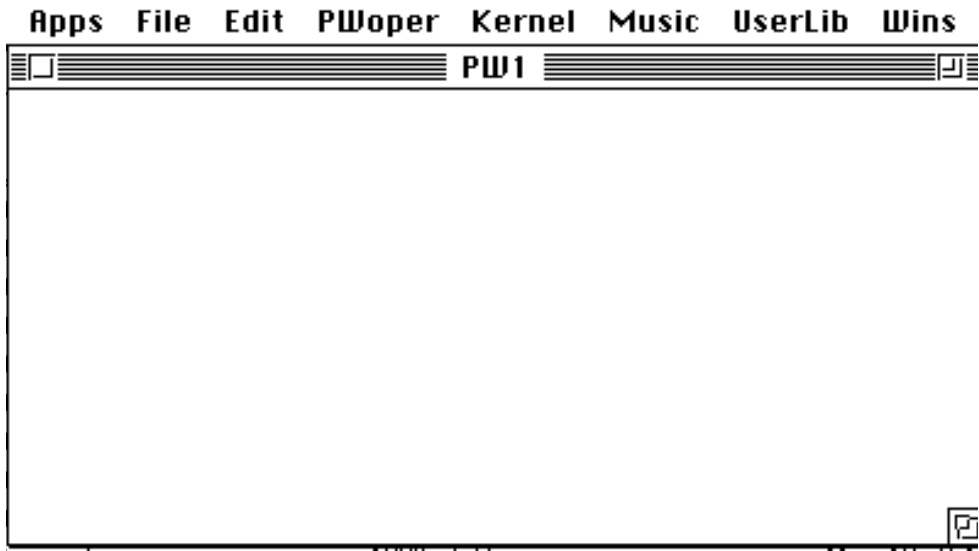


Figure 14

Programming with modules

PatchWork is an interactive environment that provides a graphical way of programming by connecting boxes. A box (or module) is just a graphical representation of a specific Common Lisp function. To make your own programming patch in PatchWork, you should invoke different modules (sometimes also called boxes). Once on the PW window, you can connect them together.

That is, if you want to calculate the sum of 6000 and 700, you need a box to write the input numbers: 6000 and 700. Then, you need another box for obtaining the desired sum. Now let's do together, step by step, this simple example: how to make $6000 + 700$ just using PatchWork boxes.

We suggest you to read chapter 4 of the Introduction manual for the PatchWork menus structure, before going on.

How to invoke a module¹

To program with PatchWork boxes we must invoke boxes in our PatchWork window. To invoke the modules from the PatchWork menus, place the cursor on the desired menu and select, by dragging the cursor, the box you want to invoke².

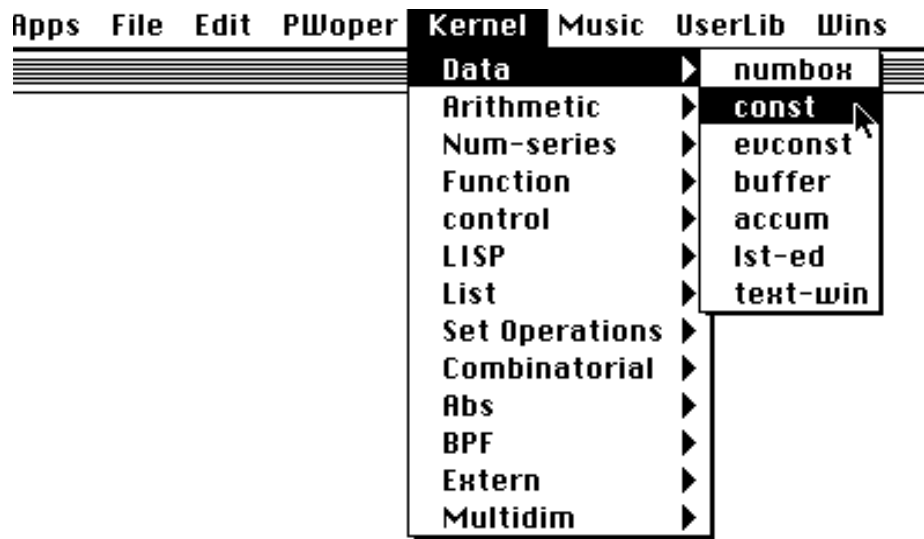


Figure 15

The cursor will turn into the one shown in figure16.

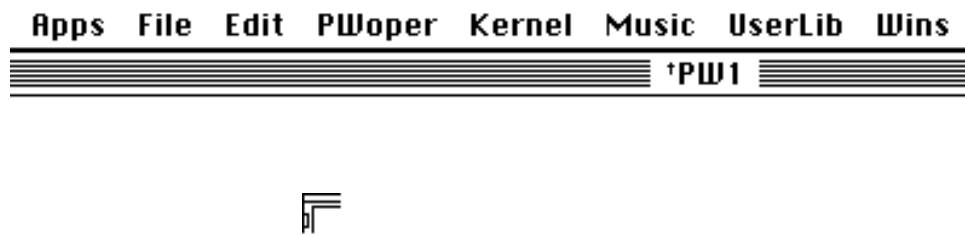


Figure 16

1. See also chapter 4 of PatchWork Introduction manual.
2. See also the PatchWork Newsletters.

Then click in the PatchWork window where you want to place the module.

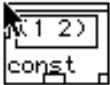


Figure 17

Now you can move the box wherever you want by selecting and dragging it with the mouse. You can do the standard Macintosh operations, like Cut, Copy, Duplicate, Select All... by selecting these functions in the Edit menu¹.

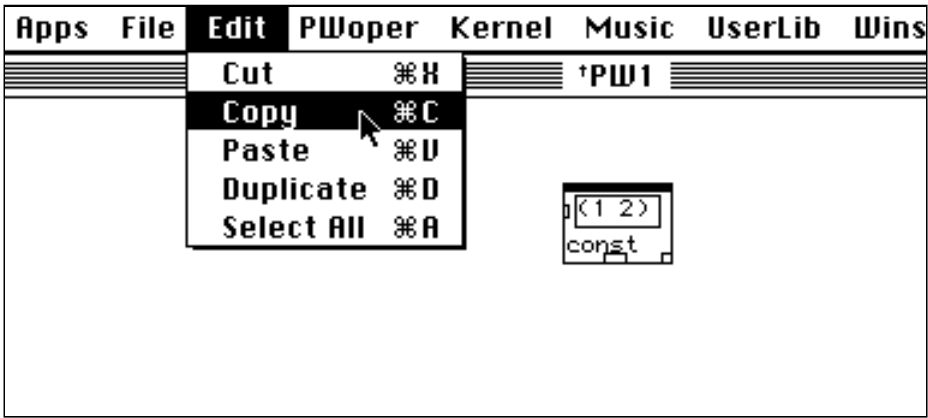


Figure 18

1. See also chapter 3 of your Introduction manual.

How to edit a module

In order to solve our simple exercise ($6000 + 700$) with PatchWork boxes, we now need to edit our inputs in a box in order for it to recognise numbers. For this reason we need two boxes like the “const”¹ one (figure 17). To edit the numbers 6000 and 700 in our PatchWork window we begin by invoking a modules “const” from Kernel menu bar.

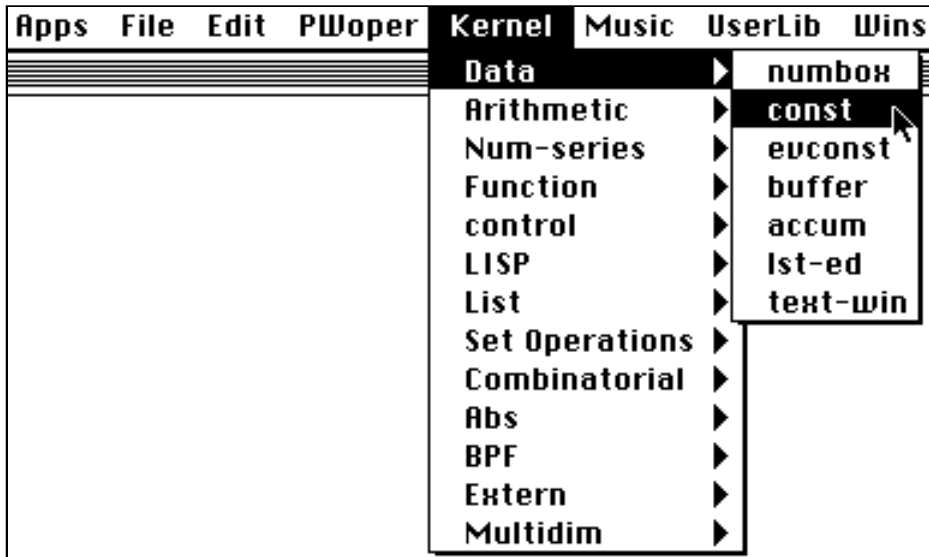


Figure 19

Then, when you click where you want the box to be placed, the “const” box will appear on your window. Since we need two “const” boxes, you can select the box “const” we have already invoked and then duplicate it by using the menu Edit.

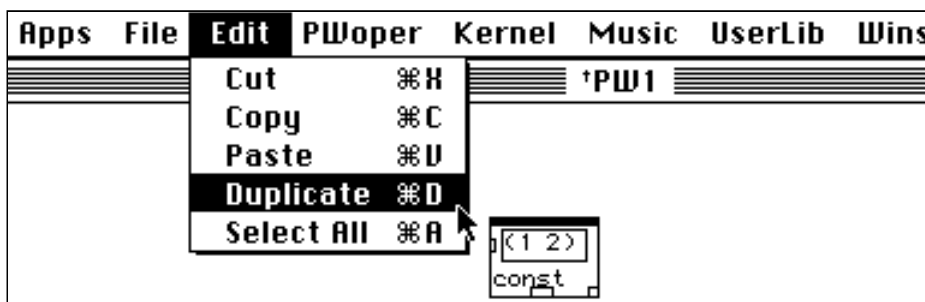


Figure 20

1. See also “Const” in chapter Data Modules of your Reference manual.

You'll then have the following boxes in the PatchWork window:

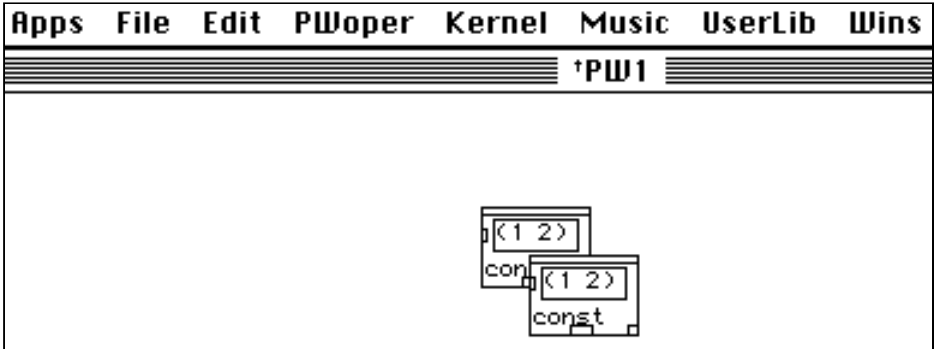


Figure 21

Now, in order to finish this example, select the two boxes

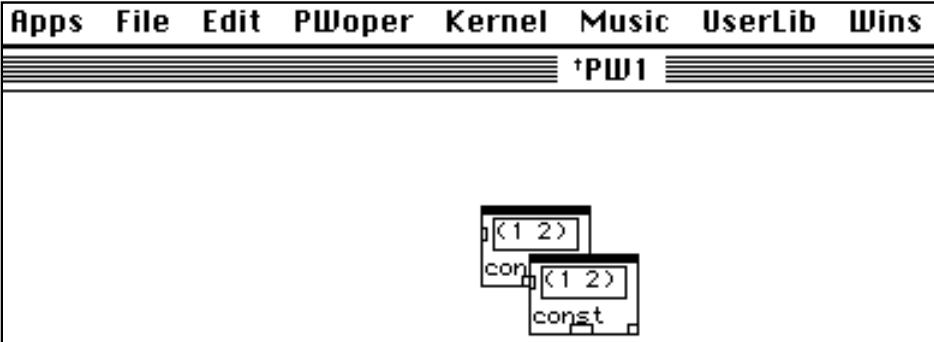


Figure 22

and type “y” to align horizontally the boxes.

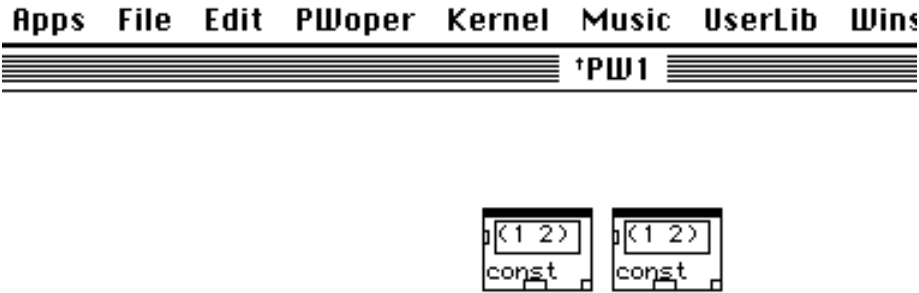


Figure 23

Now we have to edit our inputs in the two "const" boxes. Double click in input rectangle of your "const" box. This operation will open the input box for editing, allowing the new value to be entered directly from the keyboard.

When you have double clicked in the box input, the module becomes editable as in the following figure.

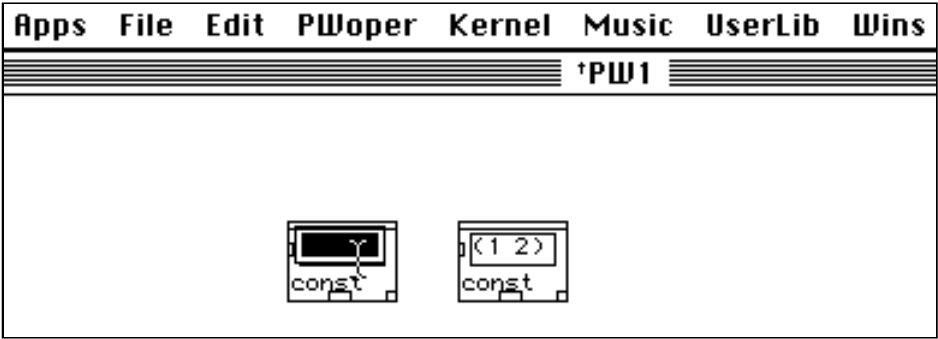


Figure 24

Now you can enter the number 6000 followed by typing "return" on the keyboard. Repeat the same operation for the second box in order to obtain the following result.

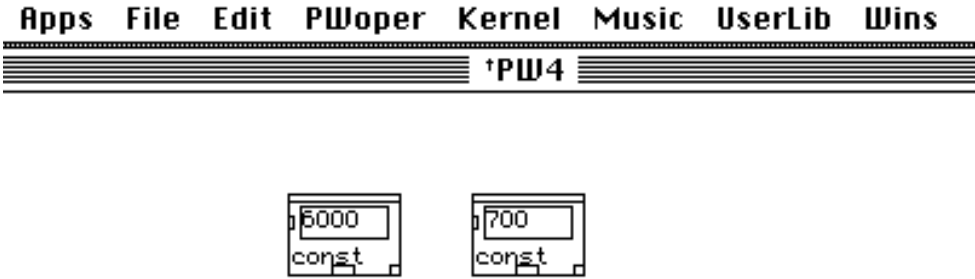


Figure 25

How to connect modules¹

Now you need another box that calculates the sum between the “const” box with number 5 and the “const” box with number 7. You have to invoke the module “g+” by selecting it in the Kernel menu.²

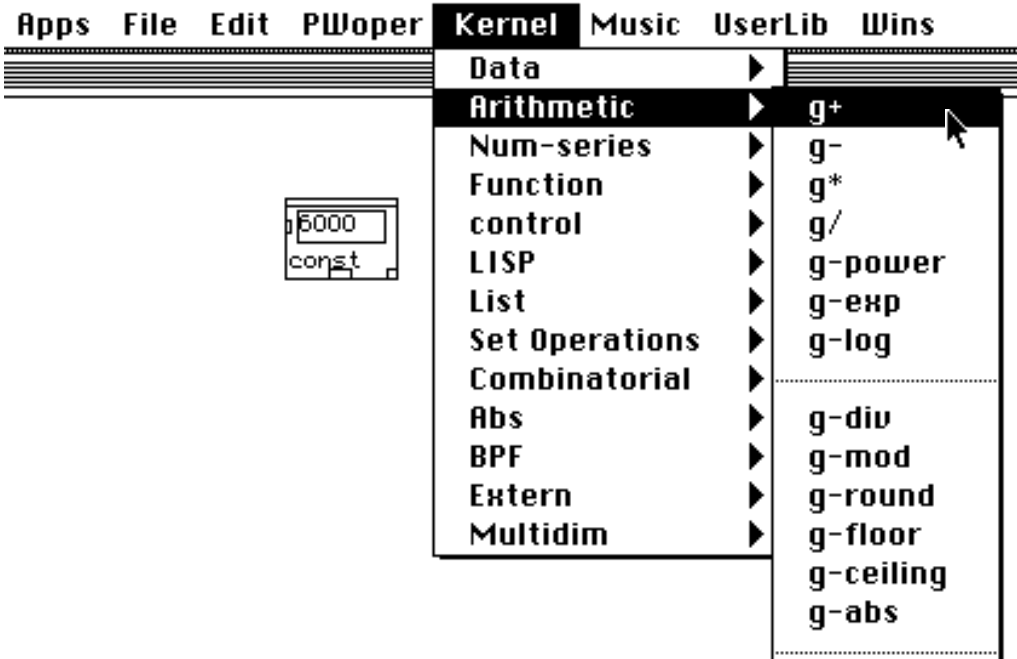


Figure 26

Now you must have on your window the following three boxes:

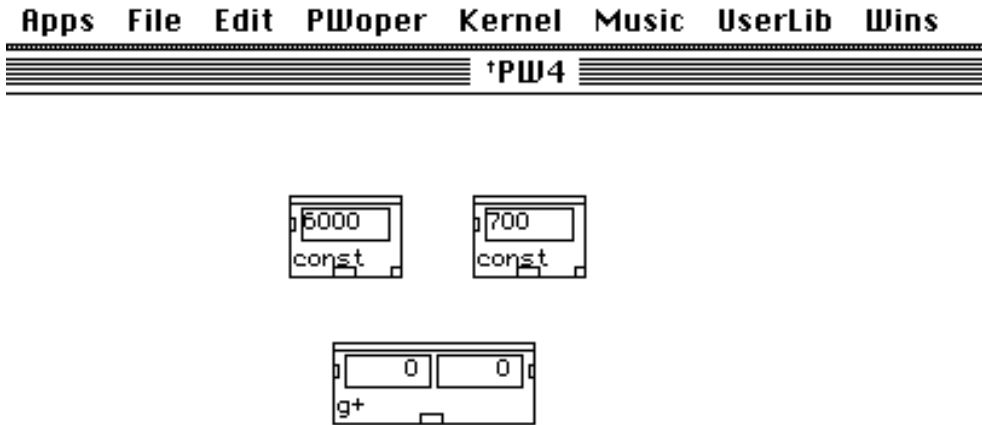


Figure 27

1. See also chapter 3 of your Introduction manual.
2. See also chapter “g+” in “Arithmetic Modules” of your Reference manual.

Now you have to connect the two "const" boxes with the sum one¹. To do this, click in the output rectangle of the "const" module containing the number 5 and drag the mouse, while keeping the botton pressed, to the left input rectangle of the sum box "g+".

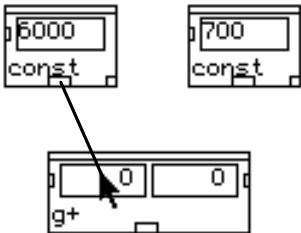


Figure 28

Now repeat the same operation for the second "const" box with the right input of the "g+" box.

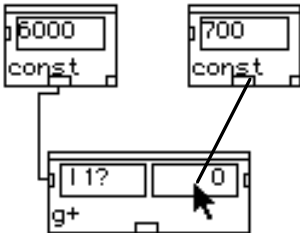


Figure 29

1. See also chapter 4 of your Introduction manual.

If you have done everything correctly, you will now have the following patch:

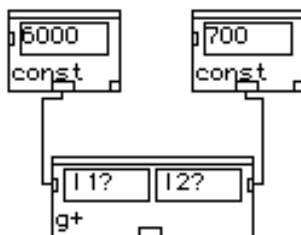


Figure 30

How to evaluate¹

Now you are ready to evaluate the little patch done with the PatchWork boxes. Select the “g+” box by clicking on the small output rectangle at the bottom of the box.

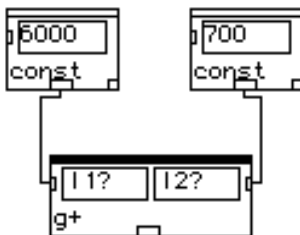


Figure 31

Type “v” on the computer keyboard. In the Listener window will appear the solution to the sum of 6000 and 700.



Figure 32

1. See also chapter 4 of your Introduction manual.

To evaluate a patch you can also “Option-click” on the output rectangle of the last module in the chain of boxes¹. In this case the last module in the chain is “g+”.

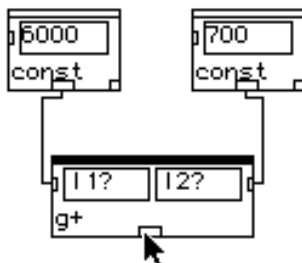
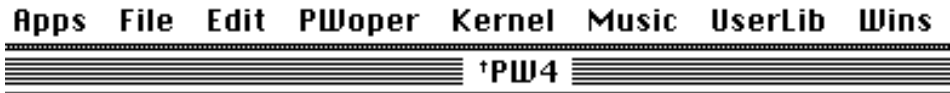


Figure 33

1. See also chapter 4 of your Introduction manual.

Structure of data

Input and output data types: atom, list, list of lists, objects...

In PatchWork we have different types of inputs and outputs. For this reason it is fundamental to know what kind of data a box can accept and send out.

In PatchWork we can have inputs like single elements that we call “atoms”, or a list of elements, a list of lists of elements and objects. All these terms are derive from the Common Lisp language. For example, in PatchWork there are boxes that accept a list or a list of lists in the input rectangle and that produce objects. Or we can have a box that accepts only objects in the input rectangle and that returns lists. Now let’s see how many kinds of data we have and how they appear in the PatchWork windows and in the Listener window.

An atom is like a single element and it can be either a word, the letter of the alphabet, a number or any kind of symbol. For numbers, PatchWork can accept a variety of types like integers, rational and floating-point numbers. In PatchWork we have a lot of modules that accept atoms.

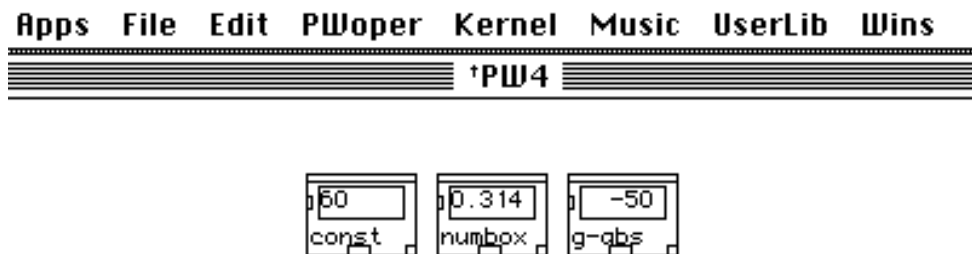


Figure 34

An atom looks like this in the Listener Window:

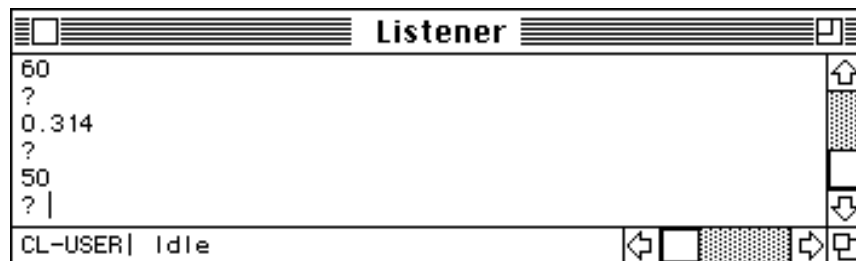


Figure 35

In Common Lisp and in PatchWork a list is a collection of atoms enclosed in parentheses. In the boxes they appear like this:

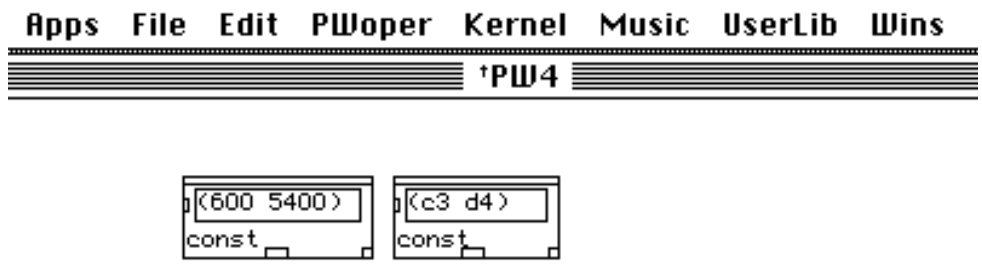


Figure 36

In the Listener window a list of elements appears like this:

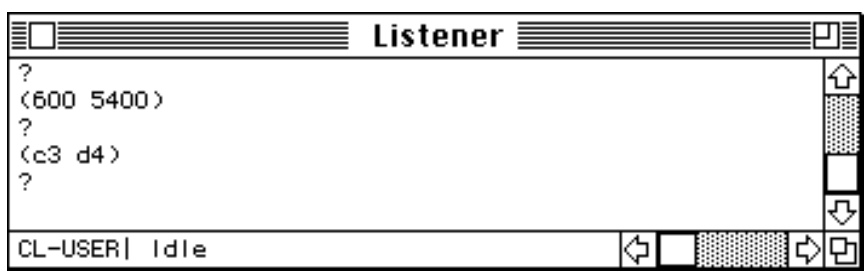


Figure 37

A list of lists is an ensemble of many lists enclosed in parentheses. This is what it looks like in a PatchWork window:

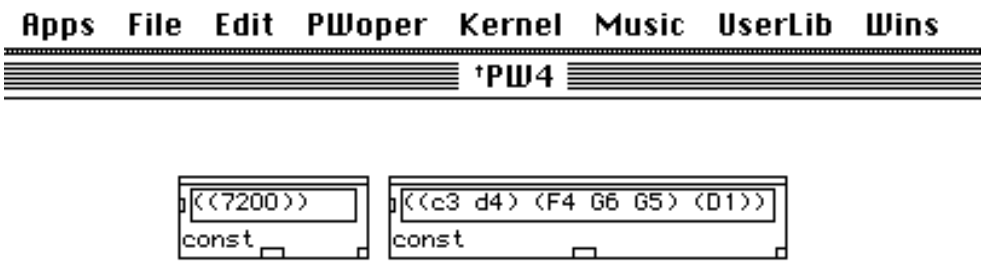


Figure 38

And like this in a Listener window:



Figure 39

An object is something more complex so we won't explain it in this manual. For this reason we suggest you consult a Common Lisp programming manual and study Common Lisp at the same time that you learn PatchWork.

There are many modules that send out data for objects like the following ones:

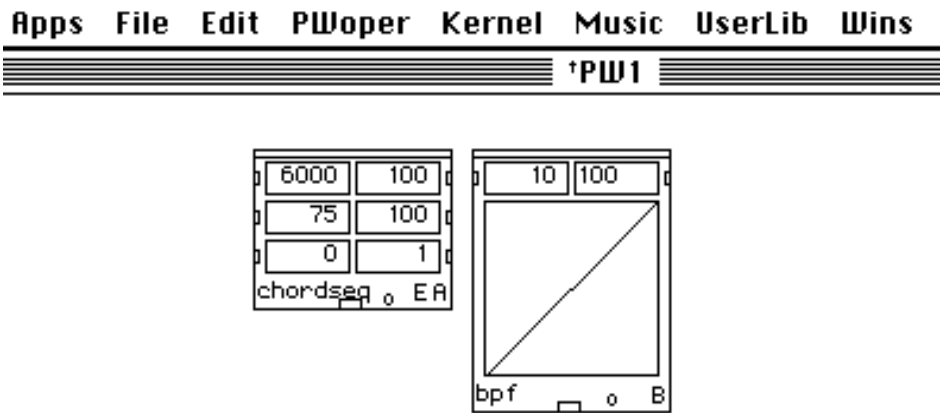


Figure 40

In the Listener window objects appear as in the window shown below.



Figure 41

When you start connecting boxes in PatchWork you often need to know the types of input a module can receive. To know the inputs of a box, “Command-click” on the lower left corner of the box.



Figure 42

In the Listener window will appear the input types accepted by the selected box.



Figure 43

To know the outputs of a box “Option-click” on the lower left corner of the box and in the Listener window you’ll see the corresponding output types sent out by the selected box.



Figure 44

Modules for representing data and music¹

In PatchWork we have modules for printing out data and other modules for music notation. Kernel is the menu for representing and editing data. It contains standard Common Lisp pre-defined functions like sum, division, multiplication etc. and general functions for generating and manipulating data (interpolation, random generators, etc.).

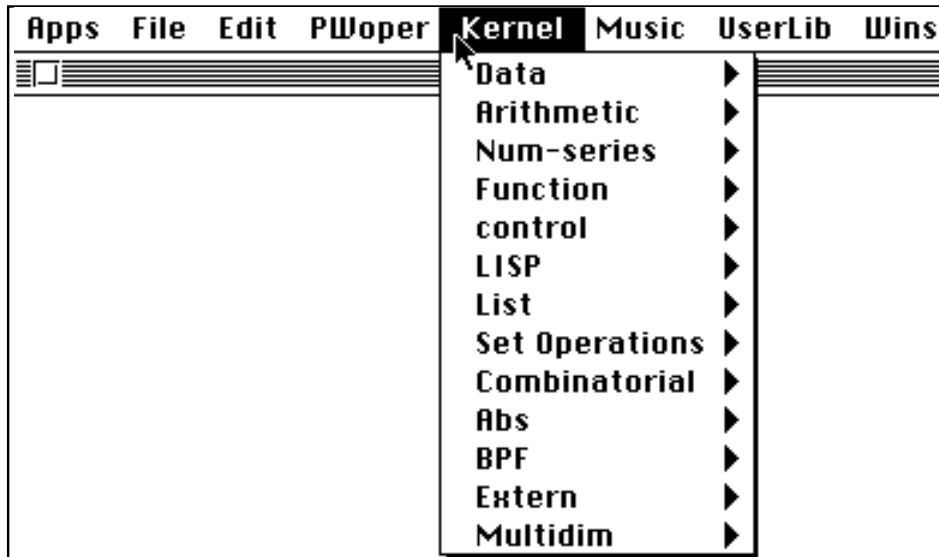


Figure 45

Finally, the PatchWork Music menu contains the musical interface between data and music notation and many specific options designed for musical operations, like displaying or editing sequences of notes, for conversions of values between different units and for Midi data.

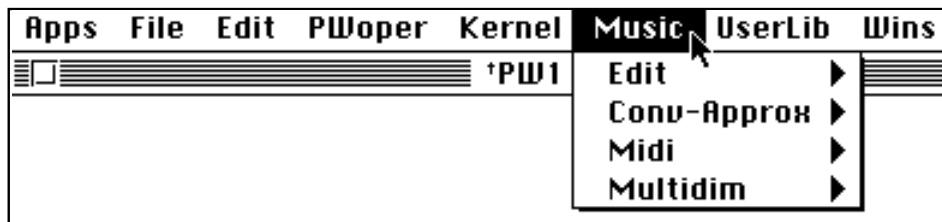


Figure 46

1. See also chapter 4 of your Introduction manual.

Tutorials

Opening and modifying tutorials

Now let's start studying PatchWork by going through 40 tutorials of gradually increasing difficulty. We suggest you open one tutorial at a time (for instance, you might like to start with tutorial number 1).

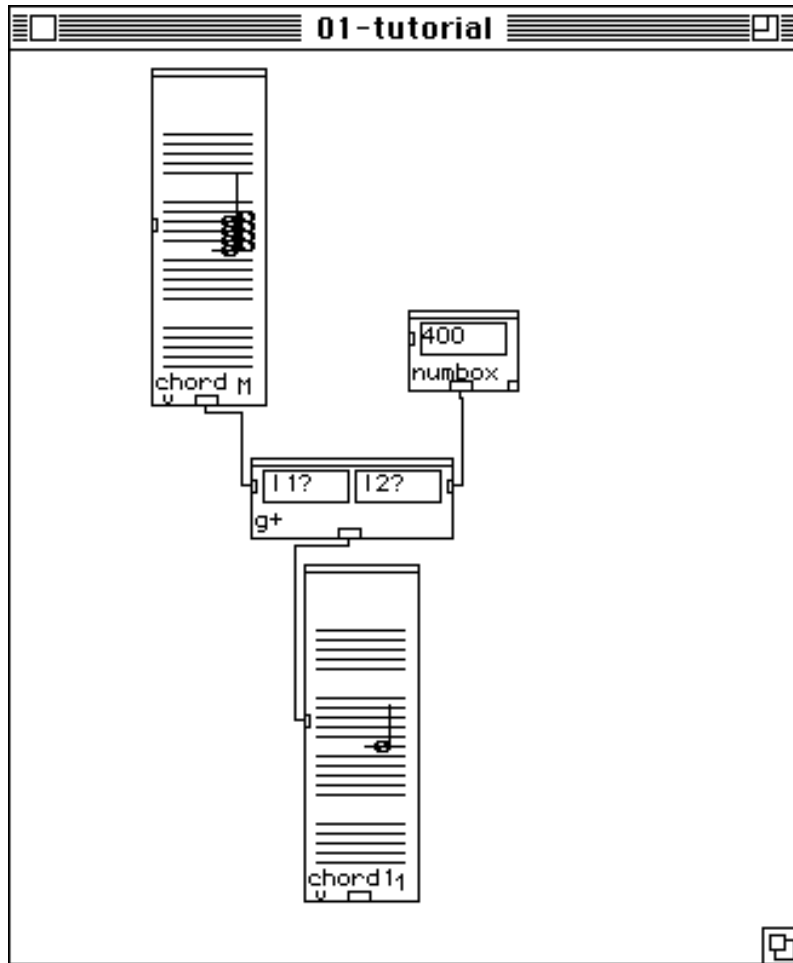


Figure 47

Save it under another name in a separate folder, so that you can modify your own copy at will.

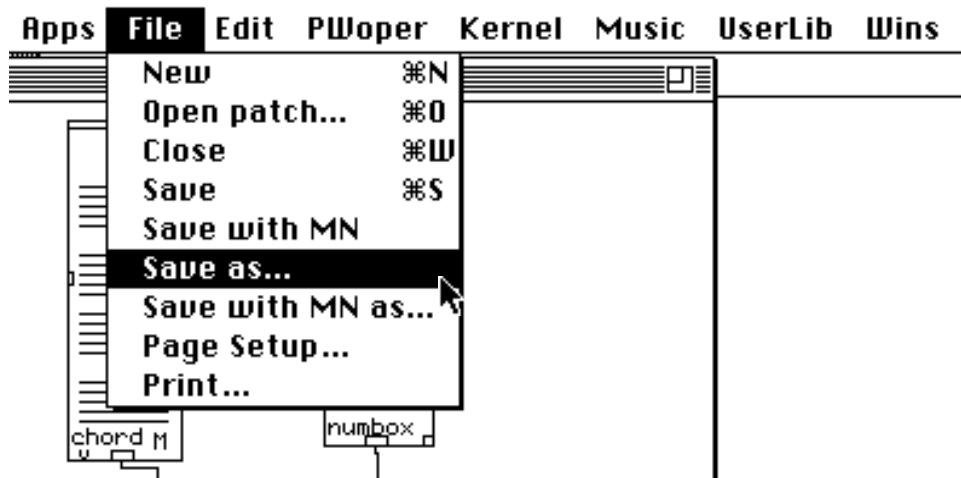
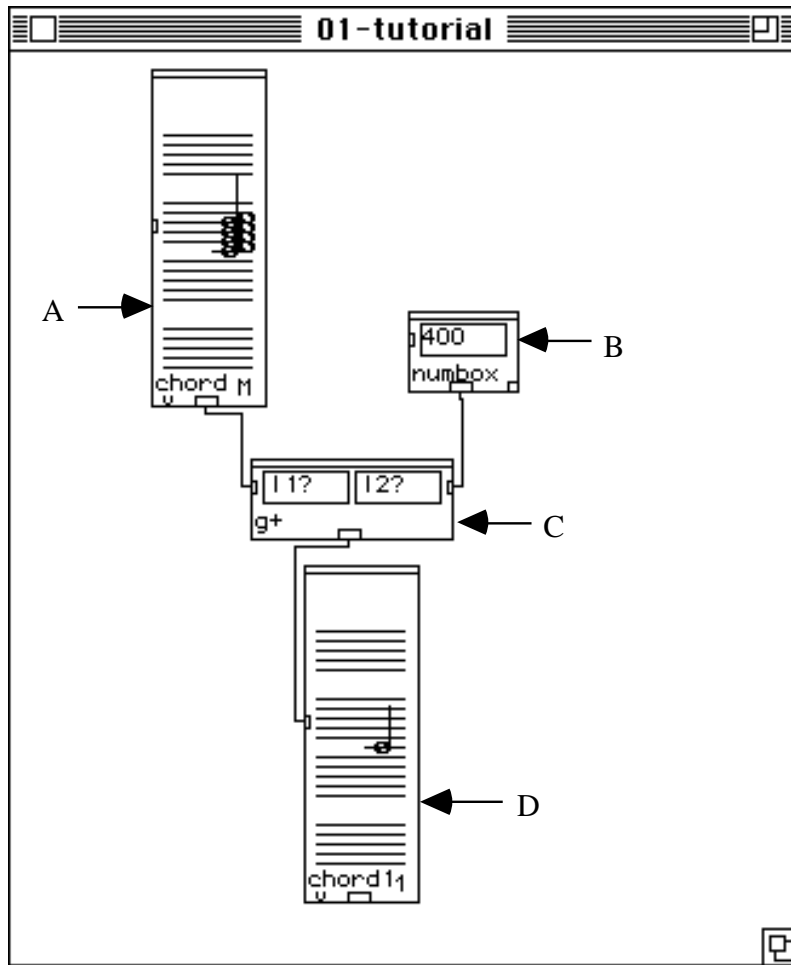


Figure 48

In this way you can change and modify any tutorial, following the instructions given in this manual, and at the same time keep a copy of the original version. Good luck!

Tutorial 1

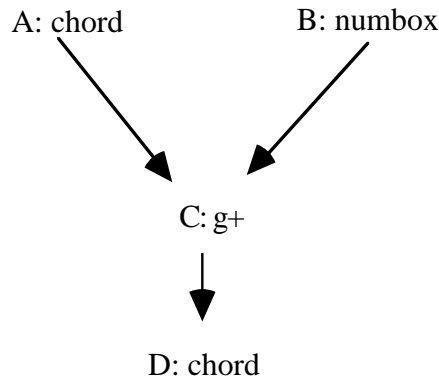
Basic operation: transposition of notes.



Description

This patch transposes the notes in chord box (A) by adding (C) the value you have entered in numbox (B) to every note.

Patch structure

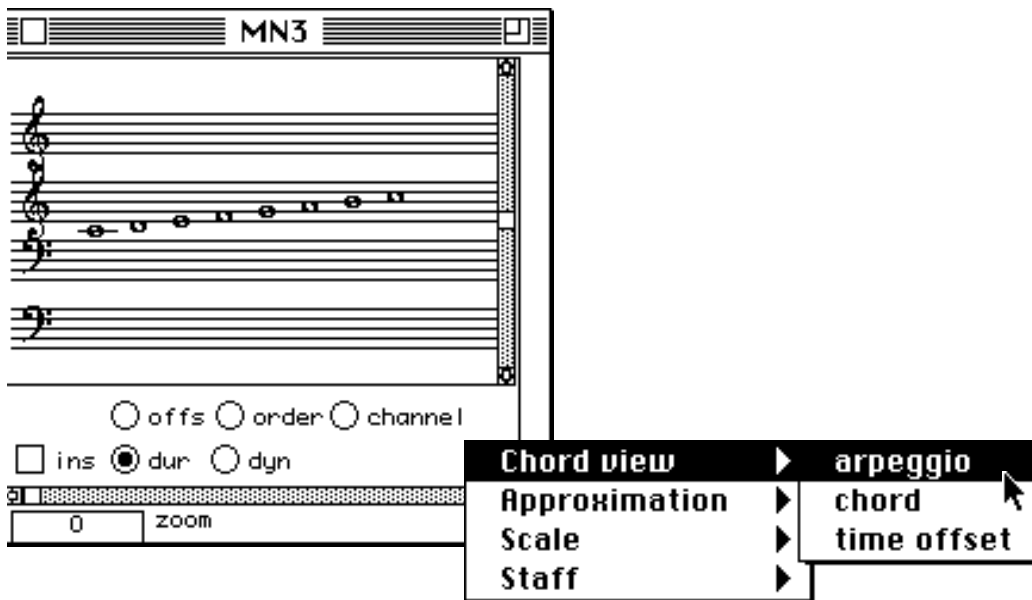


Boxes used

chord, g+, numbox.

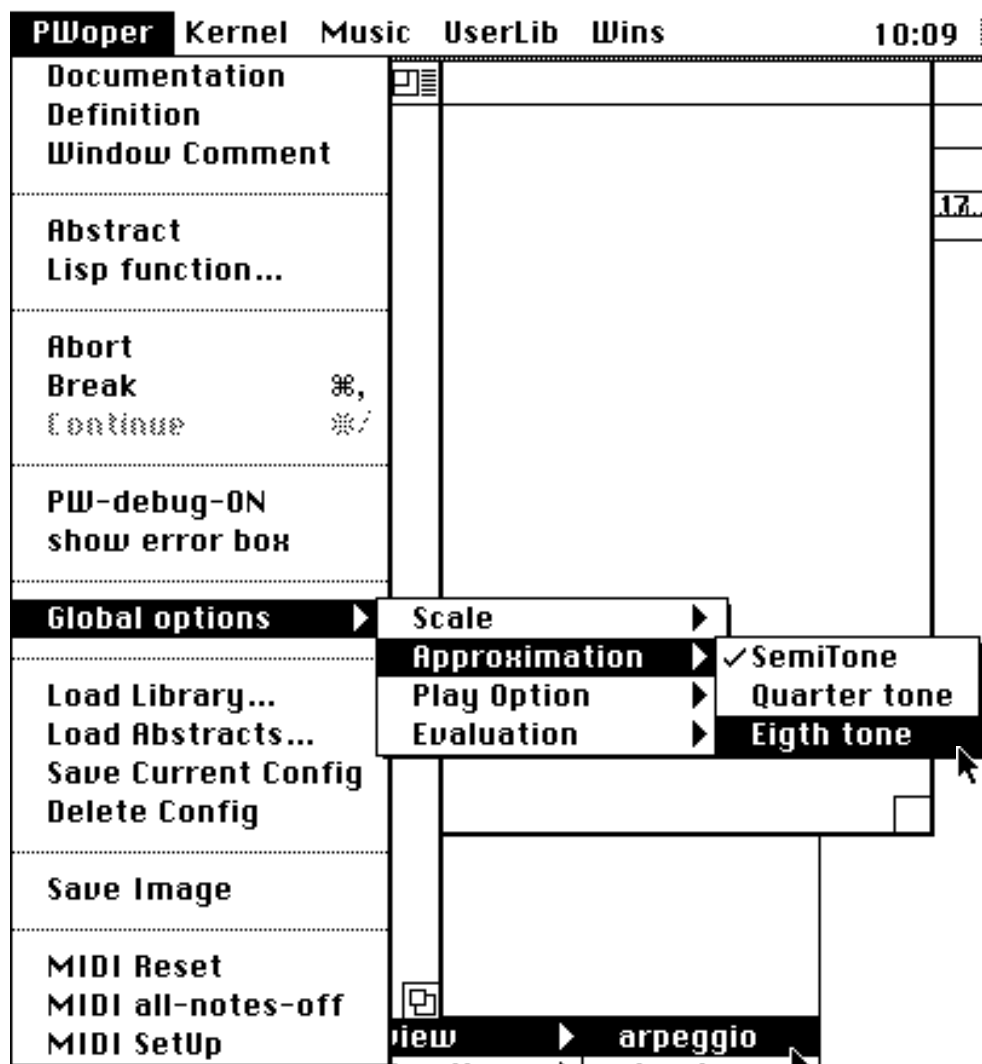
Utilisation

- A: Open the chord box (A) by clicking in the centre of the box. Choose the "arpeggio" mode to view the chord as a sequence of notes ready for editing.



To make one note appear in the chord box, «option-click» where you want the note to be written.

- B: Enter the transposition value in the numbox. To transpose by a semitone you have to enter the value 100. A quarter-tone corresponds to a value of 50 and a eighth of a tone to 25. These values are expressed in midicents. If you enter a positive number, the original sequence is transposed up, otherwise it is transposed down. To see the notation in quarter- or eighth- tone notation you have to choose the Global-option Approximation in the PWoper menu, as shown in the next figure.



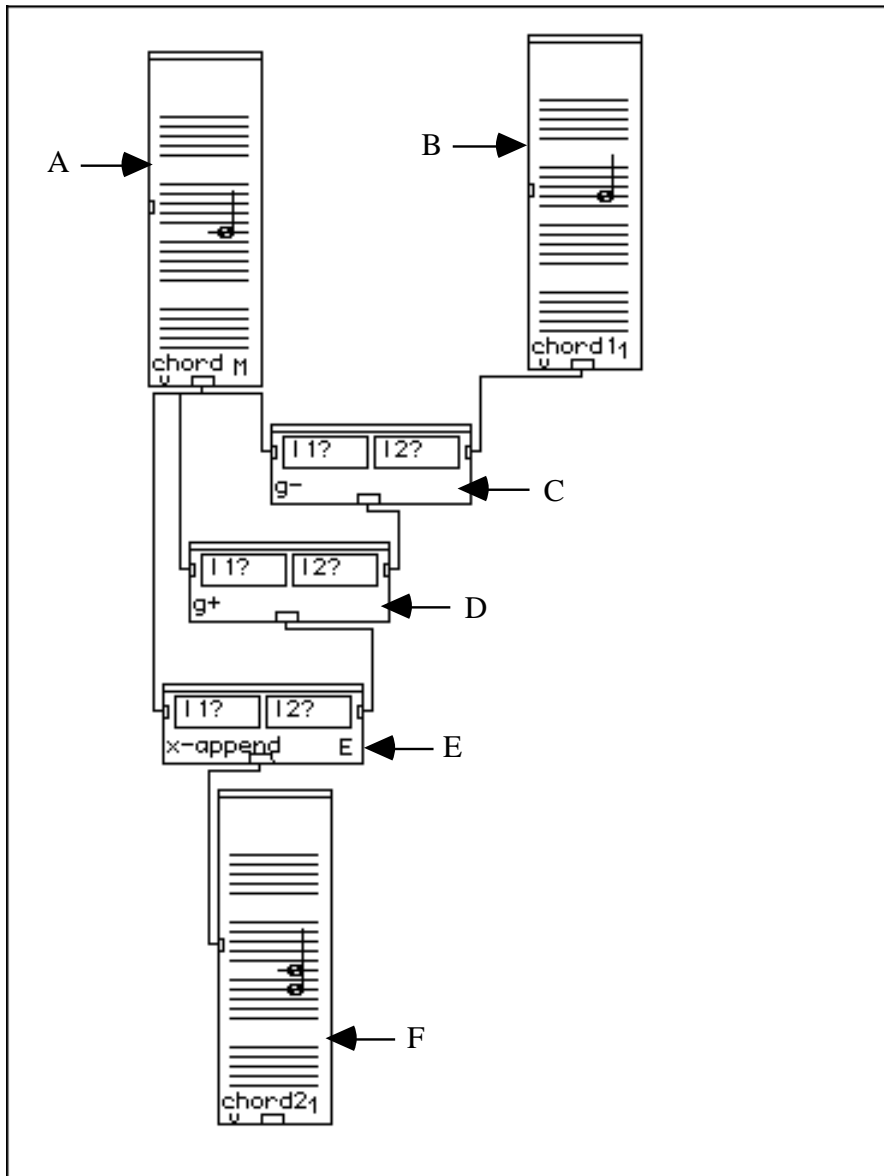
C: The addition box, g+, transposes the notes by adding the transposition value (in the numbox) to every note in chord box.

D: Evaluate the chord box and see the result by opening the box.

See also in your Reference manual: chord, g+ and numbox.

Tutorial 2

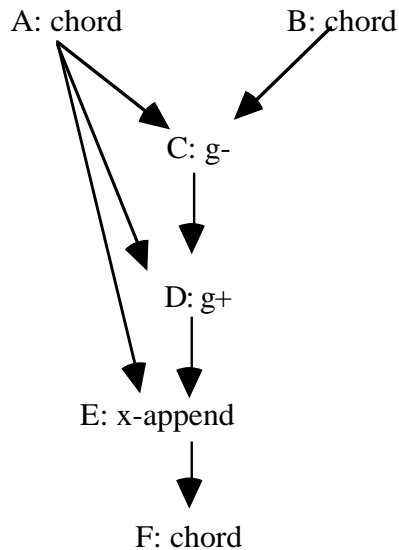
Basic operation: inversion of an interval.



Description

This patch returns the inversion of an interval in the chord box (A). That is to say that if you put the interval C3-G3 (an interval of a fifth) it will return the interval F2-C3.

Patch structure



Boxes used

chord, g+, g-, x-append.

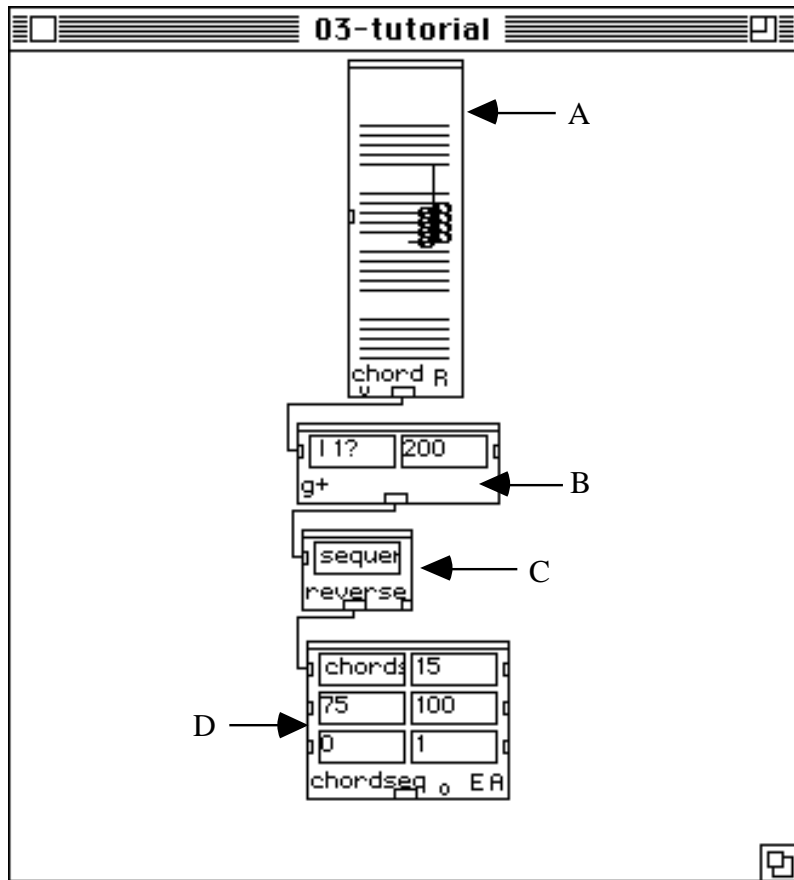
Utilisation

- A: Open the chord box (A) by clicking in the center of the box. Edit the lower note of your interval. If, for instance, you want a fifth C-G, the lower note is C.
- B: Do the same thing in the chord box (B), this time editing the upper note of your interval.
- C: The box g- calculates the difference between the note in the first chord box (A) and the note in the second (B).
- D: The box g+ calculates the sum of the note in the first chord box and the result of (C). In other words it transposes the note in the first chord box by the interval between the two notes.
- E: x-append groups both notes into a diad.
- F: Evaluate this chord box and see the result. It will return the inverted interval.

See also in your Reference manual: g- and x-append.

Tutorial 3

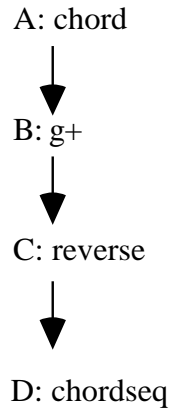
Basic operation: retrograde transposition of notes.



Description

This example first transposes the notes entered in chord (A) by adding the value put in (B) to each note, then it takes the retrograde of the note sequence.

Patch structure



Boxes used

chord, chordseq, g+, reverse.

Utilisation

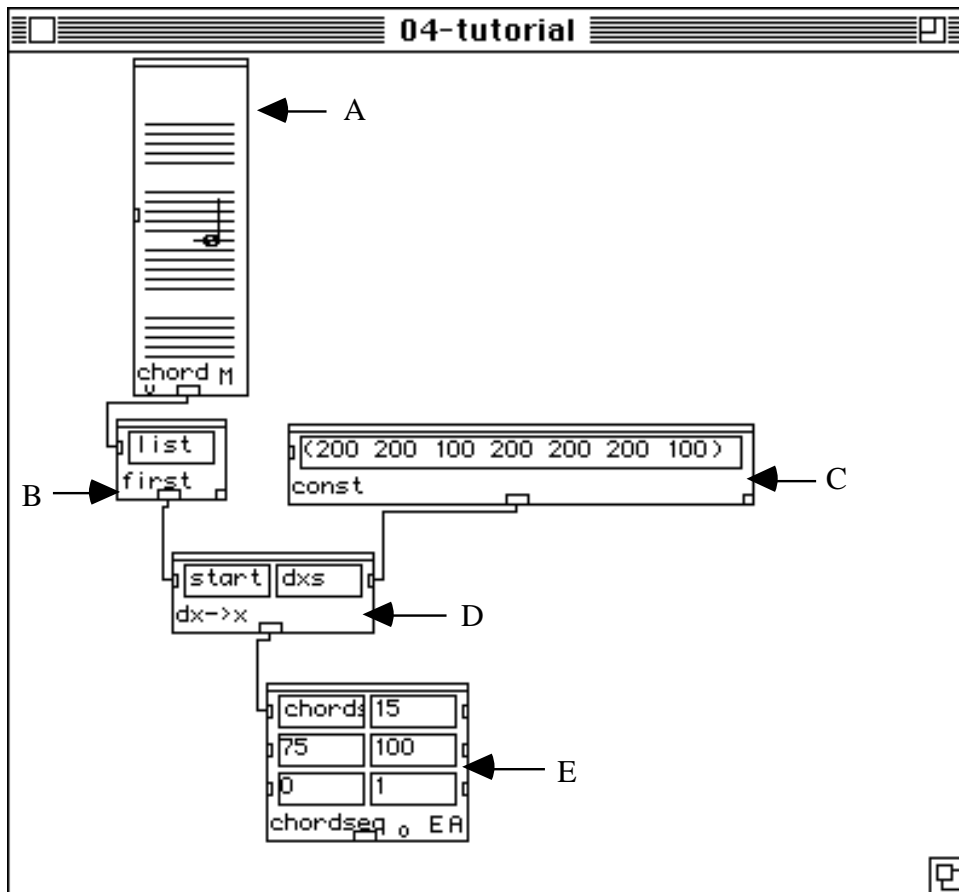
- A: Edit your own sequence of notes in the "arpeggio" mode¹.
- B: Select the interval by which you would like to transpose the notes. Remember that if you enter a positive number, the sequence is transposed up, otherwise it is transposed down.
- C: The reverse box reverses the order of the transposed note sequence (as it would be read left-to-right) thereby giving us the retrograde of the sequence.
- D: Evaluate the chordseq box and see the result.

See also in your Reference manual: chordseq and reverse.

1. See the tutorial number 1.

Tutorial 4

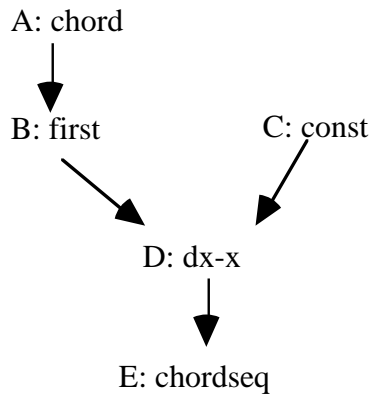
Basic operation: the construction of a scale.



Description

This patch creates a scale starting from the base note entered in the chord box (A) based on a series of intervals entered in the const box (C).

Patch structure



Boxes used

chord, chordseq, const, dx-x, first.

Utilisation

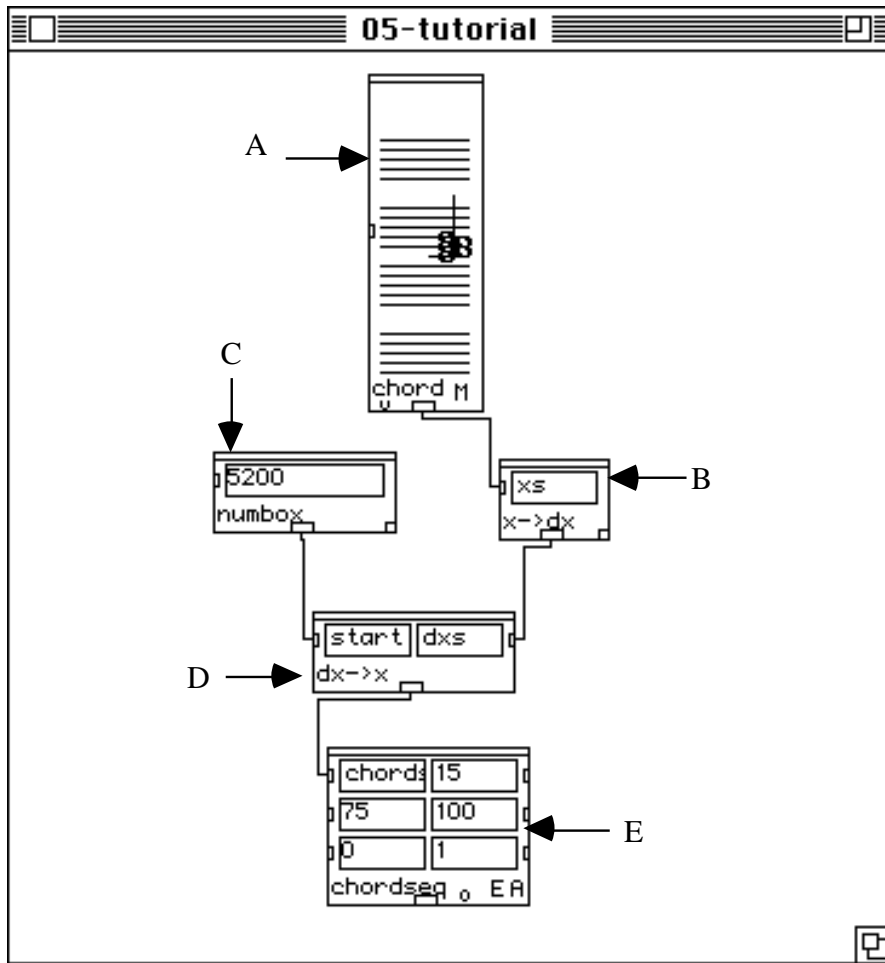
- A: Open the chord box and edit the note with which you would like your scale to begin.
- B: The first box takes the first element of the list coming out of chord box (A). The chord box always outputs a list even if it only has a single element, such as the note C3: (6000). By taking the first element of a one-element list such as this, the list is transformed into an atom¹. In contrast to the list which appears as (6000), the atom appears without parentheses: 6000. We have to do this operation because the module dx->x, to which first box is connected, accepts either an integer or a floating-point number — not a list.
- C: Enter a list of intervals in the const box in order to construct a musical scale. Remember that an interval of 100 is equal to a semitone.
- D: The dx->x module makes a scale, starting with the value coming from first box, constructed on the intervals entered in const.
- E: Evaluate the chordseq box to see the result.

See also in your Reference manual: const, dx-x and first.

1. See chapter Structure of data in this manual.

Tutorial 5

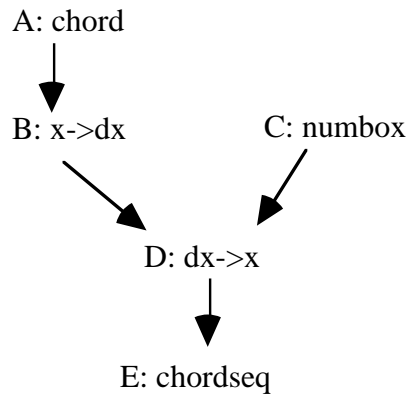
Basic operation: transposition based on interval analysis.



Description

This example analyses the intervals (B) in a sequence of notes (A), then, by using a starting value entered in (C), it re-creates the sequence, albeit transposed (B).

Patch structure



Boxes used

chord, chordseq, dx-x, numbox, x-dx.

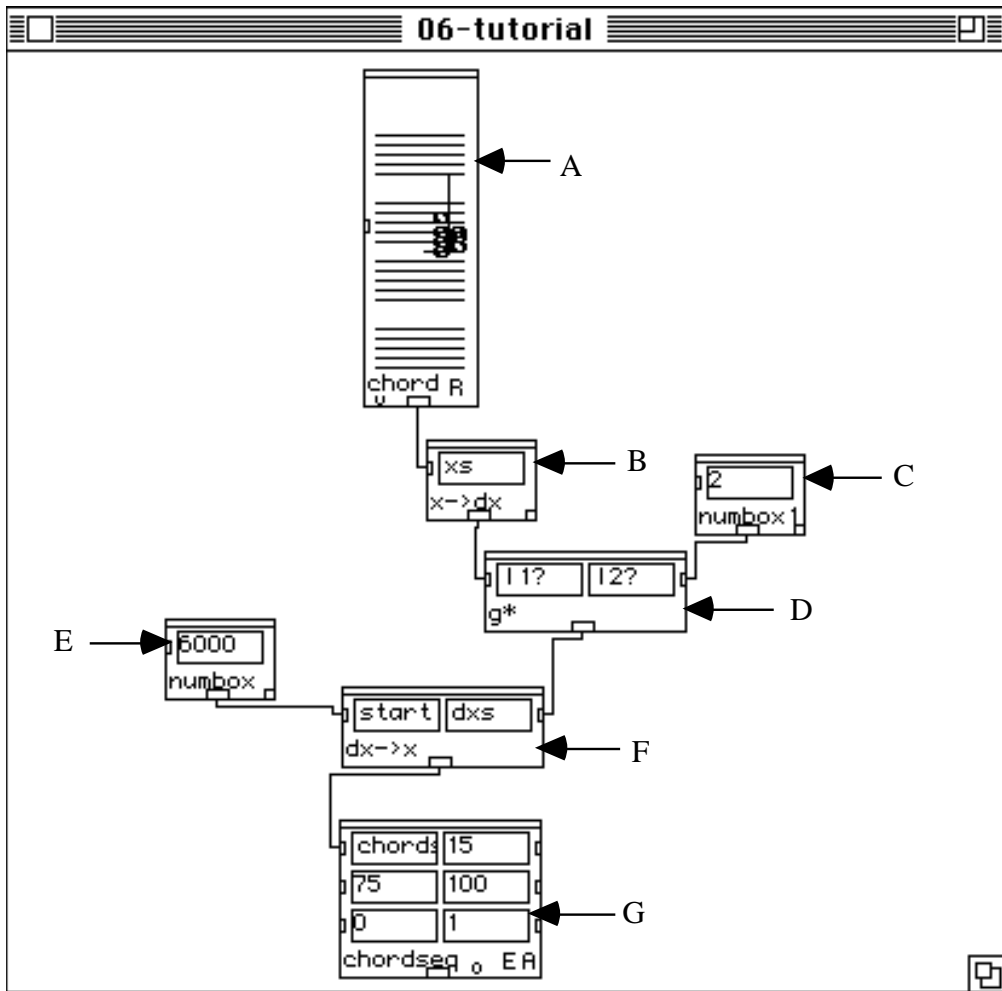
Utilisation

- A: Open this box and define a scale or note sequence to be transposed.
- B: The x->dx box calculates the differences between each pair of notes in the scale defined in (A). If you have the sequence DO-RE-MI-FA-SOL, the difference between DO and RE will be 200 (two semitones), between RE and MI, 200 (again, two semitones), and between MI and FA, 100 (one semitone)..... By doing this you calculate the intervals between the notes of your sequence.
- C: The numbox sets the base note for the transposed sequence or scale.
- D: The transposition is made as before, by constructing a scale based on the intervals derived from x->dx.
- E: Evaluate the chordseq and see the result by opening chordseq box.

See also in your Reference manual: x-dx.

Tutorial 6

Basic operations: expansion or compression of a sequence of notes.

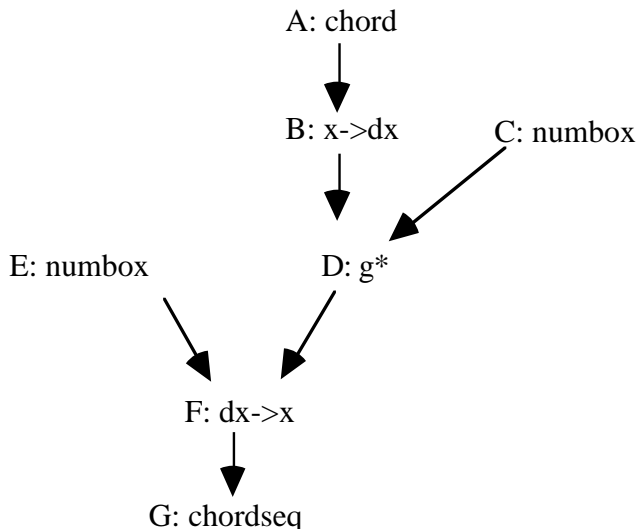


Description

This patch makes either an expansion or a compression of the intervals in a series of notes. The box $x \rightarrow dx$ (B) calculates the intervals of the sequence of notes (A), as in the previous example. Then g^* (D) changes these intervals proportionally by multiplying

them by a constant in the numbox (C). The result of the multiplication is a new list of intervals that, once entered in the dx->x box (F) can be used to create a new series of notes.

Patch structure



Boxes used

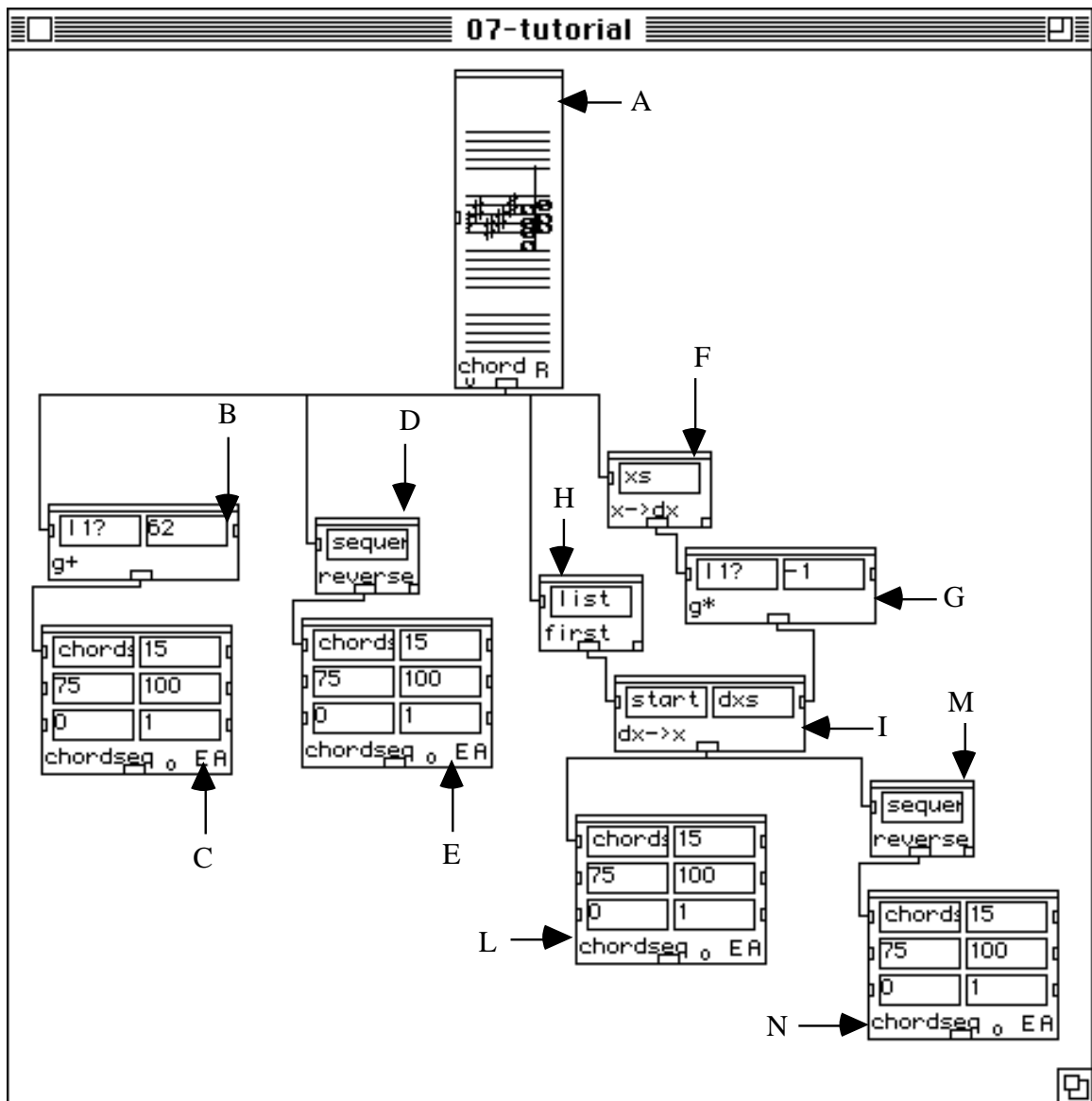
chord, chordseq, dx->x, g*, numbox, x->dx.

Utilisation

- A: Open the chord box by clicking in the centre of the box and edit a sequence of notes in the "arpeggio" mode.
- B: As before, the x->dx box calculates the intervals between the notes in the sequence (A).
- C: Enter a multiplication factor for the expansion or compression of the intervals in this numbox. If you enter a number between 0 and 1 (for example 0.2 0.365 0.99999 etc...) the note series will be compressed. On the contrary, if you enter a number greater than 1 (for example 2, 2.00001, 10 etc...) the series will be expanded. (A value of 1 will neither compress nor expand the series.)
- D: The g* box multiplies the intervals coming from the x->dx box by the value entered in numbox (C).
- E: In this numbox (E) you may enter the starting note value for the compressed or expanded series as in the previous examples.
- F: The dx->x module makes the new sequence, starting with the value coming from numbox (E), and constructed using the intervals deriving from the x->dx box.
- G: Evaluate and open the chordseq to see the result.

Tutorial 7

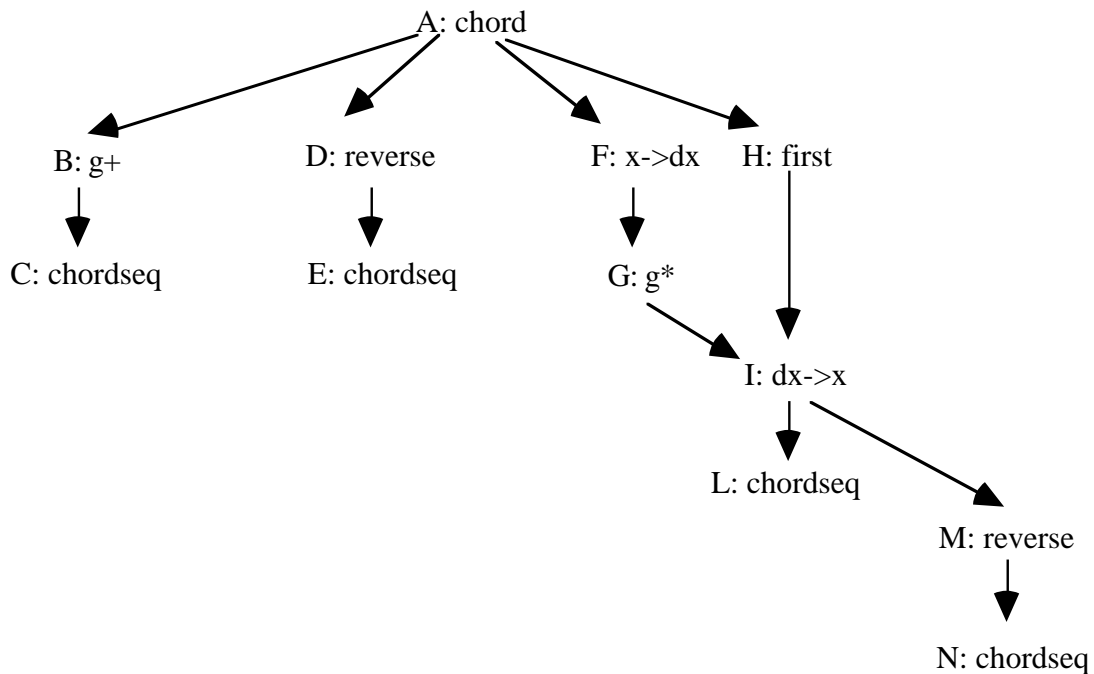
Four operations on a twelve-tone row.



Description

This patch is subdivided into four operations on the same sequence of notes (A): transposition, retrograde, inversion and retrograde of the inversion.

Patch structure

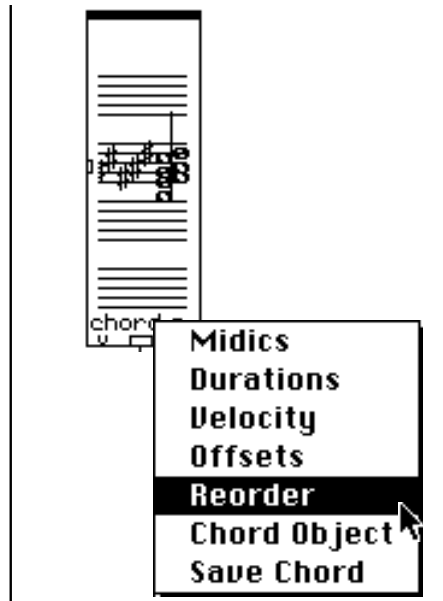


Boxes used

chord, chordseq, dx->x, first, g*, g+, reverse, x->dx.

Utilisation

A: Edit your series of notes in the "arpeggio" mode, close the box and choose the "reorder" mode in the chord menu.



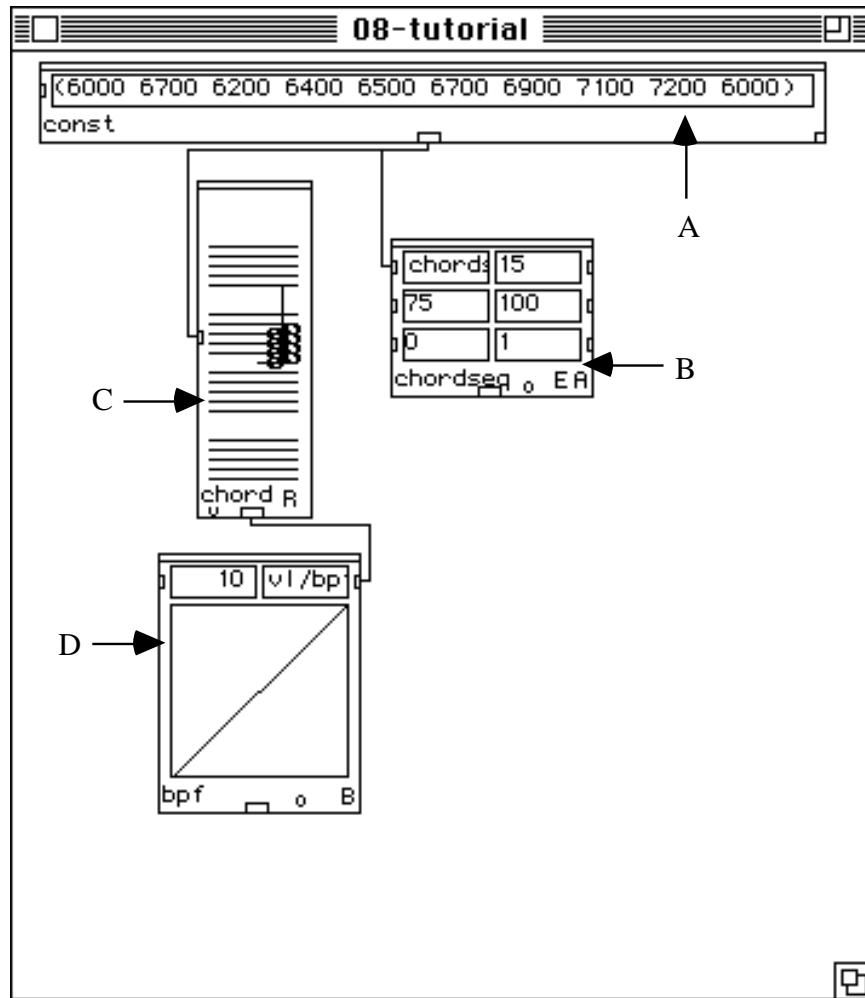
If you use the "Midics" mode (the default mode), the notes are read and sent out from the smallest to the largest. If you use the "Reorder" mode the sequence is read and sent out in the order in which they were entered. (don't ask...)

- B: The g+ box transposes the sequence¹.
- C: Evaluate and open this chordseq to see the result of the transposition.
- D: The reverse box gives us the retrograde of the original sequence, that is to say that the notes are read from right to left.
- E: Evaluate and open this chordseq to see the result of the reverse box's operation.
- F: The x->dx calculates the intervals in the original sequence of notes.
- G: The module g* multiplies each interval coming out of x-dx by -1. In doing this, it changes the sign of each interval; if we multiply 100 by -1 the result will be -100 and if we multiply -200 by -1 the result will be 200.
- H: The module first takes the first note of the original series (outputting it as an atom, not as a list of one element).
- I: The dx->x module creates a new sequence, starting from the first note of the original sequence (given by the first box), constructed on the intervals deriving from the g* box.
- L: Evaluate and open this chordseq to see the result.
- M: The reverse box makes the retrograde of the sequence coming out of dx->x box.
- N: Evaluate and open this chordseq to see the result of the last operation.

1. See tutorial number 1.

Tutorial 8

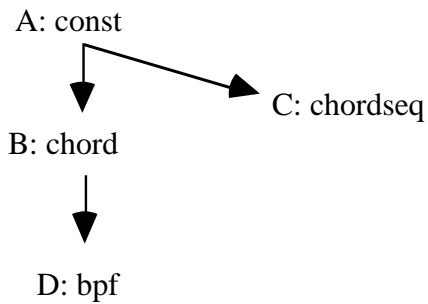
Using break-point function (BPF): the graphic representation of a sequence of notes.



Description

In this patch you can enter a series of values in the const box (A), which are first converted into musical notation and then into a graphic representation.

Patch structure



Boxes used

chord, chordseq, bpf, const.

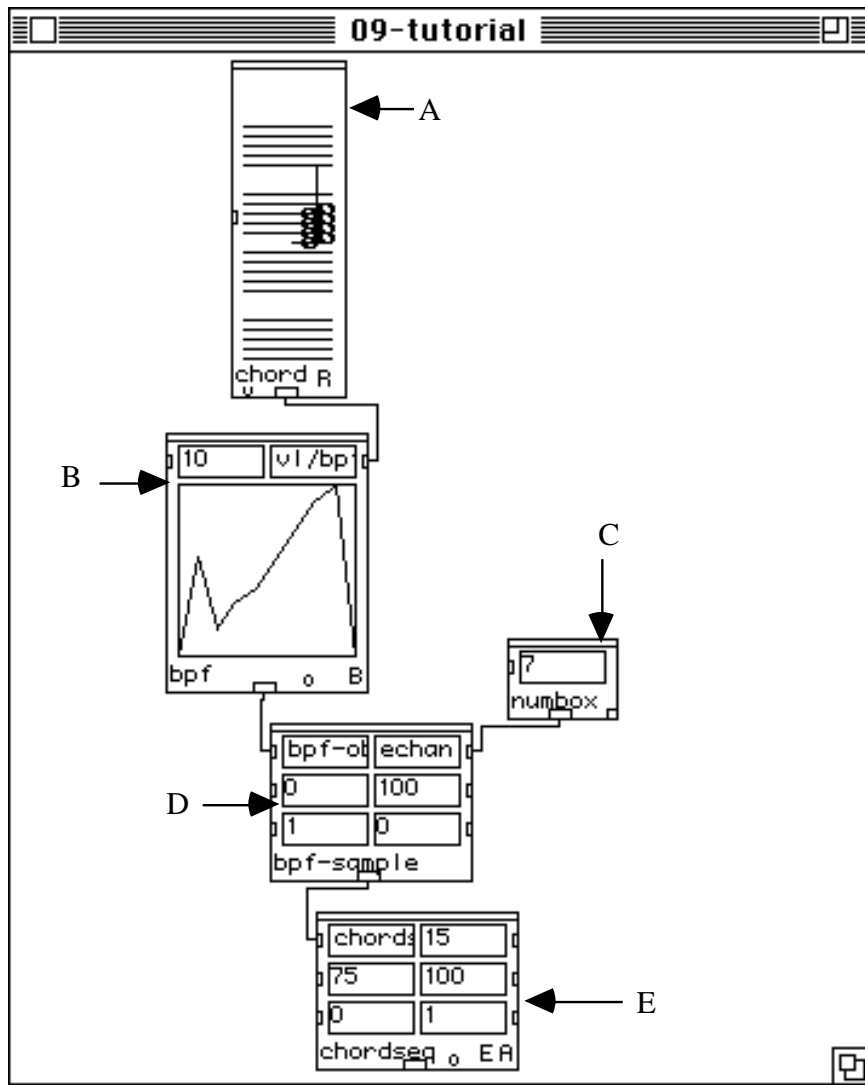
Utilisation

- A: Enter some values in midicents in this const box. Remember that in midicent-notation C3 is equal to 6000, C#3 is equal to 6100, and so forth.
- B: The chordseq box receives these midicent-values and translates them into musical notation. Evaluate this box to see the musical result.
- C: The chord box can visualise the same result either as a chord or as a sequence ("arpeggio" mode). Evaluate the box and see the different types of musical notation.
- D: The bpf box translates the midicents into a break-point function. Evaluate the bpf box to see the graphic representation of your sequence of notes.

See also in your Reference manual: bpf.

Tutorial 9

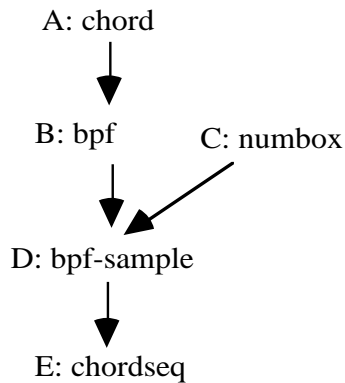
Using break-point function (BPF): sampling a sequence of notes.



Description

This patch takes the notes from the chord box (A), converts them into a break-point function (B) and re-samples (D) the break-point function using a number or samples defined in numbox (C).

Patch structure



Boxes used

chord, chordseq, bpf, bpf-sample, numbox.

Utilisation

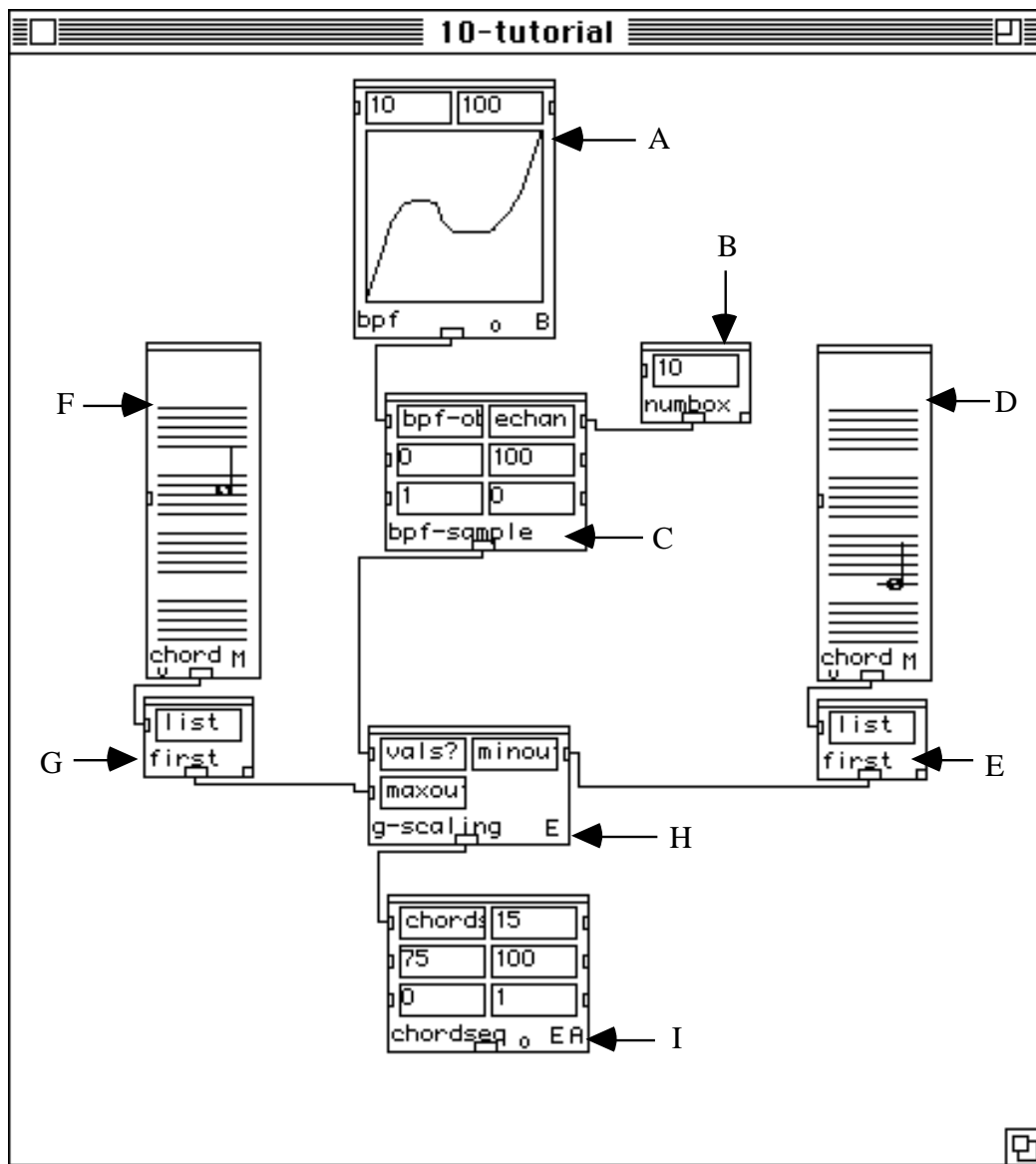
- A: Enter a sequence of notes and select the "Reorder" mode in the chord menu¹.
- B: The bpf box converts the midicents coming out of chord box (A) into a break-point function.
- C: In this numbox , you may enter the number of samples by which you would like to re-sample your curve.
- D: The module bpf-sample can re-sample a curve using any number of samples. In this patch, you can define the number of samples in the numbox (C).
- E: Evaluate the chordseq and see the result. Try changing the number of samples in the numbox, re-evaluate, and look at the result in the chordseq.

See also in your Reference manual: bpf-sample.

1. See tutorial 7 paragraph A.

Tutorial 10

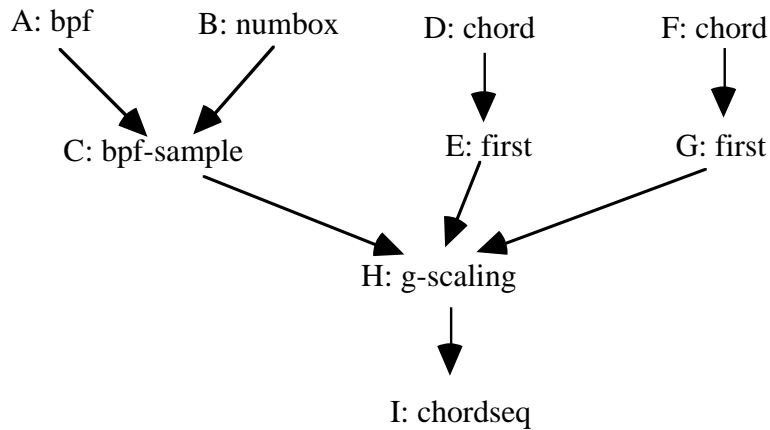
Using break-point function: transformation of a break-point function into a sequence of notes.



Description

In this patch you can design a curve which can then be converted into a melodic profile and scaled to different ranges.

Patch structure



Boxes used

chord, chordseq, bpf, bpf-sample, first, g-scaling, numbox.

Utilisation

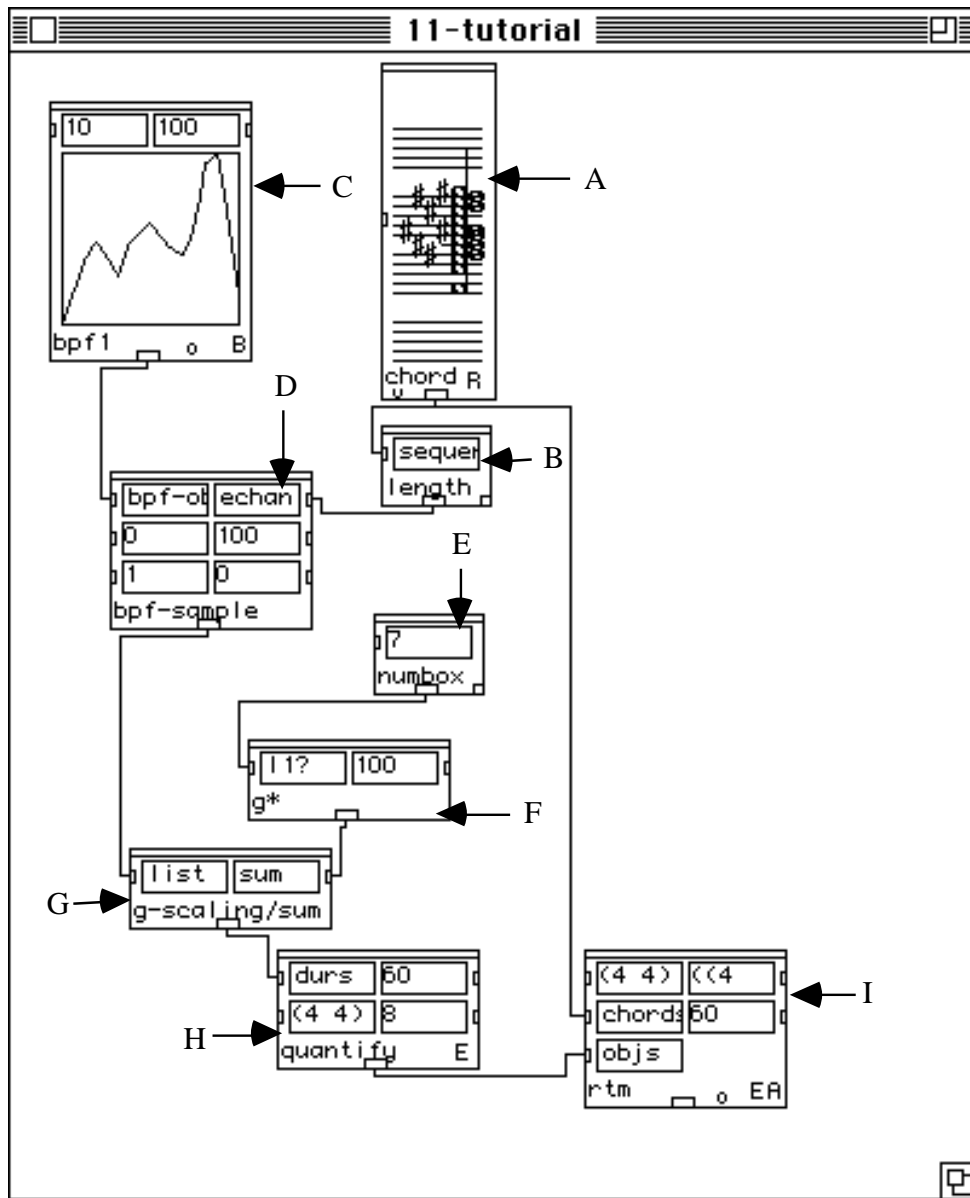
- A: Open the bpf module and edit a curve by clicking directly in the rectangle of the bpf.
- B: In this numbox, enter the number of samples by which you want your curve be sampled.
- C: bpf-sample samples your curve in the bpf box (A) using the number of samples entered in numbox (B).
- D: Open the chord box on the left (D) and enter one note. The note you enter, will be the lower note of your melodic profile.
- E: The first box takes the first element of the list coming out of chord box (D)¹.
- F: Open the chord box on the right (F) and enter one note. The note you enter will be the upper note of your melodic profile.
- G: The first box takes the first element of the list coming out of chord box (F)
- H: The module g-scaling scales the values coming out of bpf-sample between the value entered in the chord box on the left (D) and the value entered in the chord box on the right (F).
- I: Evaluate the chordseq box and see the result.

See also in your Reference manual: g-scaling.

1. See also the tutorial number 4 at paragraph B.

Tutorial 11

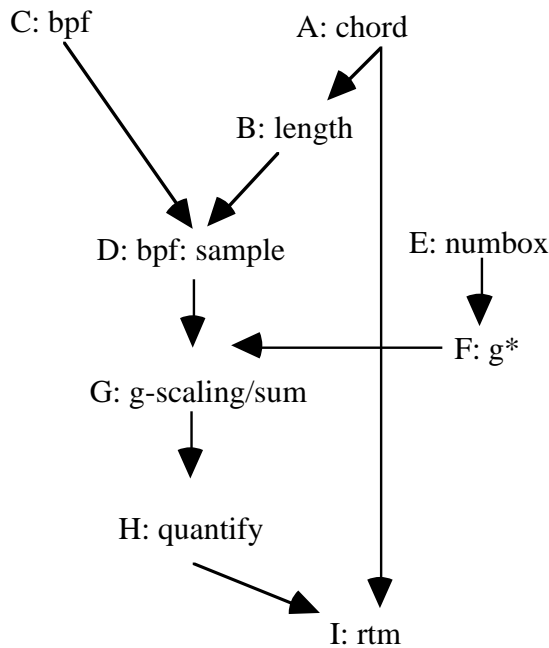
Using break-point function: transformation of a curve into a sequence of durations.



Description

This patch transforms a break-point function into a list of durations. The length of the list of durations is taken from the length of the list of notes entered in the chord box.

Patch structure

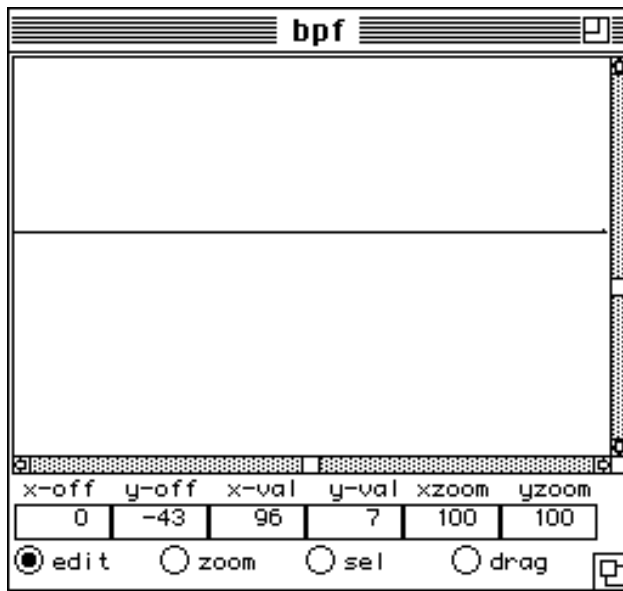


Boxes used

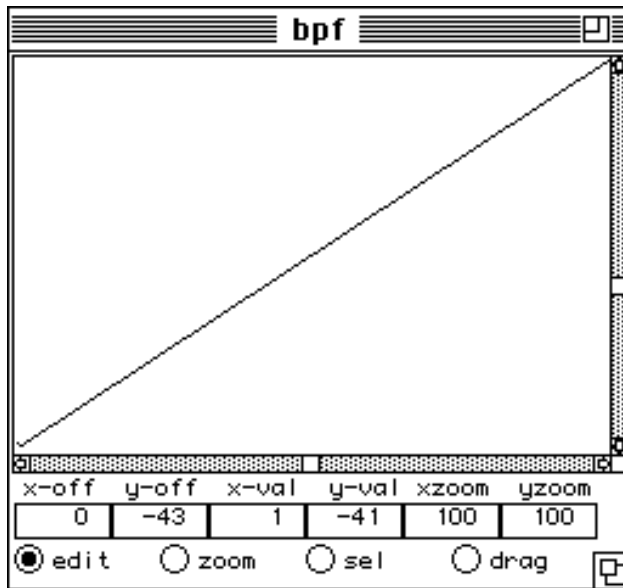
chord, chordseq, bpf, bpf-sample, g*, g-scaling/sum, length, numbox, quantify, rtm.

Utilisation

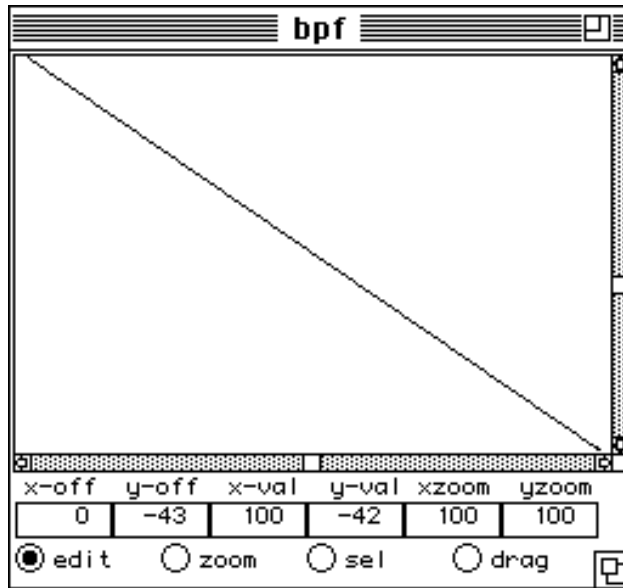
- A: Open the chord box and edit your sequence of notes. Then select the "Reorder" mode as we have already done in tutorial number 7, paragraph A.
- B: The module length calculates the length of an incoming list. In this example, length calculates the number of elements belonging to the sequence of notes you have entered in the chord box (A).
- C: Open and edit the bpf module. The curve you enter here will be transformed into a rhythmic sequence. Remember that if you want to have a constant rhythm you must enter a horizontal line, as shown in next figure.



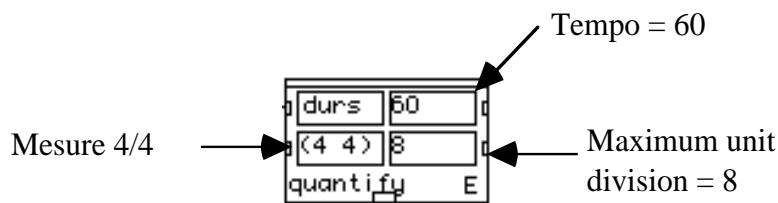
If you want a decelerating rhythm you have to create an upward-sloping curve like the following one:



Contrarily, if you want a accelerating rhythmic sequence you have to create an downward-sloping curve:



- D: The bpf-sample samples the curve you have entered in bpf. The number of samples corresponds to the length of the list of elements entered in the chord box. That is to say that the list coming out of bpf-sample will be the same as the list in the chord module.
- E: Use the numbox to define how many seconds long your rhythmic sequence will be. A value of 5 means that your sequence will be five seconds long (obviously).
- F: The module g^* multiplies the number entered in the numbox (D) by 100 (which has been manually entered in the box's right input).
- G: The module g-scaling/sum scales the elements in order to make their sum equal to the number coming out of the g^* module. In this example, as the result of g^* is 700. This means that the sum of all of the elements in the bpf-sample will be equal to 700.
- H: The module quantify quantizes a list of values into a given metric measure (such as 4/4), with a given tempo (quarter-note = 60) and with a maximum beat division (8 divisions per quarter-note).



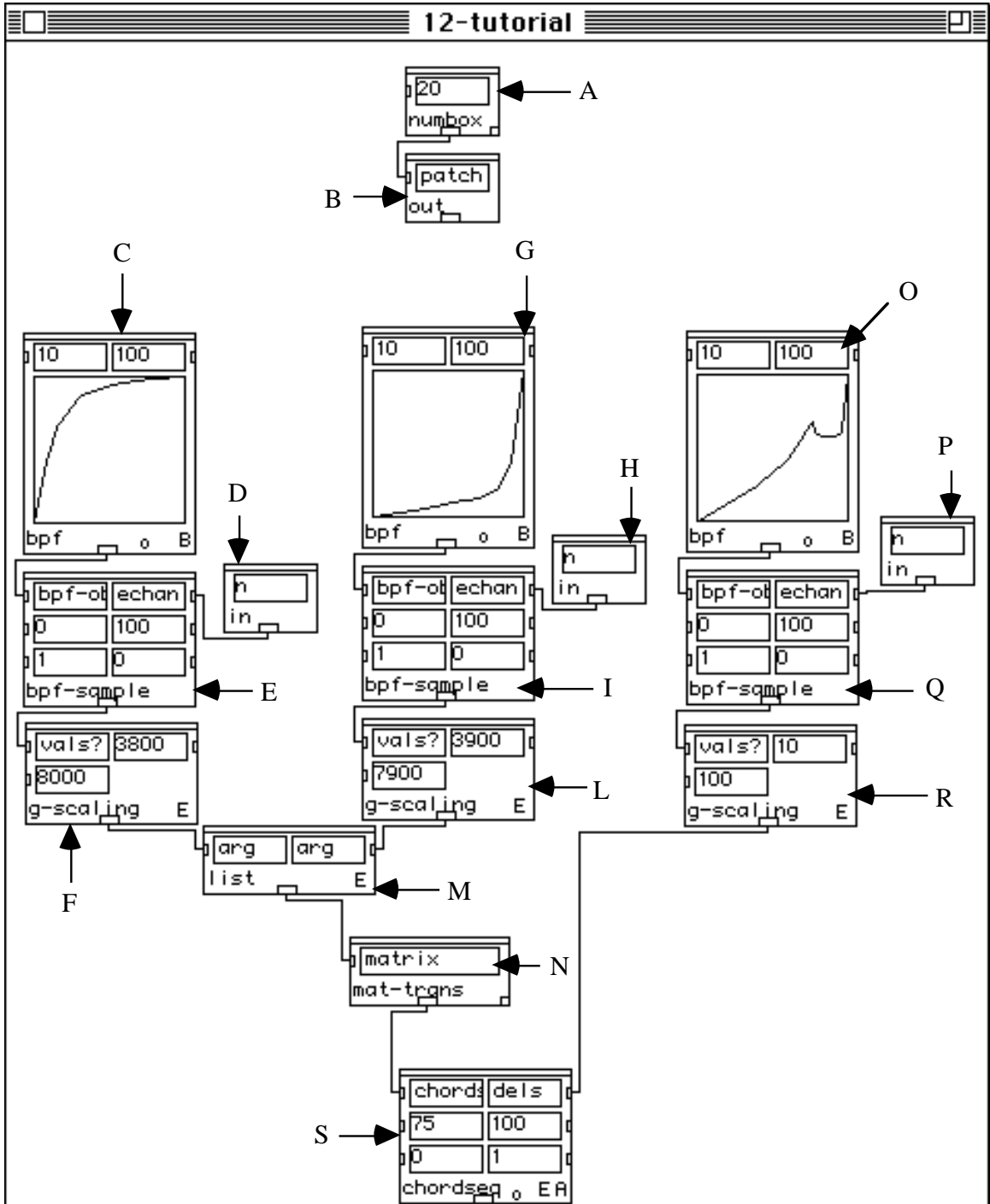
- I: rtm is a module for rhythmic notation. Evaluate this box, open it and see the result given by all of the patch modules. Try changing the value in numbox (E) and re-evaluating the rtm box to see the differences.

See also in the reference manual: g-scaling/sum, length, numbox, quantify¹, rtm.

1. We strongly suggest you deeply study the two modules quantify and rtm in the PatchWork reference.

Tutorial 12

Transformation of curves into sequences of notes and durations.

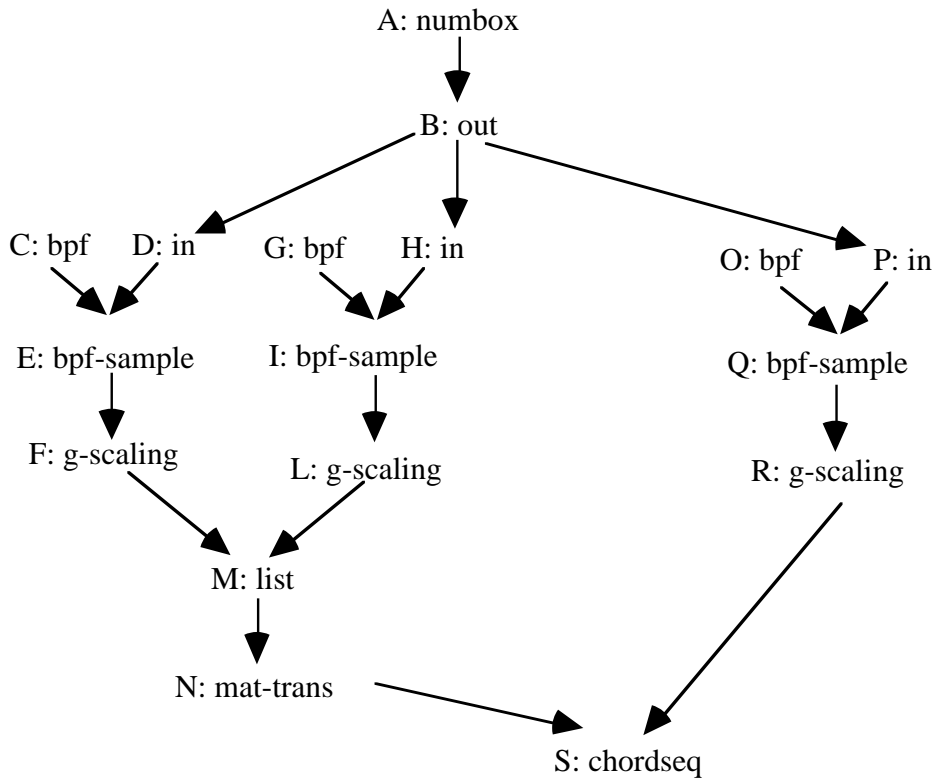


Description

This patch converts two break-point functions into two melodic profiles, then groups the two curves into pairs of values in order to create a sequence of diads. A third break-

point function generates a sequence of durations which is then converted into proportional musical notation.

Patch structure



Boxes used

chordseq, bpf, bpf-sample, g-scaling, in, list, mat-trans, numbox, out.

Utilisation

- A: This numbox specifies the length of the three bpf-sample modules. The number you edit it is transmitted through the box out (B) to the three bpf-sample modules.
- B: The out box is connected to all in boxes that share the same name. In contrast to all PatchWork modules, out and in work always together; the out box sends any kind of message to the corresponding in box. The connection between in and out is made automatically, for this reason you don't have to make an on-screen connection with cables.
- C: Open and edit the bpf box (C). The curve you make will be converted into a melodic profile.
- D: This module in directly receives the number sent by the out box (B), without necessitating any graphical connection.
- E: As we have already seen, the bpf-sample box samples the break-point function (C) using a pre-defined number of samples. In this patch the number of samples is given in the numbox (A)¹.

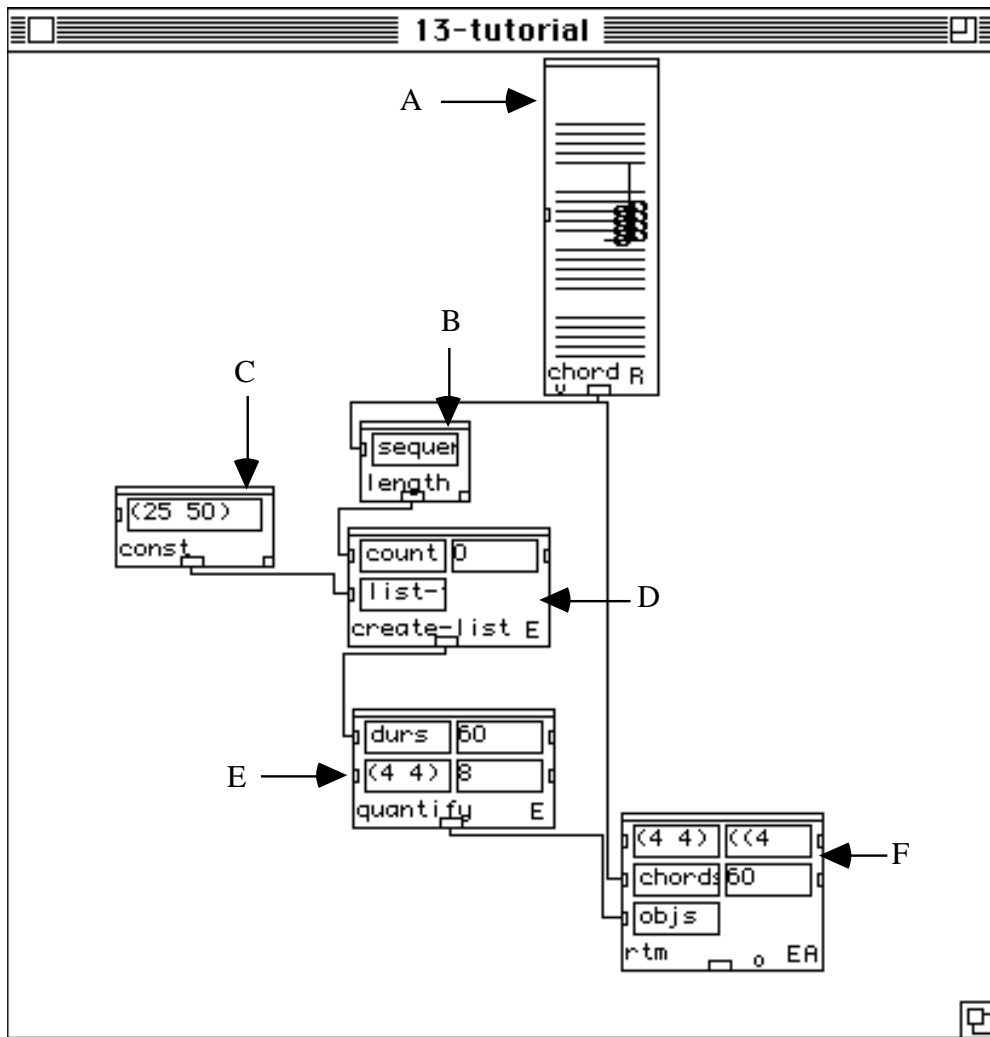
1. For bpf-sample see tutorial number 9.

- F: In this example, the module g-scaling scales the value coming out of bpf-sample between the values 3800 and 8000.
- G: Open and edit the bpf box (G). The curve you make will be converted in another melodic profile.
- H: This module in receives the number sent by out box (B), without necessitating any graphical connection.
- I: This bpf-sample box does the same thing as the previous bpf-sample (E). It samples the break-point function (G) in a number of samples defined in the numbox (A).
- L: The module g-scaling (L) scales the value coming out of bpf-sample (I) between the values 3900 and 7900.
- M: The module list makes a list with the results of both above-mentioned bpf-sample boxes (E, I). As each result is a list, the module list will make a list of two lists.
- N: The module mat-trans takes the output of the list box (M), and creates first a list with the first element of bpf-sample (E) and the first element of bpf-sample (I), then a second list with the second element of bpf-sample (E) and the second element of bpf-sample (I), then a third list with the third element of each, and so on.
- O: Open and edit the bpf box (O). The curve you make will be converted into a sequence of durations.
- P: This module in receives the number sent by out box (B), without necessitating any graphical connection.
- Q: The bpf-sample box samples the break-point function (O) in a specific number of samples defined in the numbox (A).
- R: The module g-scaling (F) scales the value coming out of bpf-sample between the values 10 and 100.
- S: The chordseq box (S) receives the pairs of notes coming out of the mat-trans module in its «chord» input and also receives the duration of each pair of notes in its «dels» input. Evaluate the chordseq box to see the result. Once you have done this, try changing either the curves in the bpf modules, the values of the g-scaling modules, or the value in the numbox (A) and re-evaluate the chordseq box to see the differences.

See also in your Reference manual: in, list, mat-trans, out.

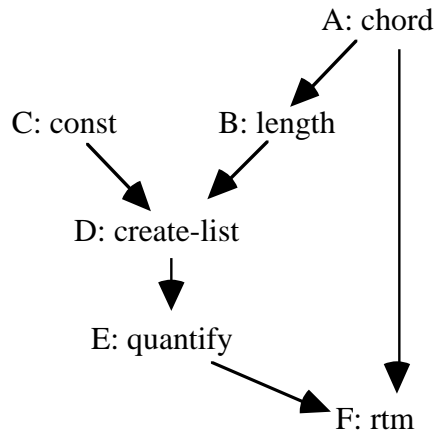
Tutorial 13

List operations: creating lists of the same length.



Description

This patch creates a list of durations (D) which has the same number of elements as the list of notes (A).



Boxes used

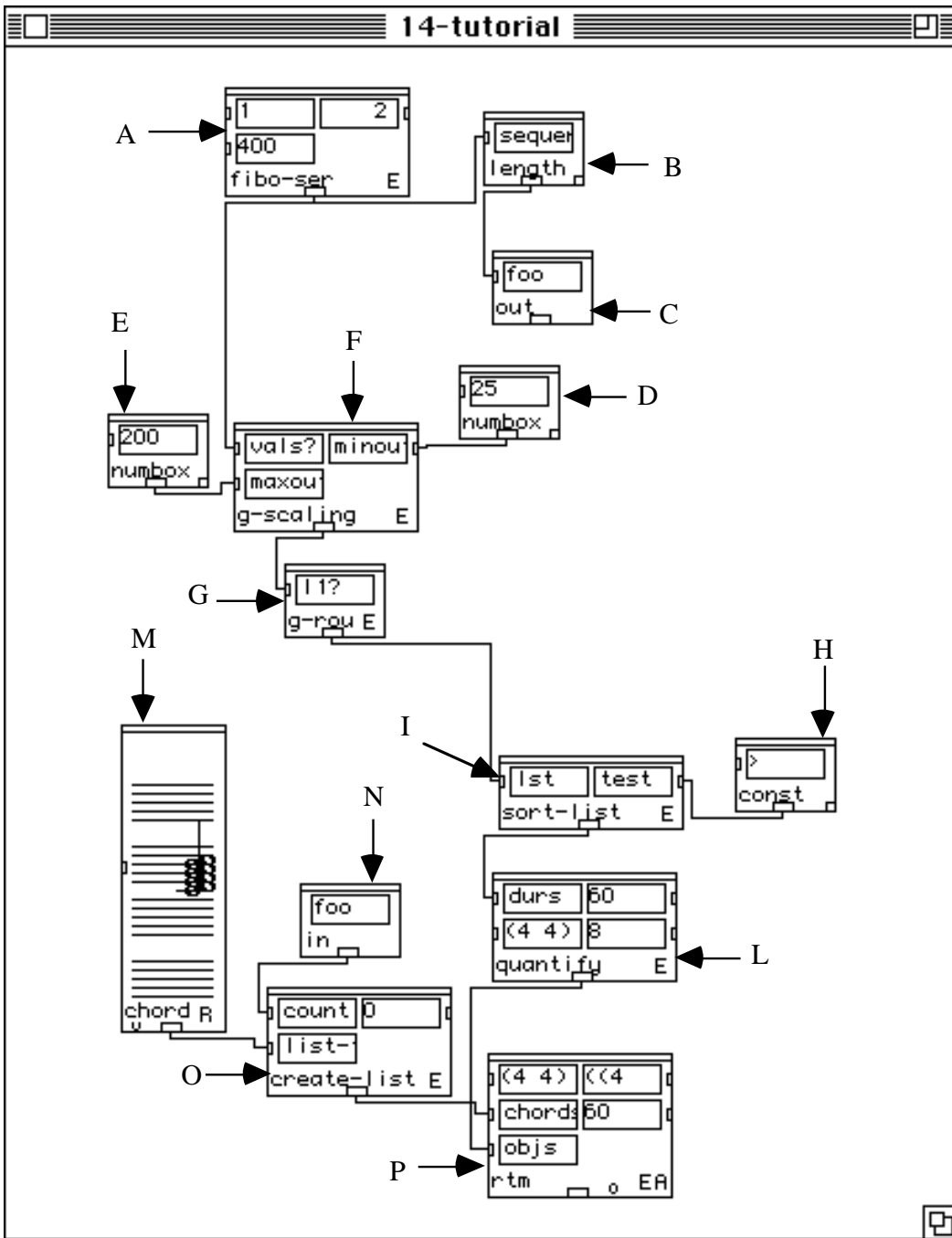
chord, const, create-list, length, quantify, rtm.

Utilisation

- A: Open the chord box (A) and edit a sequence of notes. Then select the "Reorder" mode in the chord box menu.
- B: The length box (B) counts how many elements you have entered in chord box (A).
- C: In the const box (C), enter a list of some elements that will be converted into durations. The inputs you enter in the const box will be eventually read by the module quantify. Remember that for the quantify box a value of 100 is equal to 1 second (with tempo = 60). Therefore a value of 50 is equal to a half a second, 25 to an eighth of a second, etc...
- D: The create-list box makes a list with the elements you have entered in the const box (C). The length of the created list is defined by the value coming out of length (B).
- E: In this example, the quantify box (E) quantizes the list of values coming out of create-list box (D), into measures of 4/4, with a given tempo of 60 to the quarter-note and with a maximum unit division of 8 (thirty-second-note resolution).
- F: Evaluate the rtm box (F), open it and see the result. Try changing the elements entered in the chord box (A) and in the create-list box (D). Re-evaluate the rtm box to see the new result.

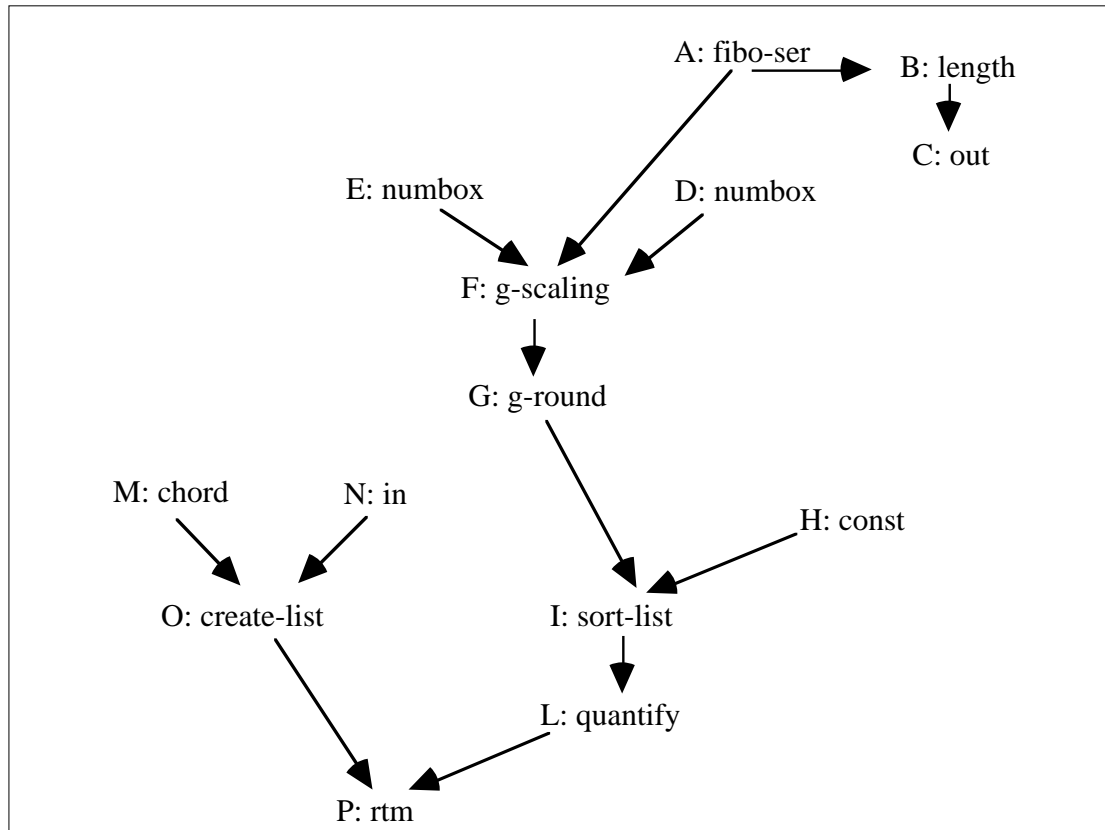
Tutorial 14

Lists operation: generation of lists of same length.



Description

This example creates a list of notes (Q) containing the same number of elements as a given list of durations (I) which are derived from a Fibonacci series (A).

**Boxes used**

chord, create-list, fibo-ser, g-round, g-scaling, in, length, numbox, out, sort-list, quantify, rtm.

Utilisation

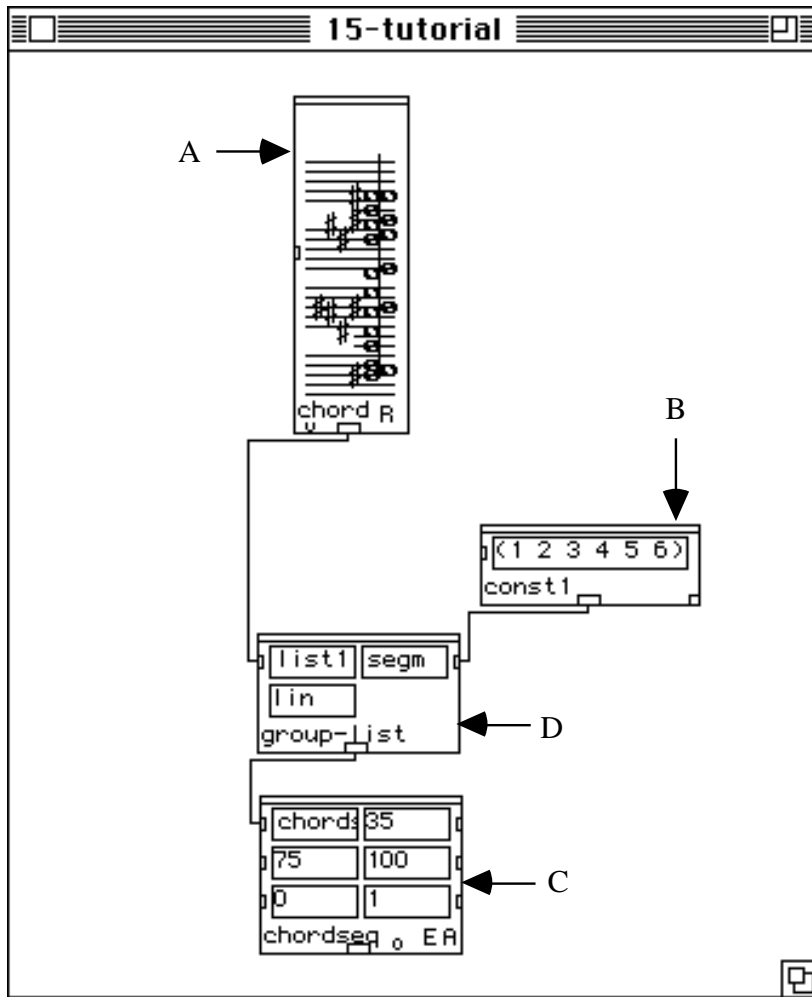
- A: The fibo-ser box (A) creates a Fibonacci series from value 1 to value 400.
- B: The length box (B), as we have already seen, counts the number of elements in the list coming out of fibo-ser box (A).
- C: The box out (C), as seen in tutorial 12, sends the value of the length box (B) to the corresponding in boxes.
- D: The value you enter in the numbox module on the right (D) will correspond to the shortest duration in your rhythmic sequence.
- E: The value you enter in the numbox module on the left (E) will correspond to the longest duration in your rhythmic sequence.
- F: The module g-scaling (F) scales the values coming out of the fibo-ser box (A) between the values entered in numbox (D) and numbox (E).
- G: In this example, the module g-round (G) converts the fractional numbers, coming out of the g-scaling box, into a list of integers.
- H: The module const (H) sends the symbol ">" to the sort-list box.
- I: The sort-list box (I) orders the elements coming out of g-scaling (F) from largest to smallest (defined by the symbol ">", above).

- L: The quantify box (E) quantizes the list of values coming out of sort-list (I), into measures of 4/4, with a tempo of 60, and with a maximum unit division of 8 (thirty-second-note resolution).
- M: Open and edit a sequence of notes in the chord box (using the "arpeggio" mode). Then close the box and set it to the "Reorder" mode.
- N: The module in receives the output sent by the module out. In this example the value received is the length of the Fibonacci series.
- O: The create-list box (O) makes a list with the elements you have entered in the chord box (M). The length of the list is defined by the value coming out of in box (N) — in this case it will be equal to the length of Fibonacci series.
- P: Evaluate the rtm box (P) and open it to see the result. Change either the values of the fibo-ser box, the values of the boxes numbox (D) and (E), or the symbol in the const box (H) (try using the symbol "< ") and look at the new results.

See also in your Reference manual: fibo-ser, g-round and sort-list

Tutorial 15

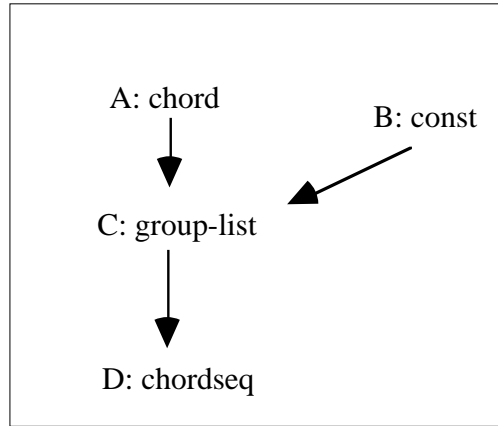
List operations: grouping lists into many sub-lists.



Description

This patch groups a list of notes (A) into a defined number of subgroups (C).

Patch structure



Boxes used

chord, chordseq, const, group-list.

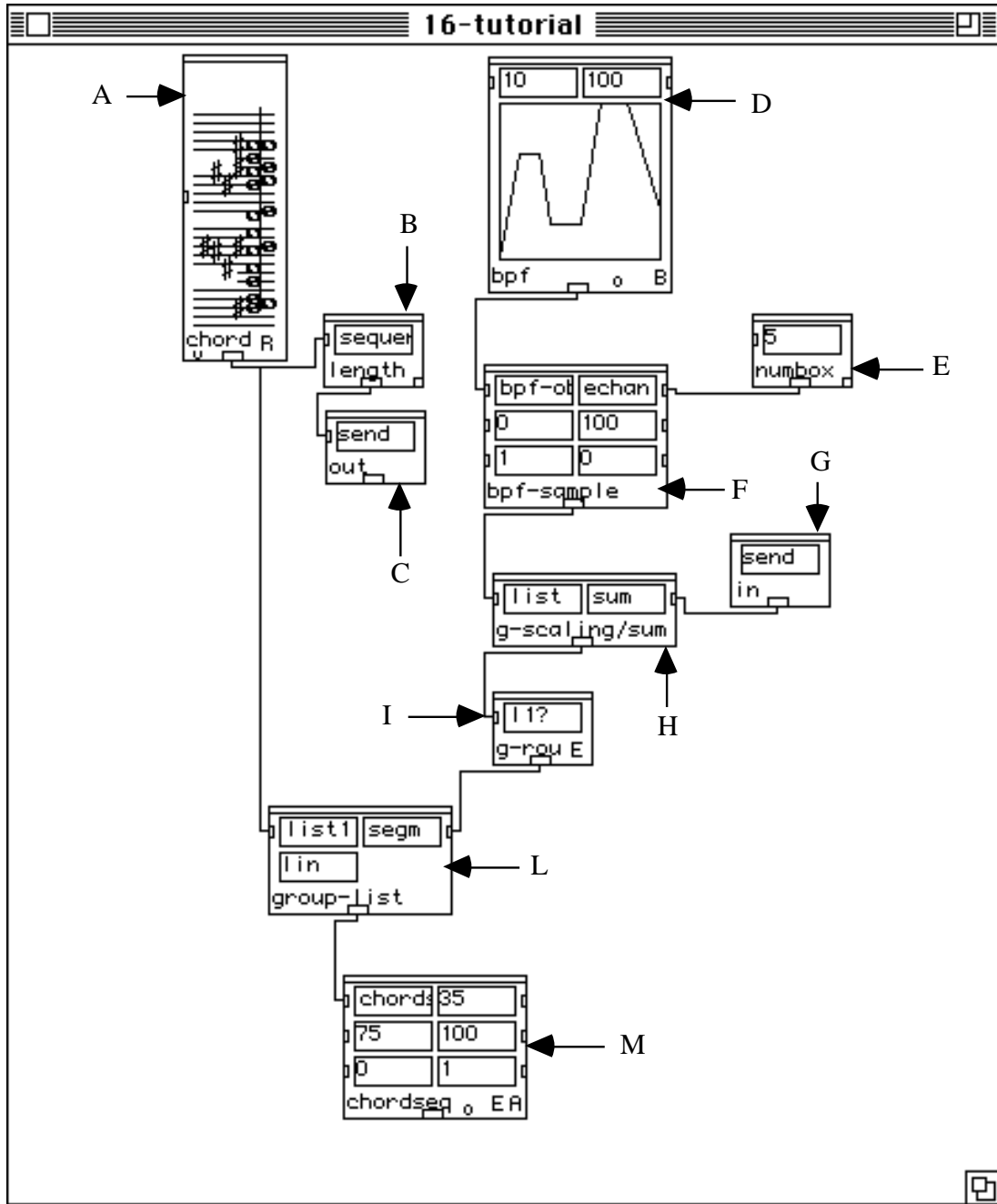
Utilisation

- A: Open the chord box and enter a list of notes using the "arpeggio" mode, then close the module and set it to the "Reorder" mode.
- B: Enter a list of numbers in the const box (B) whose elements will correspond to the length of each subgroup.
- C: The module group-list takes the notes coming out of the chord box (A) and groups them into the subgroups you have defined in the const box (B).
- D: Evaluate the chordseq box to see the result obtained. Now change the elements in the const box (B), evaluate the chordseq box once again, and look at the new result.

See also in your Reference manual: group-list.

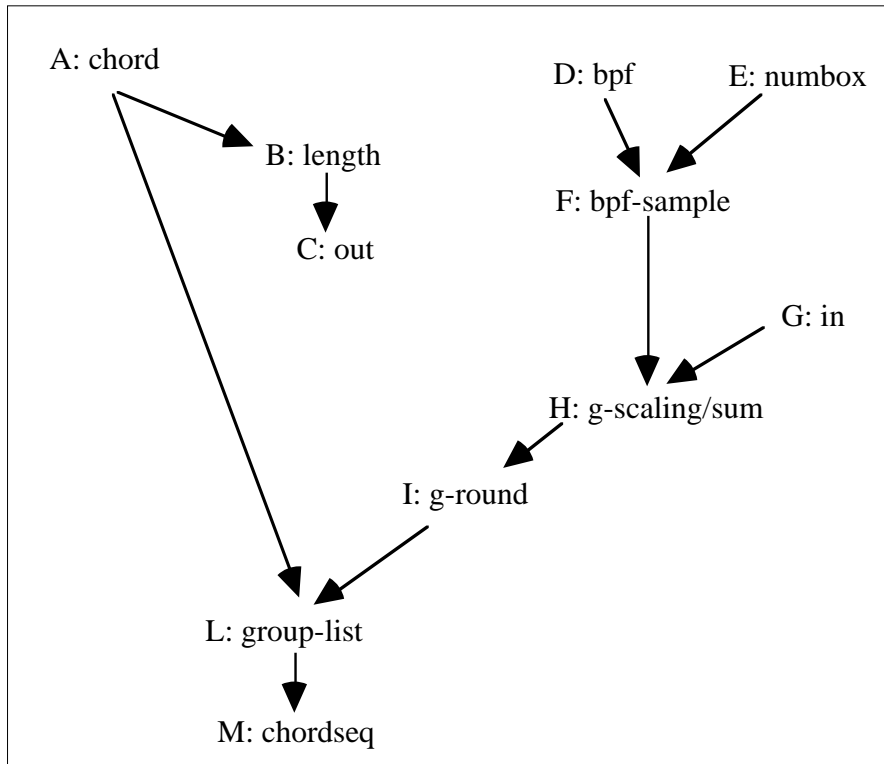
Tutorial 16

List operations: grouping list into many lists.



Description

This example groups a list of notes (A) into subgroups that are defined by a break-point function (D).



Boxes used

bpf, bpf-sample, chord, chordseq, group-list, g-round, g-scaling-sum, in, length, numbox, out.

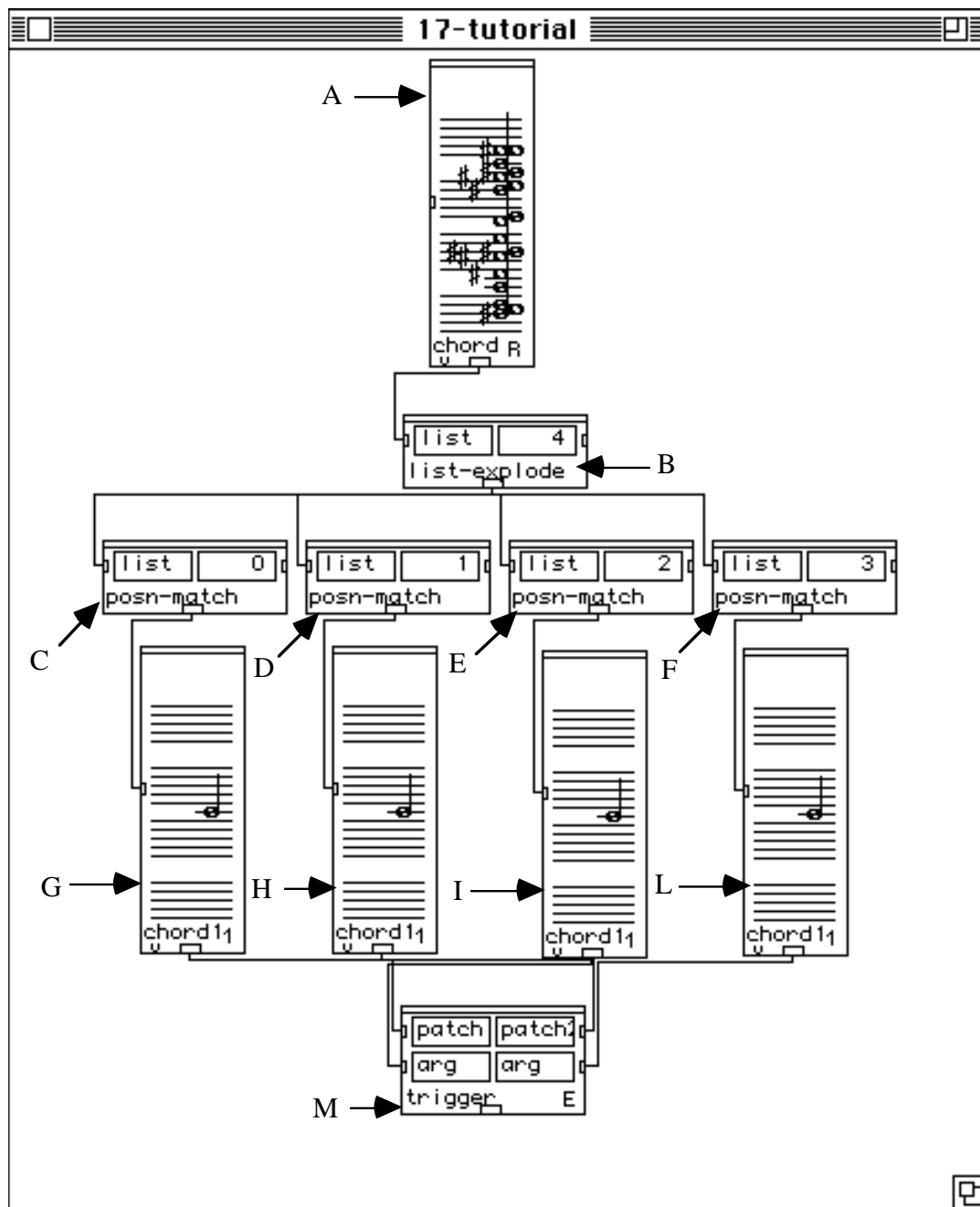
Utilisation

- A: Open the chord box (A), enter a list of notes in the "arpeggio" mode, close the module and set it to the "Reorder" mode.
- B: The length box (B) counts how many notes you have entered in the chord box (A).
- C: The module out sends the value coming out of the length box (B) to the module in..
- D: Open the bpf module (D) and edit a curve with the mouse. This curve will be sampled and the samples will correspond to the lengths of the subgroups.
- E: In this numbox module (E) you may enter how many samples with which you want the curve to be sampled.
- F: The bpf-sample box (F) samples the break-point function (D) using the number of samples defined in the numbox (E).
- G: The module in (G) receives the value of the length box (B) from the box out (C).
- H: The module g-scaling/sum (H) scales the elements in order to make their sum equal to the number coming out of the in module (G).
- I: The module g-round (G) converts the fractional values coming out of the g-scaling/sum box (H) into a list of integers.
- L: The module group-list takes the notes coming out of the chord box (A) and groups them into subgroups whose lengths are defined by the output of g-scaling/sum (H).

M: Evaluate the chordseq box and see the result. Change either the curve in the bpf module, or the value in the numbox module (E), then re-evaluate the chordseq box to see the result.

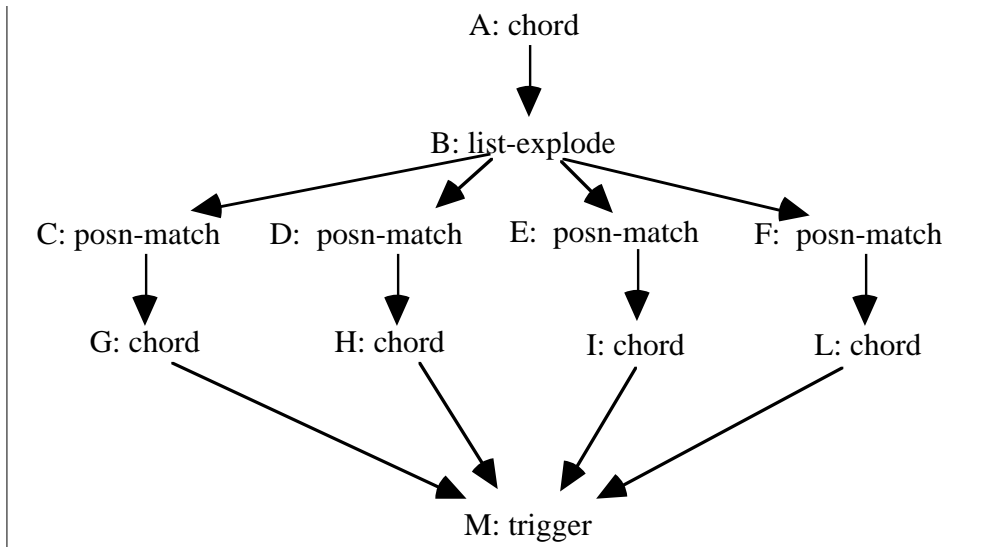
Tutorial 17

List operations: grouping list into many lists.



Description

This patch groups a list of notes (A) into four subgroups and displays each group in a different chord box.

**Boxes used**

chord, list-explode, posn-match, trigger.

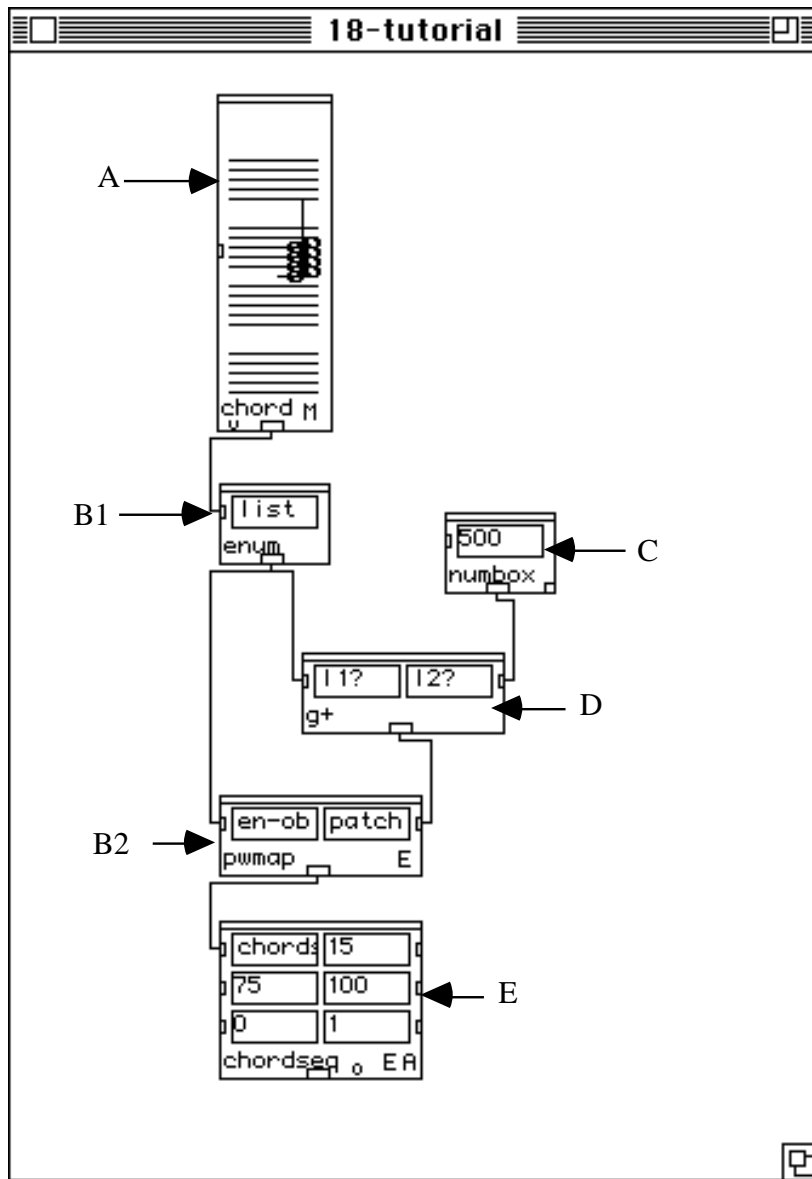
Utilisation

- A: Open the chord box (A) and enter a list of notes in the "arpeggio" mode, then close the module and set it to the "Reorder" mode.
- B: The module list-explode (B) divides the list of notes entered in the chord box (A) into a number of groups defined by the right input of this module.
- C: The posn-match box (C) takes the first element of the list coming out of list-explode (B). If you enter the value 0 in the right input it will take the first element, if you enter the value 1 it will take the second element, if 2, the third, and so on. This box follows a specific rule of Common Lisp which counts the elements in a list starting from 0.
- D: The posn-match box (D) takes the second element of the list coming out of list-explode (B).
- E: The posn-match box (E) takes the third element of the list coming out of list-explode (B).
- F: The posn-match box (F) takes the fourth element of the list coming out of list-explode (B).
- G: This chord box (G) shows the first subgroup of the list you have entered in the chord box (A).
- H: This chord box (H) shows the second subgroup of the list you have entered in the chord box (A).
- I: This chord box (I) shows the third subgroup of the list you have entered in the chord box (A).
- L: This chord box (L) shows the fourth subgroup of the list you have entered in the chord box (A).
- M: The trigger box launches the evaluation of many patches in sequence. The order of evaluation is the same as the order of the inputs. Evaluate the trigger box (M) and see the result in the chord boxes G, H, I and L.

See also in your Reference manual: list-explode, posn-match and trigger.

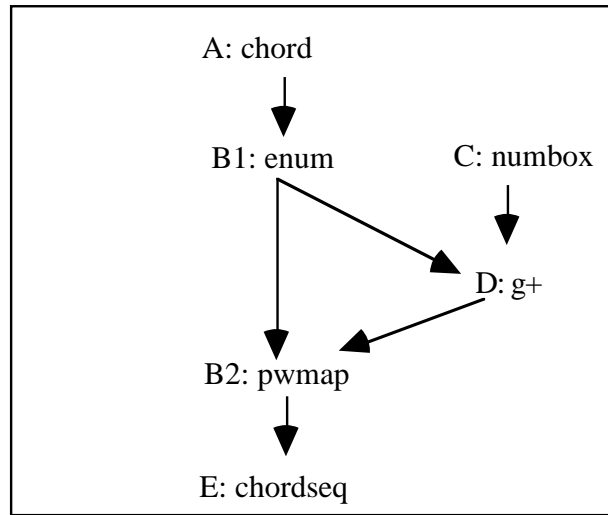
Tutorial 18

Iteration: transposition of each note of a sequence.



Description

This patch transposes of each note entered in a chord (A). Here, the operation is repeated identically for each element: it is an iterating operation.



Boxes used

chord, chordseq, g+, pmap (enum), numbox.

Utilisation

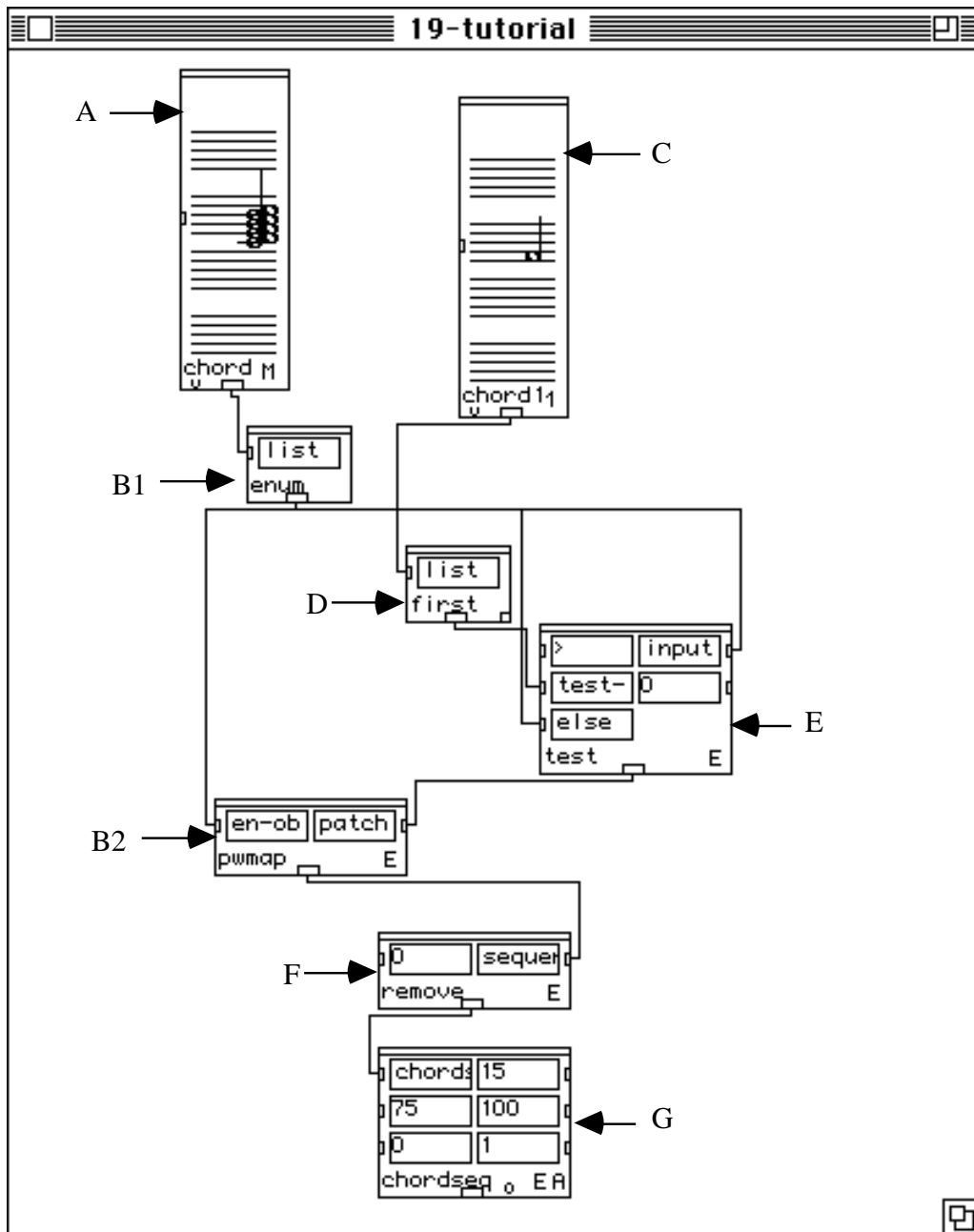
- A: Open the chord and enter a sequence of notes. As before, use the "arpeggio" mode to enter the notes, and then the "Reorder" mode to send out the values.
- B1: The module enum is always connected with the module pmap. When you invoke the module pmap it appears, in the patch window, already connected with a second module enum. These two modules together are able to make iterating operations. enum takes the elements one by one in its input.
- C: Enter the transposition value in the numbox. (C)¹.
- D: The addition box, g+, transposes the notes by adding the transposition value, entered in the numbox (C), to the single value caught by the enum box.
- B2: The module pmap (B) is used to close the transposition process. That is to say that the enum box transposes only one element of a list, and the pmap box allows it to repeat the same process for all the elements entering in the enum box.
- E: Evaluate the chordseq box to see the result. Next, try changing the list of notes in the chord box (A) and the transposition value in the numbox module. Re-evaluate the chordseq box and look at the new result.

See also in your Reference manual: pmap.

1. See also tutorial number 1.

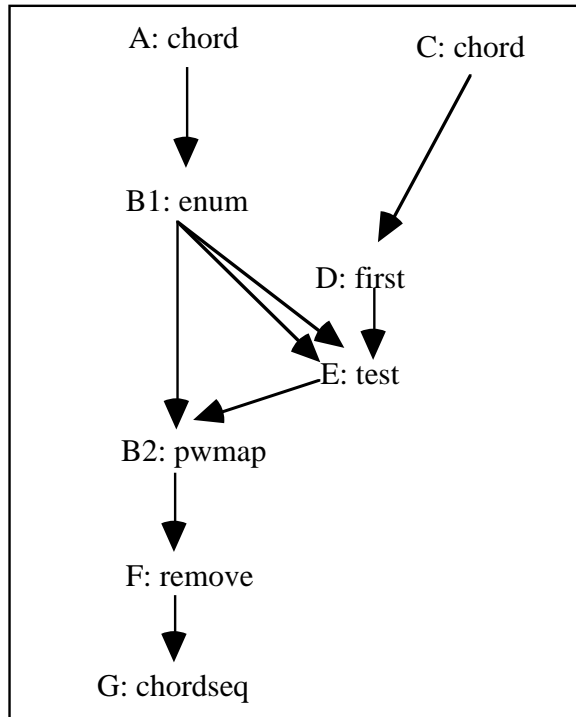
Tutorial 19

Iteration: filtering notes.



Description

This patch applies a low-pass or a high-pass filter to a list of notes (A). The process is done through an operation repeated for each note in the list: it is an iterating operation.



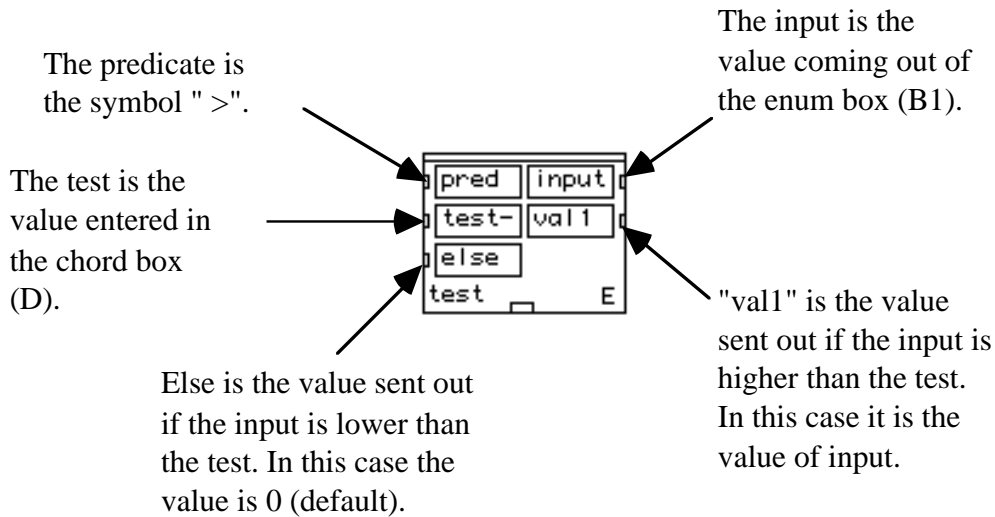
Boxes used

chord, chordseq, first, pormap (enum), remove, test.

Utilisation

- A: Open the chord and enter a sequence of notes using the "arpeggio" mode first, then the "Reorder" mode to send out the values.
- B1: The module enum is connected, as before, with the module pormap. enum catches the elements, one by one, as they come out of the chord box (A).
- C: Open this chord box (C) and enter the note that will be used as a threshold for the filter to be imposed upon the list of notes coming out of the chord box (A).
- D: The first box (D) takes the first element of the list coming out of chord box (C)¹.
- E: The test box (E) makes a comparison between the "input" and "test" inputs, accordingly, with a predicate "pred".

1. See tutorial number 4, paragraph B.



In this example, the predicate is the symbol ">", the "input" is the value coming out of enum box and the "test" is the value entered in the chord box (D).

First the enum box takes the first note of the sequence entered in chord box (A), then the test box makes a comparison between this note and the note entered in the other chord box (D). If the note sent out by enum is higher than the note entered in the chord box (D), the test output will be the note sent by enum box. On the contrary, if the note sent out by enum is lower than the note entered in the chord box (D), the test output will be the value 0.

B2: The module pwmap (B) closes the process of comparison. That is to say that the comparison is repeated for all the notes of the sequence you have entered in the chord box (A). The result of the process is the following list of values: (6000 6200 6400 6500 0 0 0 0)

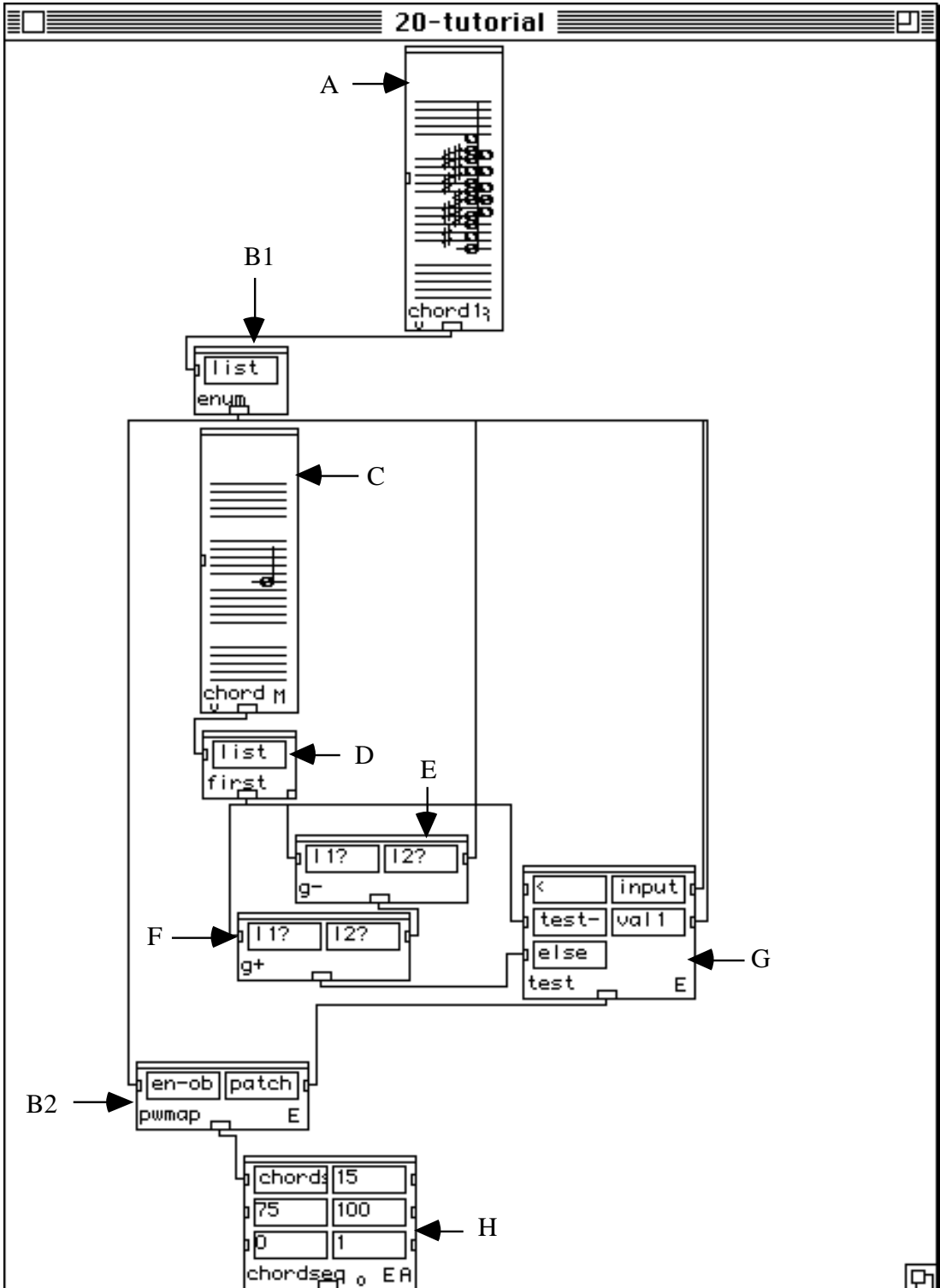
F: The remove box (F) removes all zeros from the list coming out of the module pwmap (B).

G: Evaluate the chordseq to see the result. Try changing either the sequence of notes in the chord box (A), or the predicate "pred" in the test box with the symbol "<", then re-evaluate the chordseq and open the box view the result.

See also in your Reference manual: remove and test.

Tutorial 20

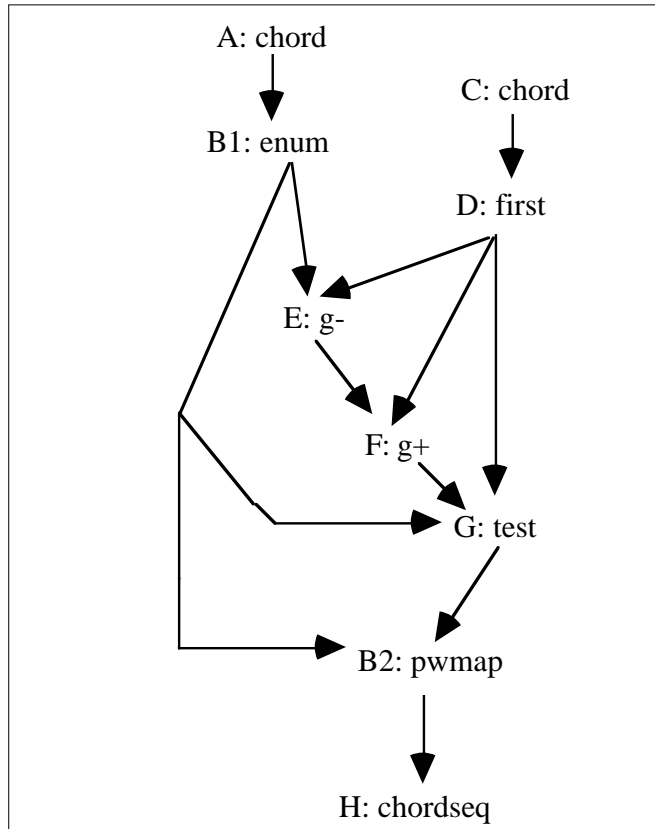
Iteration: reflection of notes.



Description

This example creates a reflection of the notes (A) that are lower or higher than an axis-note (C).

Patch structure



Boxes used

chord, chordseq, first, g+, g-, pwrap (enum), test.

Utilisation

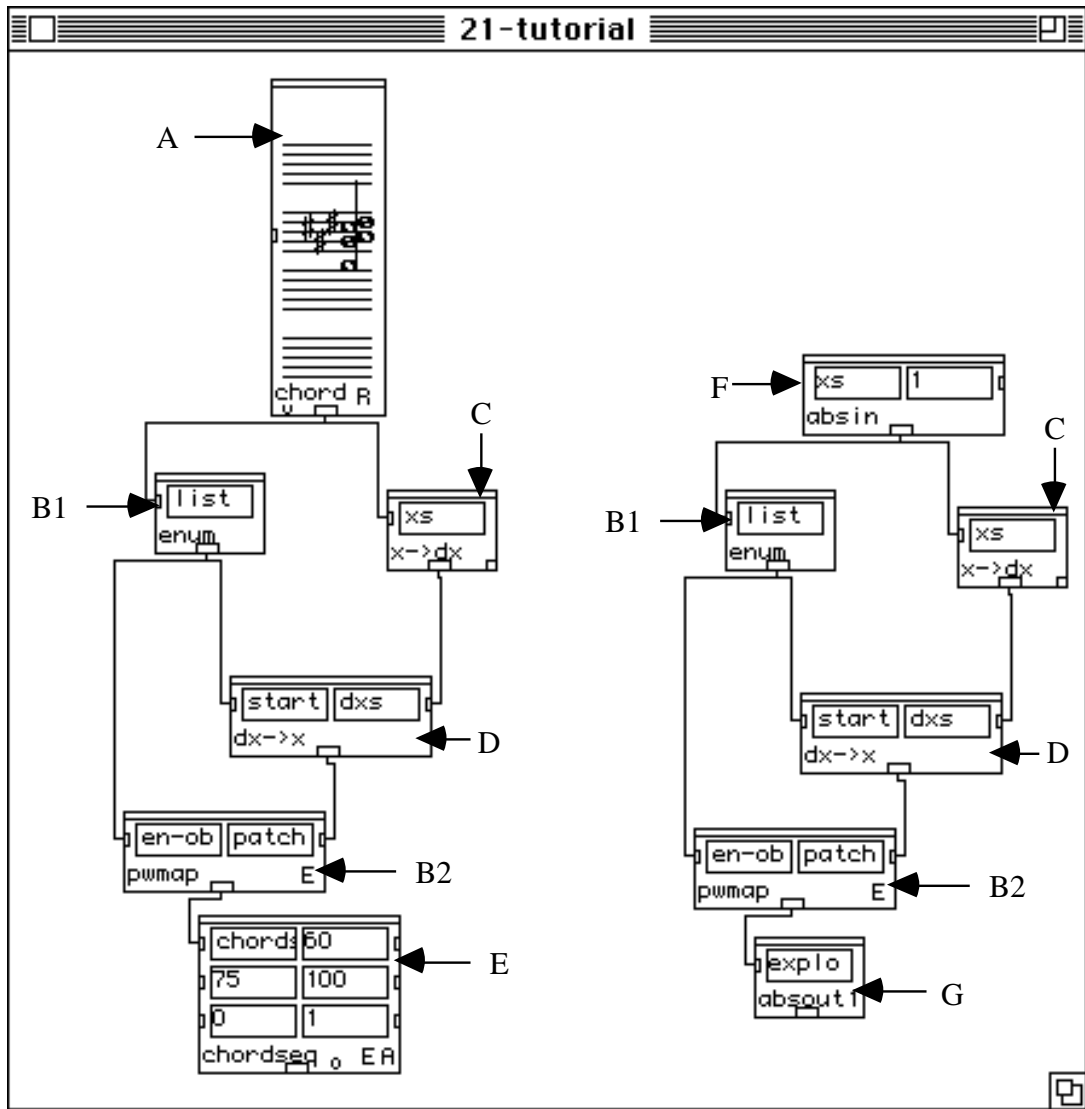
- A: Open the box, enter a sequence of notes in the "arpeggio mode", close the box and select the "Reorder" mode.
- B1: The module enum is automatically connected to the module pwrap. enum takes the elements coming out of the chord box (A) one by one.
- C: In the chord box (C), edit the note that will correspond to the reflection axis of the notes entered in chord box (A).
- D: The first box takes the first element of the list coming out of chord box (C).
- E: The module g- (E) outputs the difference between the note entered in the chord box (C) and the value coming out of enum box (B1).
- F: The g+ box (F) calculates the sum between the value coming out of the module g- (E) and the note entered in the chord box (C).
- G: The test box (G) makes the following comparison with the predicate ">": if a note of the sequence entered in chord box (A) is higher than the note defined as an

axis of reflection, we keep this note. Otherwise, if a note in the sequence is lower, we will calculate its reflection by sending out the analogous note from the list coming from the module g+ (F), (a reflected version of the sequence).

- B2: As we have already seen, the module pwmap (B2) allows us to repeat the same process for all the elements coming out of the chord box (A).
- H: Evaluate the chordseq box (H) in order to see the result. Now change in the test box (G) the predicate with the symbol "< ", re-evaluate the chordseq box and look at the new result.

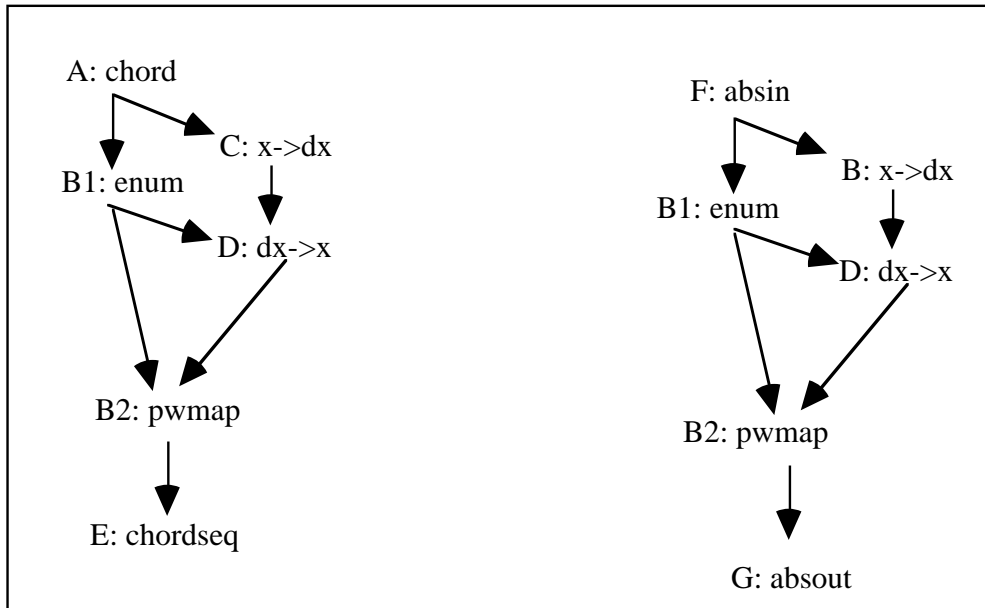
Tutorial 21

Abstraction: multiplication of chords and the concept of abstraction.



Description

This patch creates a sequence of chords created by transposing the chord entered in the chord box (A) successively to begin on each note in the sequence. The same process of transformation is rewritten on the right into a single module that we call abstraction.



Boxes used

absin, absout, chord, chordseq, dx->x, pwmap (enum), x->dx.

Utilisation

- A: Open and edit some notes in the chord box (A).
- B1: The module enum (B1), automatically connected to the module pwmap, takes the elements that come out of the chord box (A) one by one.
- C: The x->dx box (C) calculates the difference between each pair of notes in the scale defined in (A)¹.
- D: The dx->x module makes a transposition, starting with the value coming from the module enum (B1), constructed on the intervals deriving from the x->dx box (C).
- B2: The module pwmap (B2) allows the same process to be repeated for all the elements coming out of the chord box (A). For each note entered in the chord box (A), it outputs a transposed chord made up of the same intervals in the original chord.
- E: Evaluate the chordseq box, open this box and see the result.
- F: This module absin (F) together with the module absout (G) allows us to represent all the modules that are in the patch by a single box, called an abstraction².
- G: The module absout (G) is the box which corresponds to the output of the abstraction, and the place where you can give your abstraction a name.

1. See also tutorial 4 at paragraph B.

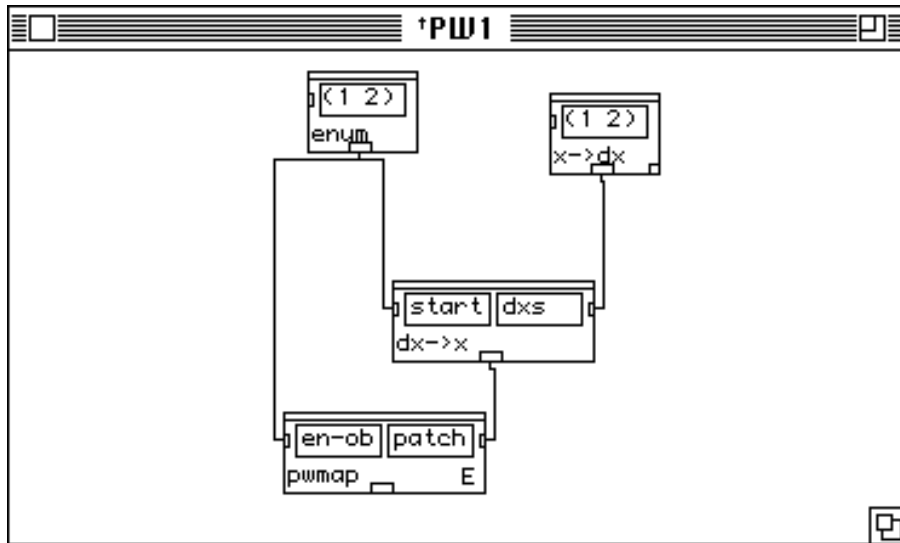
2. See also chapter 3 of your Introduction manual: creating an abstraction.

Why create an abstraction?

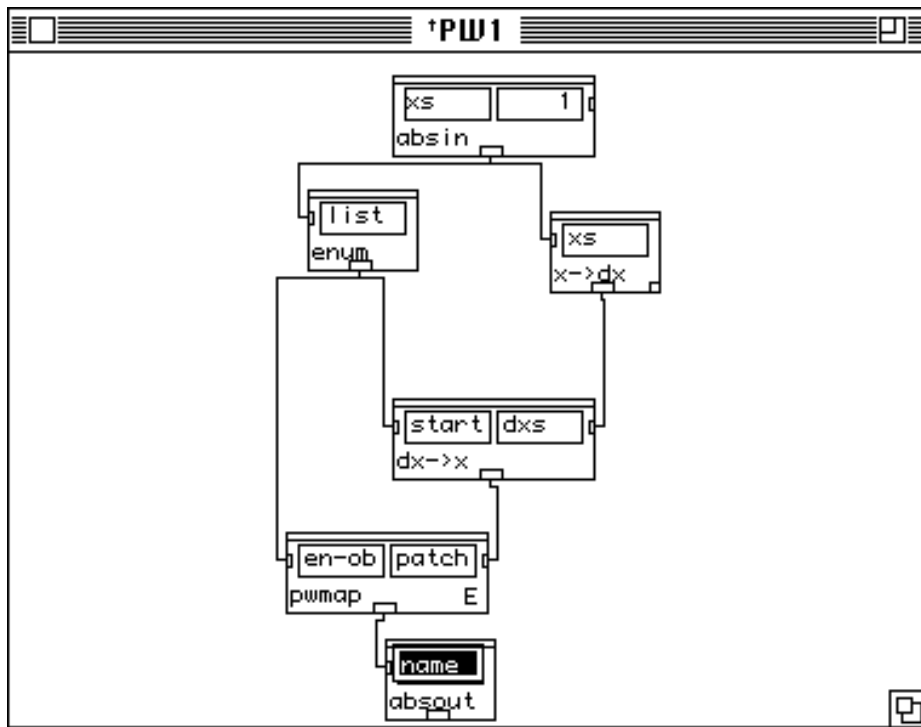
With an abstraction we can reduce a complete patch into a single module, thus saving space in the patch window. We can also copy and paste the abstraction to have many abstractions (performing the same function) in only one window. Also, by isolating a series of processes into a single box, we can simplify the utilisation of a complex patch. Another reason is that they look nice!

How create an abstraction

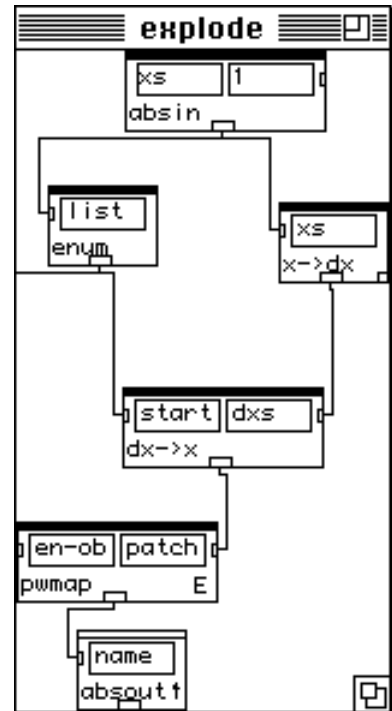
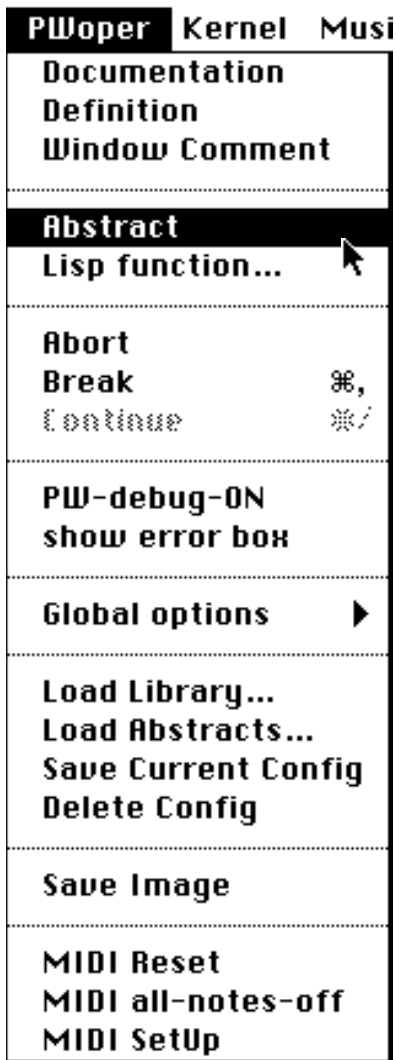
Choose the boxes that you want to reduce into a single module (i.e. an abstraction).



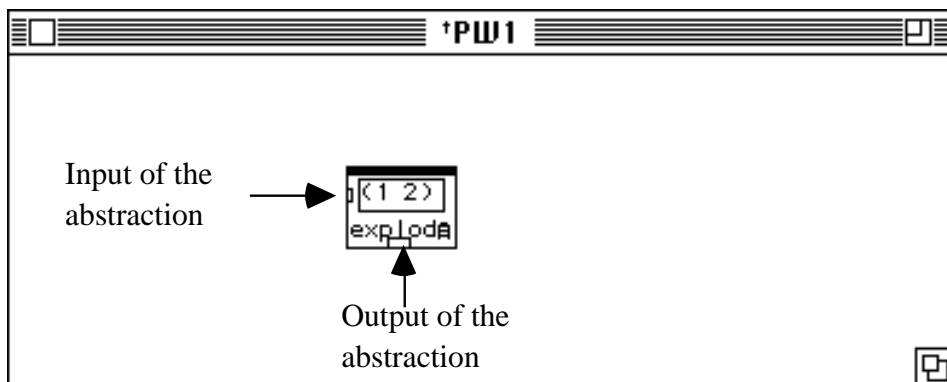
Then, add as many absin modules to the patch as you will need inputs, and add an absout box that will correspond to the output of the abstraction. Next, double-click in the absout rectangle and enter a name for your abstraction.



Select all these boxes and choose "Abstract" in the PWOper menu



All of the selected boxes are transformed into a single box whose name, in this example, is "explode".

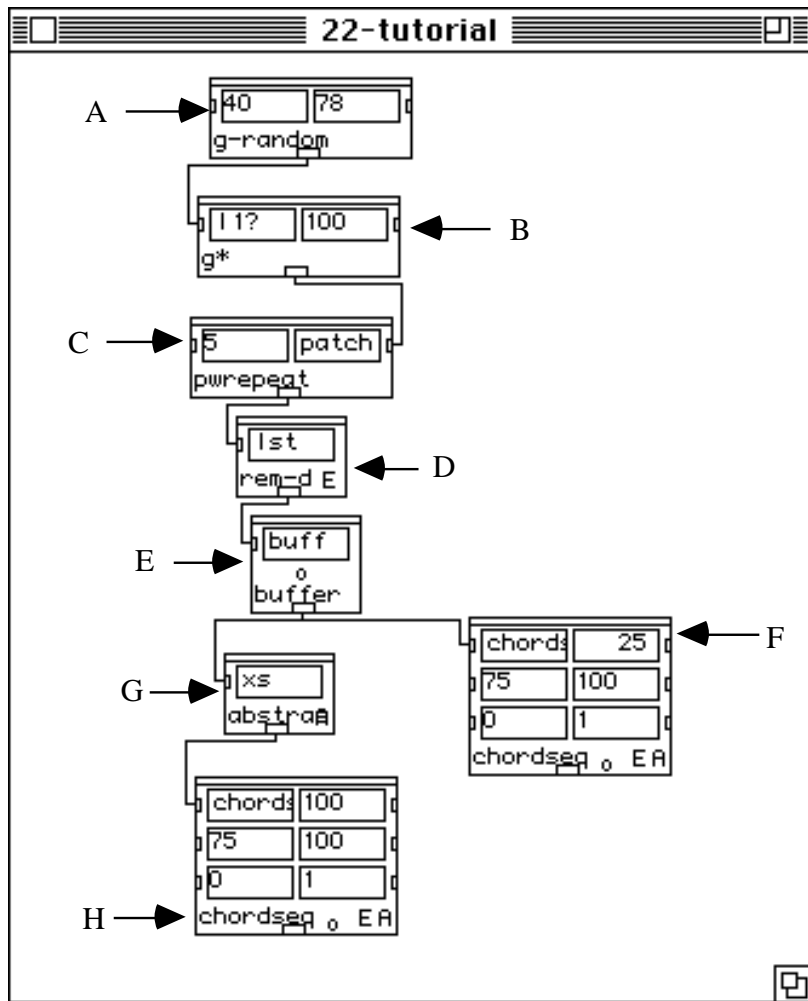


As you can see, there is an input that corresponds to the absin in the patch, and an output corresponding to the absout. In this way, you have an entire patch reduced to a single on-screen box which does the same process of multiplication of chords, as shown in left-hand side of this tutorial patch.

See also in your Reference manual: absin and absout.

Tutorial 22

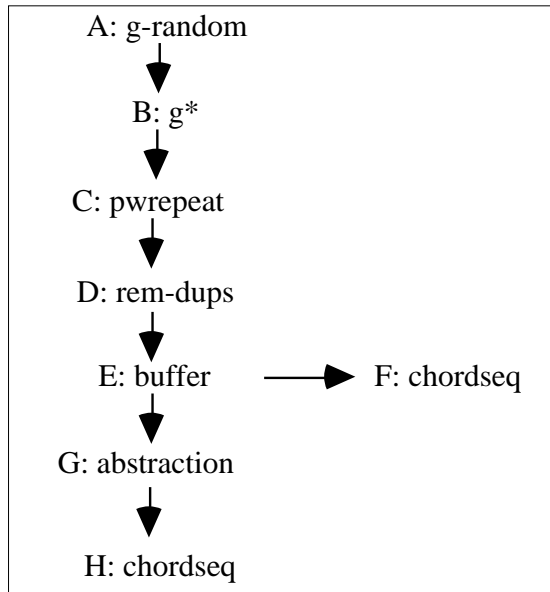
Abstraction: generation and multiplication of chords.



Description

This example generates a random series (A) of notes. Then, using an abstraction, it creates as many scales as there are notes in the sequence, based on the intervals contained in the random series.

Patch structure

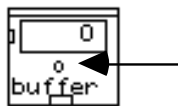


Boxes used

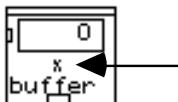
absin, absout, buffer, chord, chordseq, dx->x, g-random, g*, pwmap (enum), pwrepeat, rem-dups, x->dx.

Utilisation

- A: The g-random box (A) calculates a single random value between two limiting values (in this example between 40 and 78).
- B: The g* box (B) multiplies each number coming out of the g-random box (A) by 100.
- C: The pwrepeat box (C) makes the g-random (A) and g* (B) boxes repeat the same process many times (in this example 5 times).
- D: The module rem-dups (D) removes all the duplicates.
- E: The buffer box (E) stores the result of this part of the patch's calculation. Evaluate the buffer box and then close the buffer by clicking on the "o" in the centre of this box.



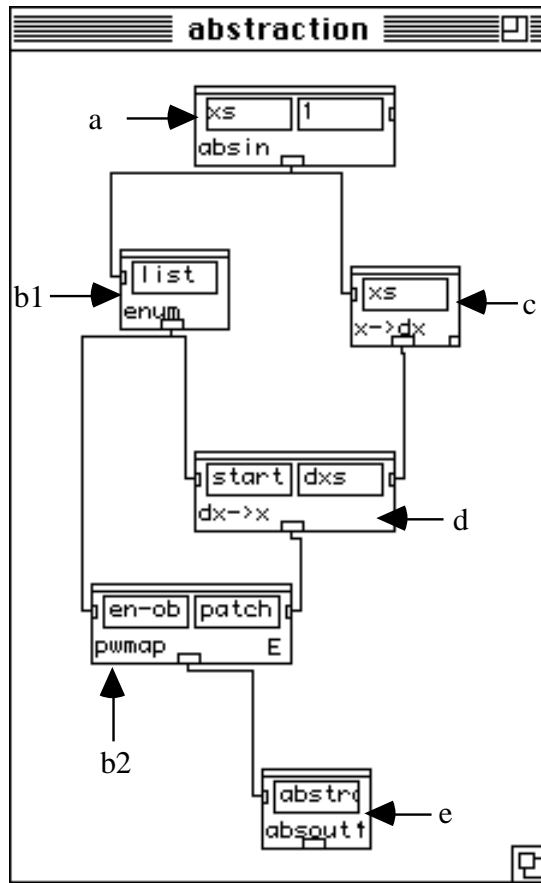
Now the buffer is closed. You can see this by the little x which appears in the centre of the box.



In this state, buffer box has stored the data coming out of the rem-dups box (D).

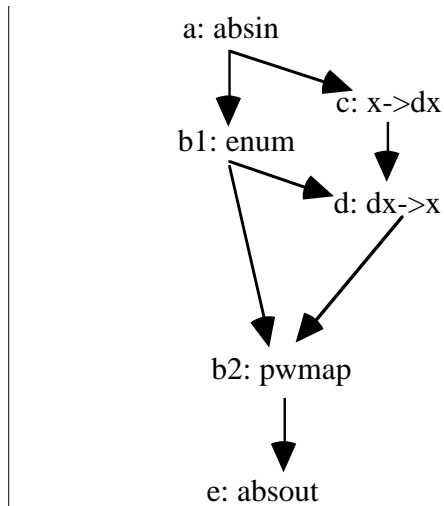
- F: Evaluate the chordseq box (F) to see the result of the random series of notes.

G: Abstraction.

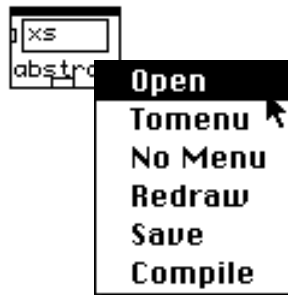


Abstraction Description

This abstraction creates as many scales as the number of notes defined in the random sequence, starting from the notes that enter the absin box (a), and based on a series of intervals calculated in the x->dx box (c).



Open the abstraction by selecting "open" in its menu (or by double-clicking in its lower-right-hand corner) to see the internal structure.

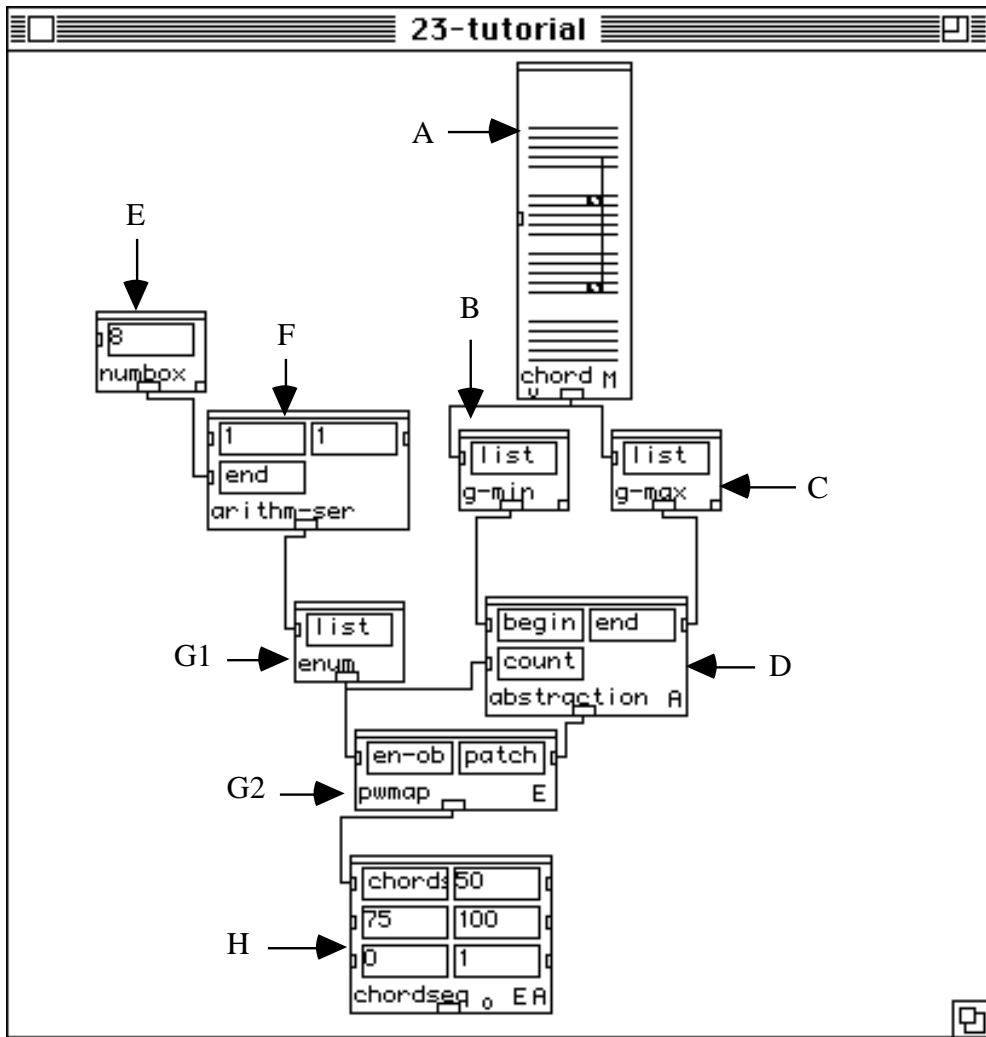


- A: The absin box (a) is the input of the abstraction and it takes the result of the buffer box (E).
 - b1: The module enum, as before, is connected with the module pwmap (b2). enum takes the elements that come out of the absin box (a), one by one.
 - C: The x->dx box (c) calculates the intervals between the notes in the sequence (a).
 - D: The dx->x module (d) makes new sequences that start with the values coming from enum box (b1), and are constructed using the intervals derived from the x->dx box (c).
 - b2: The module pwmap (b2) allows the repetition of the same process for all the elements coming out of absin (a).
 - E: The absout box (g) is the output of the abstraction.
- Back to the main patch window...
- H: Evaluate the chordseq box and open it to see the result. Open the buffer box by clicking on the little "x" in the centre of the box and re-evaluate the chordseq box (H), open it to see the new chords.

See also in your Reference manual: buffer, g-random, pwrepeat and rem-dups.

Tutorial 23

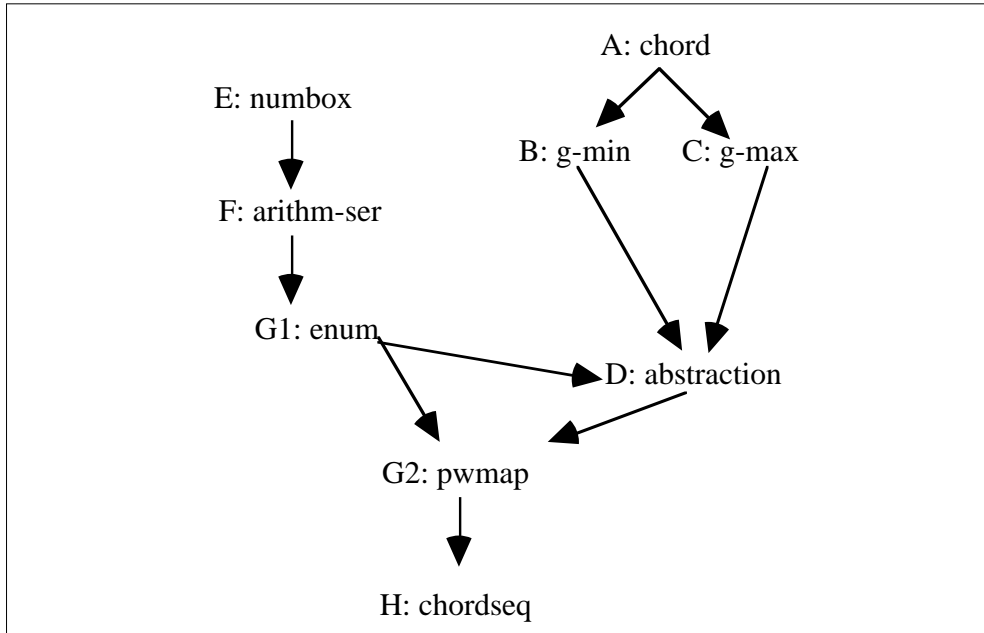
Abstraction: generation of chords.



Description

This patch makes a defined number (E) of chords with a variable number of notes. The notes are chosen randomly in a defined range (A).

Patch structure

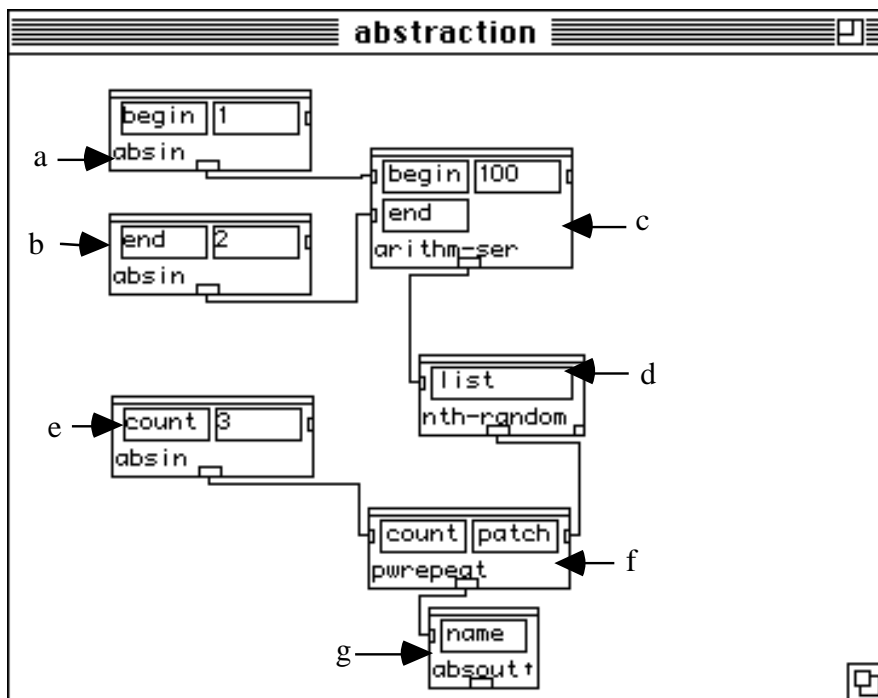


Boxes used

absin, absout, arithm-ser, chord, chordseq, g-max, g-min, nth-random, numbox, pwmap (enum), pwrepeat.

Utilisation

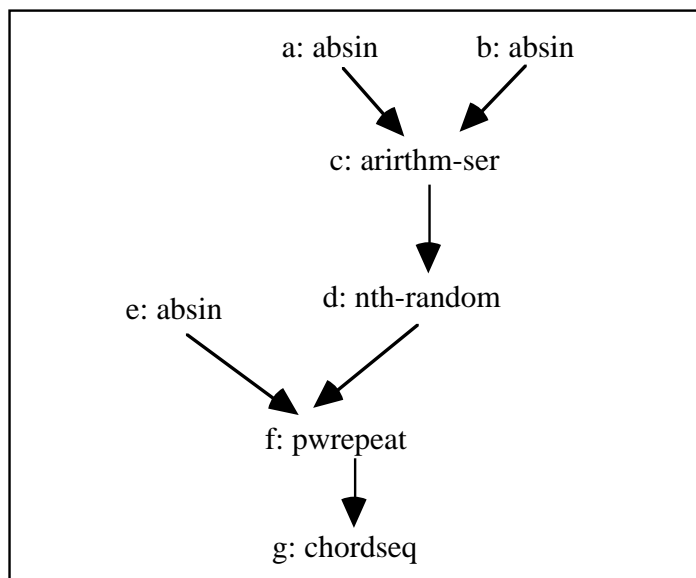
- A: Open the chord box (A) and enter a widely-spaced diad. This diad will correspond to the range in which the notes of chords will be randomly chosen.
- B: The g-min box (B) takes the minimum value of a list. In this example it takes the lower note of the diad entered in the chord box (A).
- C: The g-man box (C) takes the maximum value of a list. In this example it takes the upper note of the diad entered in the chord box (A).
- D: Abstraction.



Abstraction Description

This abstraction creates as many scales as the number of notes defined in the random sequence, starting from the notes that entered in the absin box (a), and based on a series of intervals calculated in the x->dx box (c).

Abstraction structure



Open the abstraction by selecting "open" in its menu and observe the internal structure¹.

A: The absin box (a) corresponds to the arithm-ser « begin » input.

1. See also tutorial 22, paragraph: Abstraction structure.

- B: The absin box (b) corresponds to the arithm-ser « end » input.
- C: The module arithm-ser (c) creates an arithmetic series starting from the lower note entered in the chord box (A) of the patch, and ending with the upper note entered in the same chord box (A). The step size of the arithm-ser (c), in this example, is equal to 100 (one semitone).
- D: The nth-random box (d) randomly extracts an element of the arithm-ser (c).
- E: The absin box (b) corresponds to the pwrepeat « count » input.
- F: The pwrepeat box (f) repeats the random extraction 5 times made by the nth-random box (d).
- G: The absout box corresponds to the output of the abstraction.

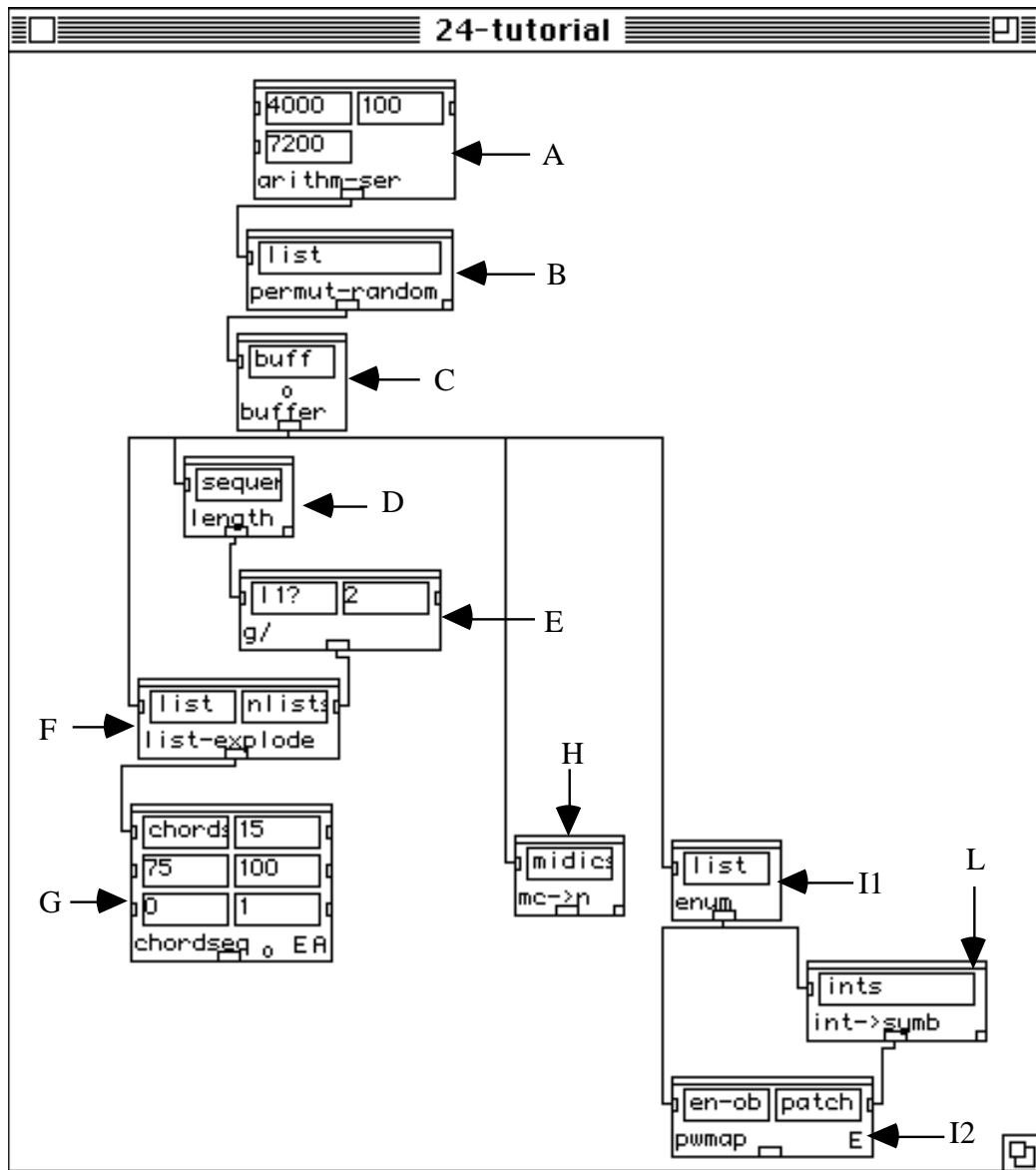
Back to the main patch...

- E: In the numbox module (E), enter how many chords you want to calculate in the patch.
- F: The module arithm-ser (F) creates an arithmetic series starting from value 1, ending with the value entered in the numbox module (E), with a step of 1.
- G1: The module enum (G1), connected with the module pwmap (G2), takes the elements that come out of the arithm-ser box (F), one by one.
- G2: The module pwmap (G2) allows us to repeat the same process for all the elements coming out the arithm-ser box (F).
- H: Evaluate the chordseq box (H) in order to see the result. Then try changing either the diad in the chord box (A), or the input number in the numbox (E), then re-evaluate the chordseq box (H), open it and look at the new result. Try evaluating the chordseq box (H) many times to better see the random generation of chords.

See also in your Reference manual: arithm-ser, g-max, g-min, nth-random.

Tutorial 24

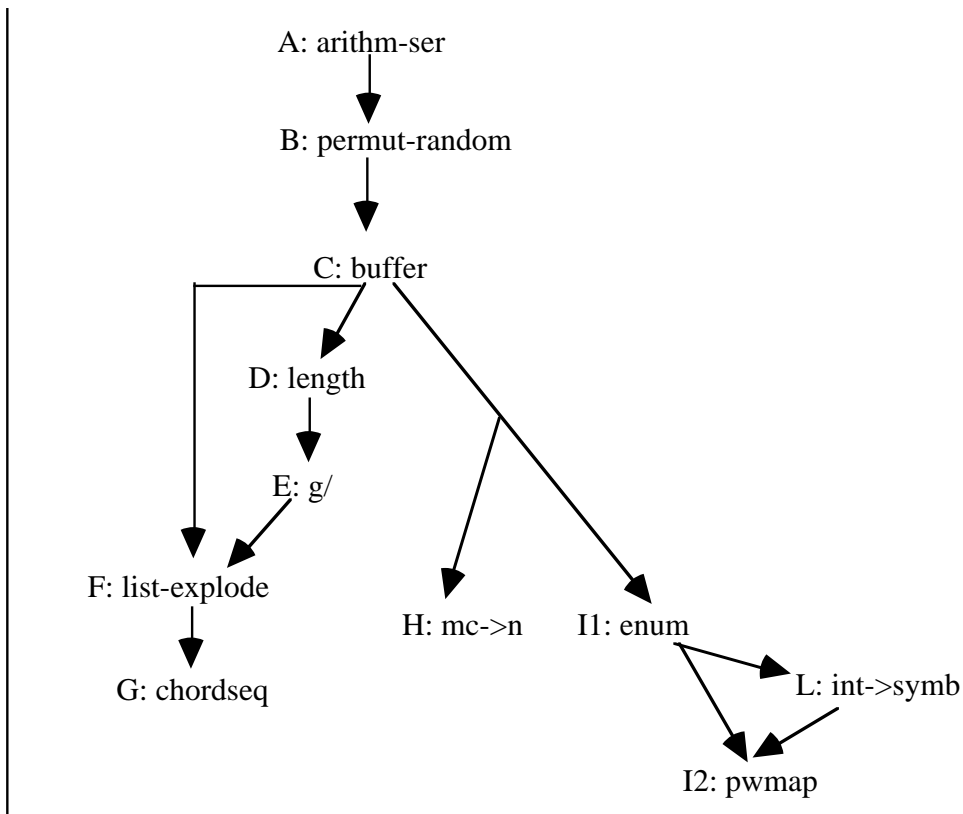
Conversions: notes to symbols.



Description

This patch makes a random list of notes that is grouped into diads. Then it converts the midicents values into a symbolic notation.

Patch structure



Boxes used

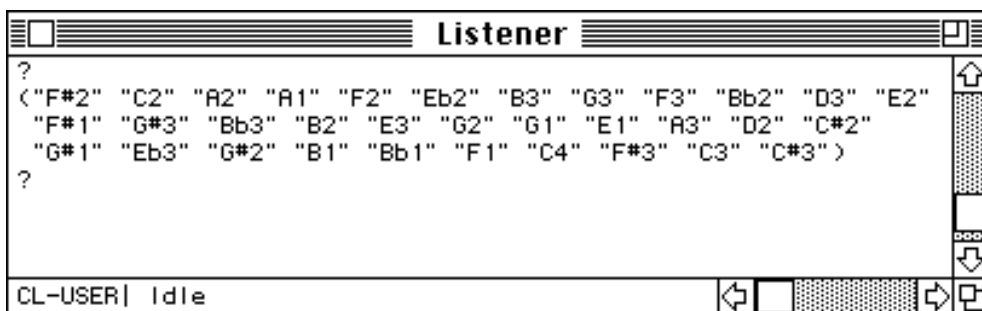
arithm-ser, buffer, chordseq, flat, g/, int->symb, length, list-explode, mc->n, permut-random, pwmap (enum).

Utilisation

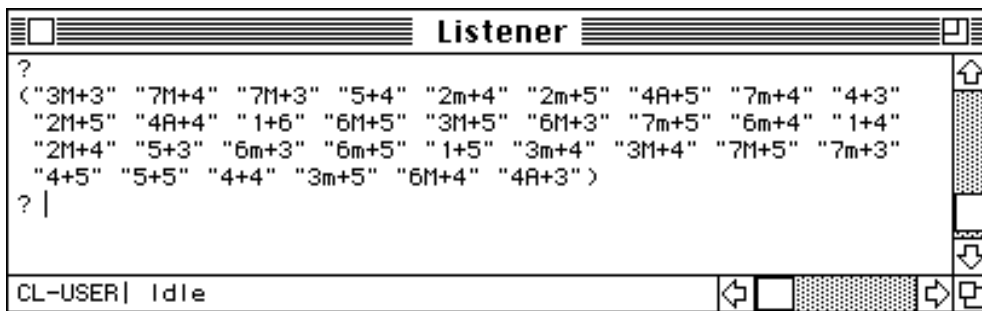
- A: The module arithm-ser (A) creates an arithmetic series starting from value 4000 (corresponding to the note E1), ending with the value 7200 (corresponding to the note C4) and with a step of 100 (a semitone).
- B: The permut-random box (B) makes a random permutation of the list coming out of the module arithm-ser (A).
- C: The buffer box (C) stores the result of this part of patch calculation, (the result of the module permut-random (B)). Evaluate and then close the buffer box by clicking in the centre of the module¹.
- D: The length box (D) calculates the number of elements coming out of the buffer box (C).
- E: The module g/ (D) divides the value sent out by the length box (C) by 2.
- F: The module list-explode (E) divides the list of values coming out of the buffer box (C) into a number of groups that is equal to the half the length of the incoming list (the list becomes grouped into diads).
- G: Evaluate the chordseq box to see the randomly generated diads.

1. See also tutorial 22 at paragraph E.

- H: The mc->n box (H) converts the midicents into symbolic notation. That is to say that if it receives the value 6000 in its input, it will return the symbol C3. If it receives the value 6100, it will return the symbol C#3, etc... Evaluate this box and look at the results in the Listener window.



- I1: The module enum (I1), connected to the module pmap (I2), takes the diads that come out of the buffer box (C), one by one.
- L: The int-symb box (L) converts a diad expressed in midicents into a symbolic notation. That is to say that if it receives the notes (6000 6700), it will return (C3 G3).
- I2: The module pmap (I2) allows the same process to be applied to all the diads coming out of the buffer box (C). Evaluate the pmap box (I2) and see the result in the Listener window.

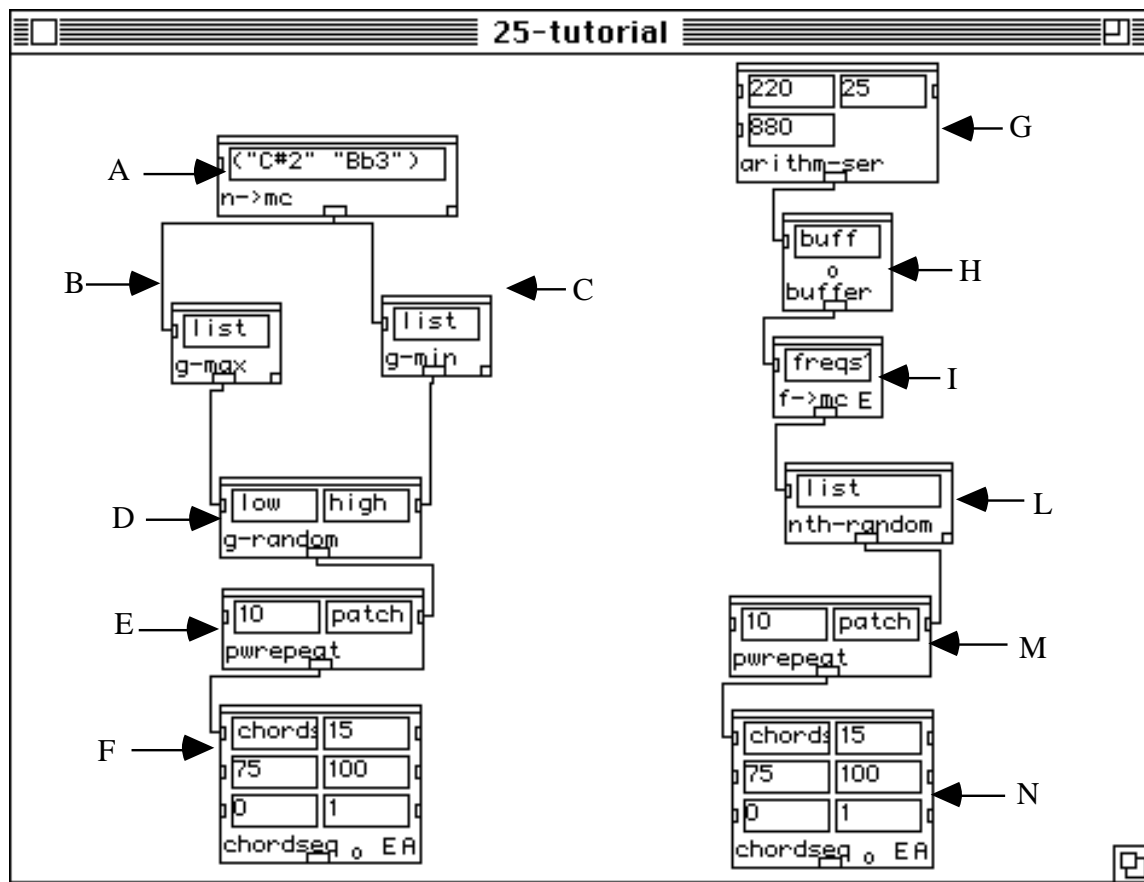


Now open and re-evaluate the buffer box (C). Then close the buffer box and re-evaluate either the chordseq box, the mc->n box or the pmap module to see the different conversions.

See also in your Reference manual: int->symb, mc->n and permut-random.

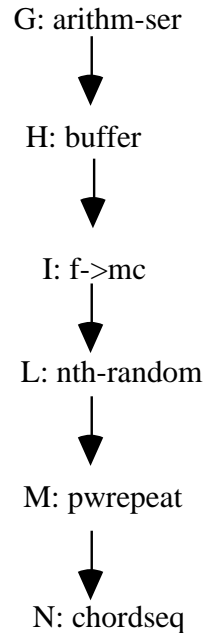
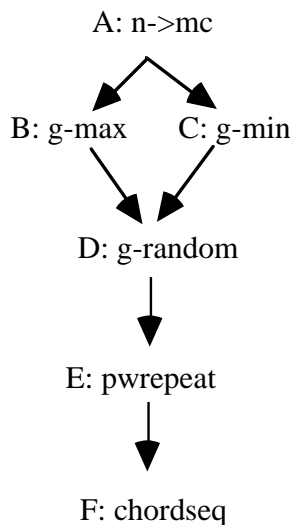
Tutorial 25

Conversions: symbols to notes and frequencies to midicents.



Description

This patch shows two different conversions. The example on the left demonstrates conversion from symbolic notation to midicents. In this case, the patch takes an interval entered in symbolic notation, then generates a random series of midicents. The example on the right demonstrates conversion of frequencies (in Hertz) to midicents.

**Boxes used**

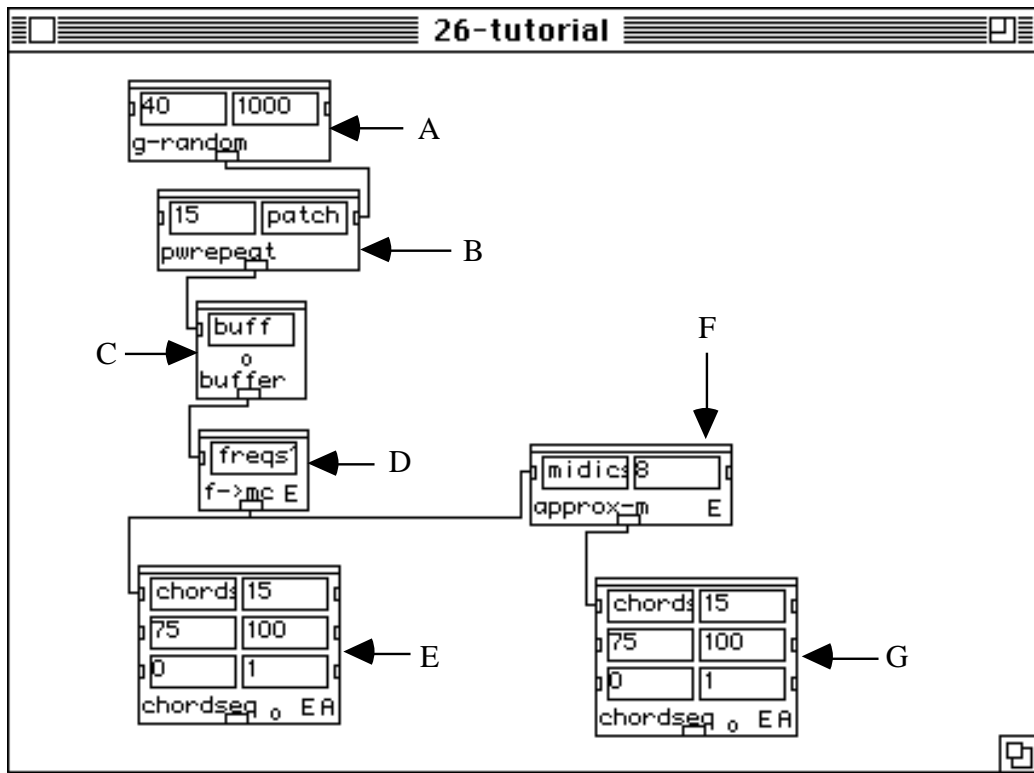
arithm-ser, buffer, chordseq, f->mc, g-alea, g-max, g-min, g/, g*, nth-random, n->mc, pwrepeat.

Utilisation

- A: Enter a diad in the n->mc box (A), using symbolic notation.
- B: The g-min box (B) takes the minimum value of a list. In this example it takes the lower note of the diad entered in the n->mc box (A).
- C: The g-max box (C) takes the maximum value of a list. In this example it takes the upper note of the diad entered in the n->mc box (A).
- D: The g-random box (D) calculates a random value between two values entered in the n->mc box (A).
- E: In this example, the pwrepeat box (E) repeats the random generation made by the g-random box (D) 10 times.
- F: Open and evaluate this chordseq box (F) many times to see the different results.
- G: In this example, the module arithm-ser (G) creates an arithmetic series starting from 220, ending with 880, and using a step of 25. These values will be considered as frequencies Hertz.
- H: Evaluate, then close the buffer box (H) by clicking in the centre of the module.
- I: The module f->mc (I) converts the frequencies it receives from the buffer box (H) into midicents.
- L: The nth-random box (L) randomly extracts an element from the buffer box (H).
- M: The pwrepeat box (M) repeats the random extraction done by the nth-random box (L) 10 times, in this example.
- N: Open and evaluate this chordseq box (N) many times to see the different results.
- See also in your Reference manual: f->mc and n->mc.

Tutorial 26

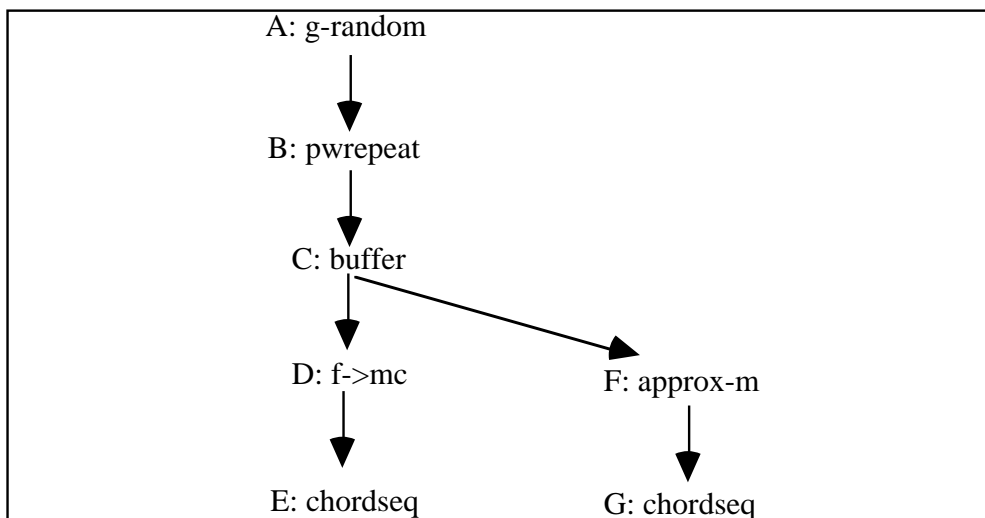
Approximation



Description

This patch generates a random series of frequencies (Hertz), that are converted into midicents using different approximations.

Patch structure

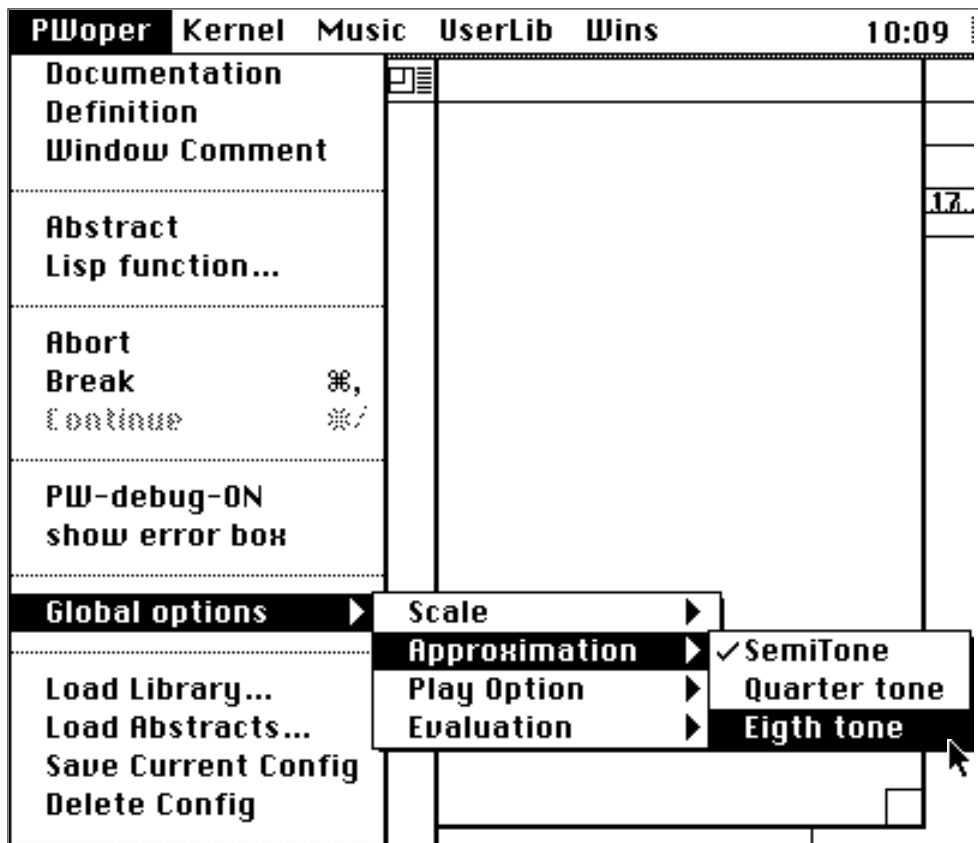


Boxes used

approx-m, buffer, chordseq, f->mc, g-random, pwrepeat.

Utilisation

- A: The g-random box (A) returns a random number, in this example, it is between 40 and 1000.
- B: The pwrepeat box (B) repeats the random number generation made by the g-random box (A) 15 times.
- C: The buffer box (C) stores the result of the above part of patch calculation. Evaluate, then close the buffer box by clicking in the centre of the module.
- D: The module f->mc (D) converts the frequencies it receives from the buffer box (C) into midicents.
- E: In the PWoper menu set the "Global option" on "Eighth [sic!] tone" approximation:



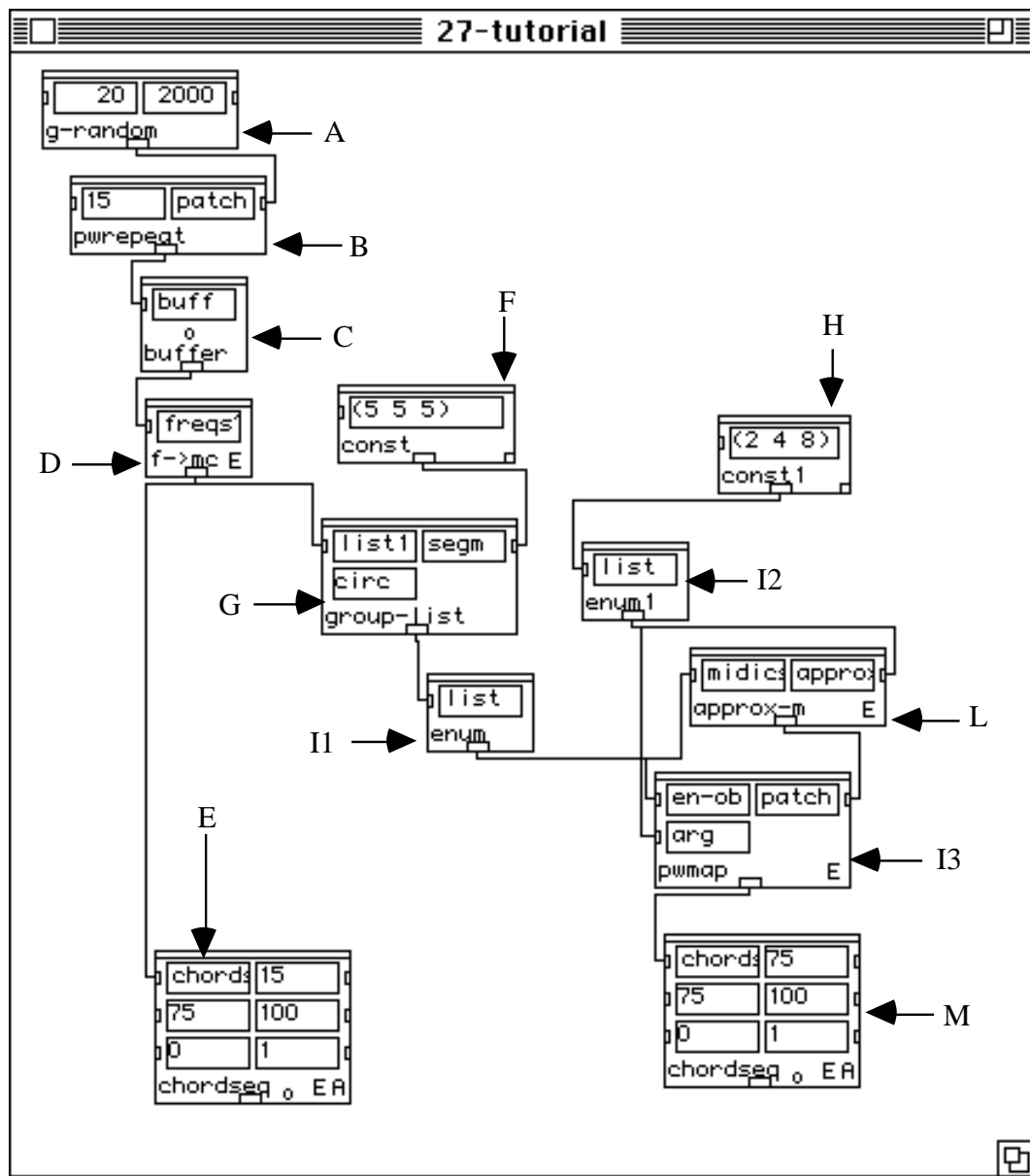
Now evaluate and open the chordseq box (E) to see the randomly chosen result.

- F: The approx-m box (F) takes a midicent value and sends out an approximation that is defined in the "approx" input. If you enter the value 2 in this input, the box approximates in semitones, if you enter 4, it approximates in quarter-tones, if 8 then in eighth-tones, etc...
- G: Evaluate the chordseq (G) and look at the result. Then try changing the approximation value, and re-evaluate the chordseq box (G) to see the newly obtained approximation.

See also in your Reference manual: approx-m

Tutorial 27

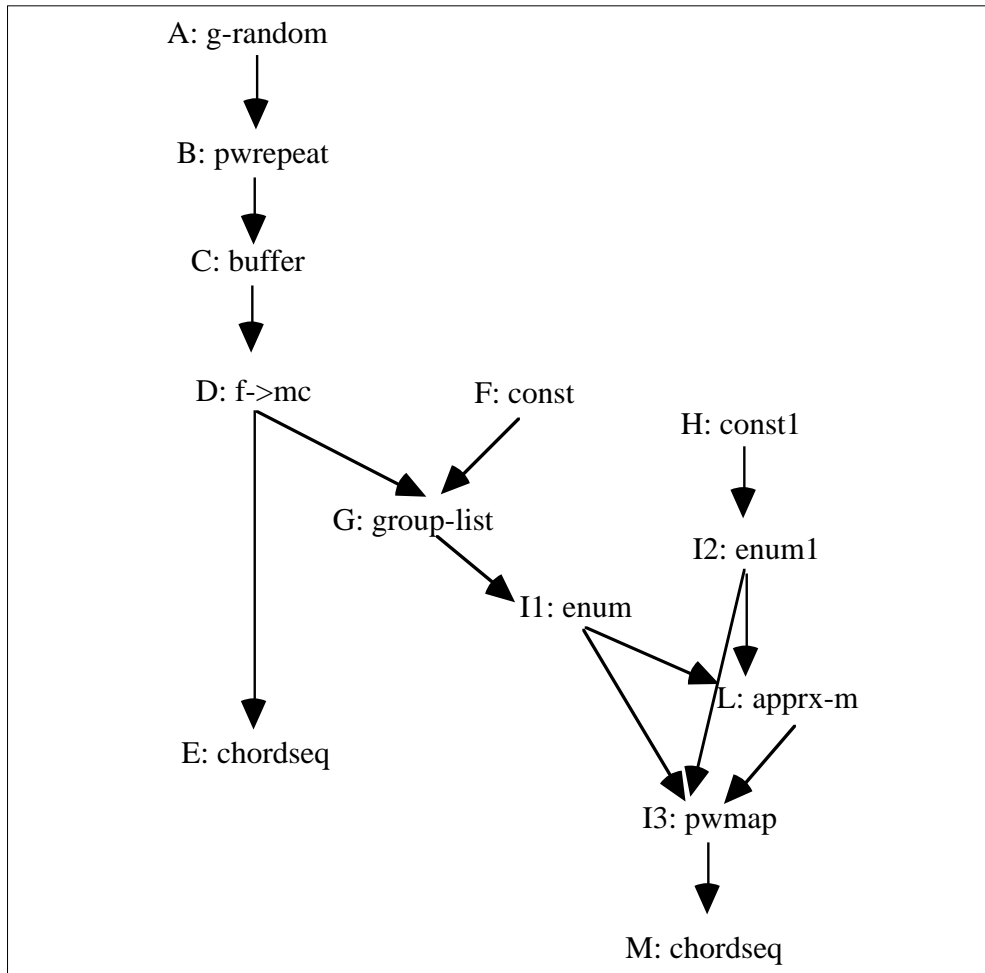
Approximation (part II)



Description

This example generates a random series of frequencies (expressed in Hertz) that is first converted into midicents. The sequence is then grouped into three subgroups which use each different approximations.

Patch structure



Boxes used

apprx-m, buffer, chordseq, const, f->mc, group-list, g-random, pwmap (enum, enum1), pwrepeat.

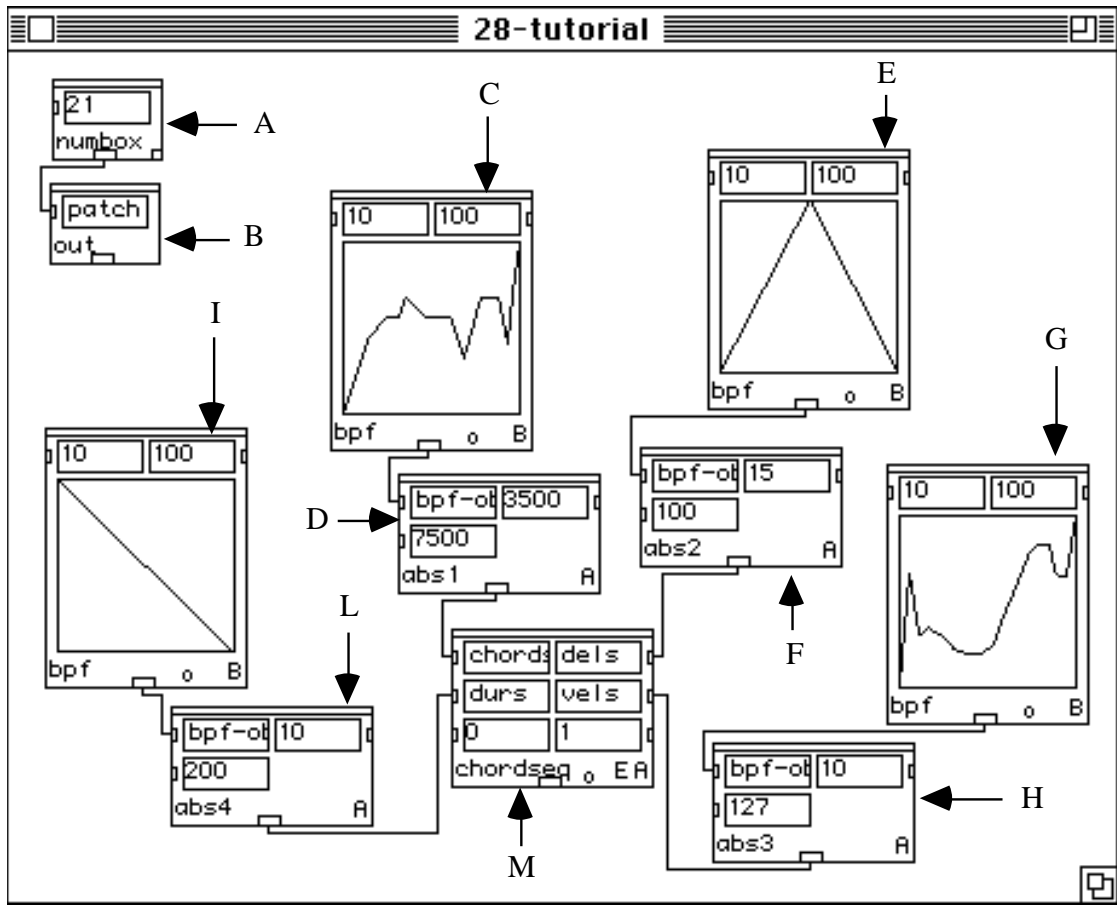
Utilisation

- A: The g-random box (A) returns a random number between two given values (in this example 20 and 2000).
- B: The pwrepeat box (B) repeats the random generation 15 times.
- C: The buffer box (C) stores the result of this random generation. As before, evaluate and then close the buffer box by clicking in the centre of the module.
- D: Now, the module f->mc (D) converts all the frequencies coming out of the buffer box (C) into midicents.
- E: As before, set the "Global option" on "Eighth tone" approximation, evaluate and open the chordseq box (E) in order to see the random generation of notes without approximation.
- F: The const box (F) is used to define the lengths of each of the subgroups you wish to obtain (here we are asking for three subgroups).

- G: The group-list box (G) groups the random sequence of notes coming out of the module f->mc (D) into three subgroups, (that have been defined in the const box (F)).
- H: In the module const (H), enter three different approximation values. Remember that value 2 corresponds to semitone approximation, 4 to quarter-tone and 8 to eighth-tone approximation.
- I1: The module enum (I1), connected to the module pormap (I3), takes each subgroup that comes out of the group-list box (G), one by one.
- I1: At the same time, the module enum1 (I2), connected to the module pormap (I3), takes each value coming out of the module const (H), one by one.
- L: The approx-m module (L) makes a different approximation for each subgroup entering in its input. That is to say that it takes the first subgroup "caught" by the enum (I1) and approximates it using the value sent out by the module enum1 (I2). Then it repeats the same operation for the second and the third subgroups.
- I2: The module pormap (I3) allows the same process to be applied to the three subgroups defined in the group-list box (G).
- M: Evaluate the chordseq box (M) to see the three different approximations. Try changing the two values in the g-random box (A), then open, evaluate and close the buffer box (C), next re-evaluate the chordseq box (M) to see the new result.

Tutorial 28

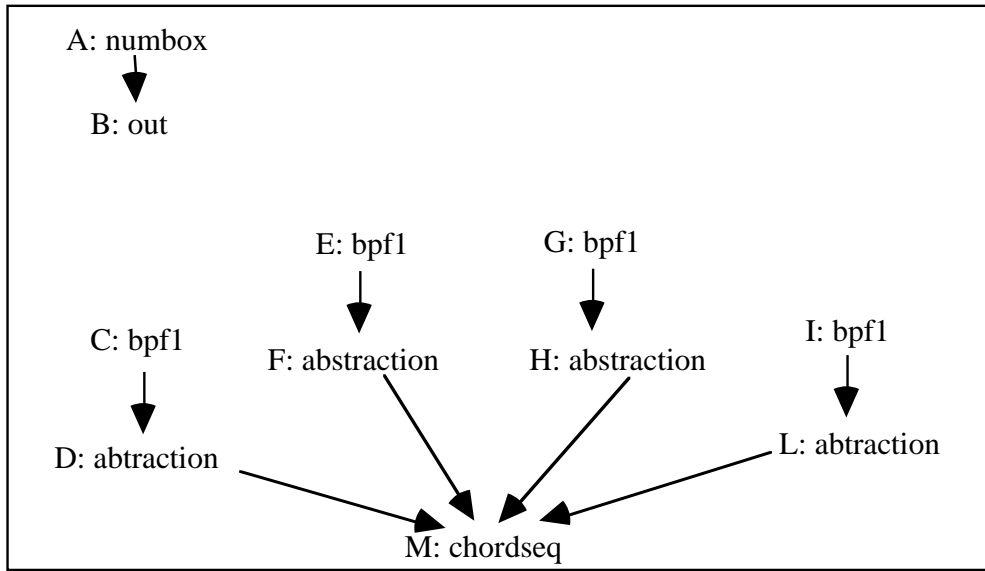
Musical notation: proportional notation.



Description

This patch uses four break-point functions that are converted into different values by four instances of the same abstraction.

Patch structure



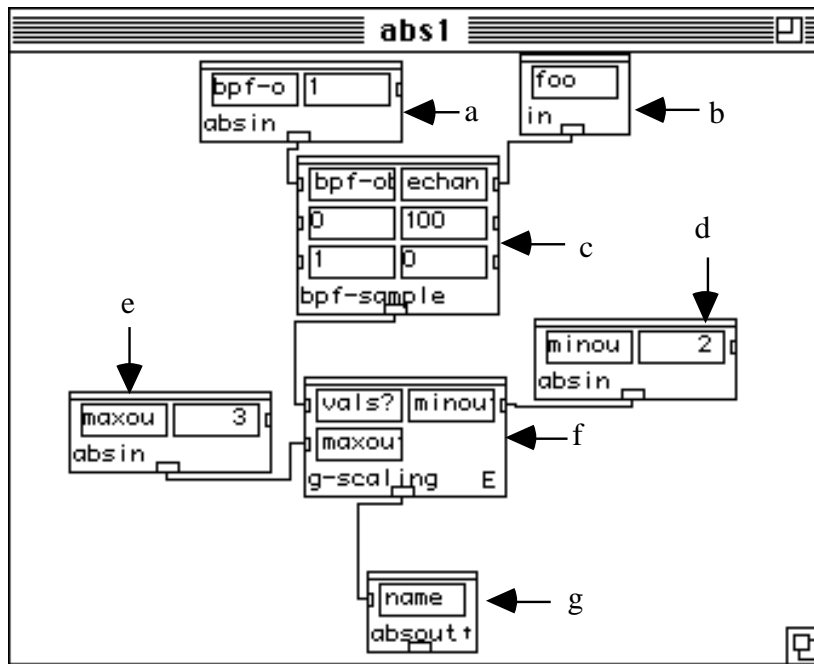
Boxes used

absin, absout, bpf, bpf-sample, g-scaling, in, numbox, out.

Utilisation

- A: In the numbox (A) enter how many values you will use to sample each break-point function.
- B: The module out (B) sends the value coming out of the numbox (A) to the in modules that are in the abstractions themselves.
- C: Open and edit a curve in the bpf box (C). This break-point function will be converted into a different sequence of notes by each of the four abstractions (D)¹.
- E: Abstraction.

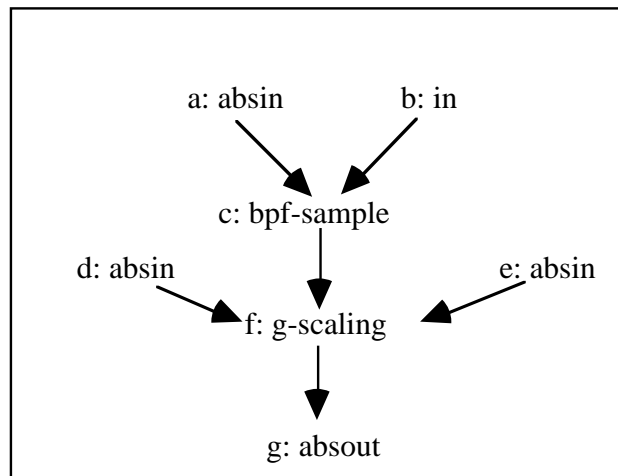
1. See tutorial 10.



Abstraction Description

This abstraction samples the break-point function a pre-defined number of samples coming from the numbox (A) located in the main patch. The abstraction then scales the samples between the values entered in the two absin boxes (d) and (c).

Abstraction structure



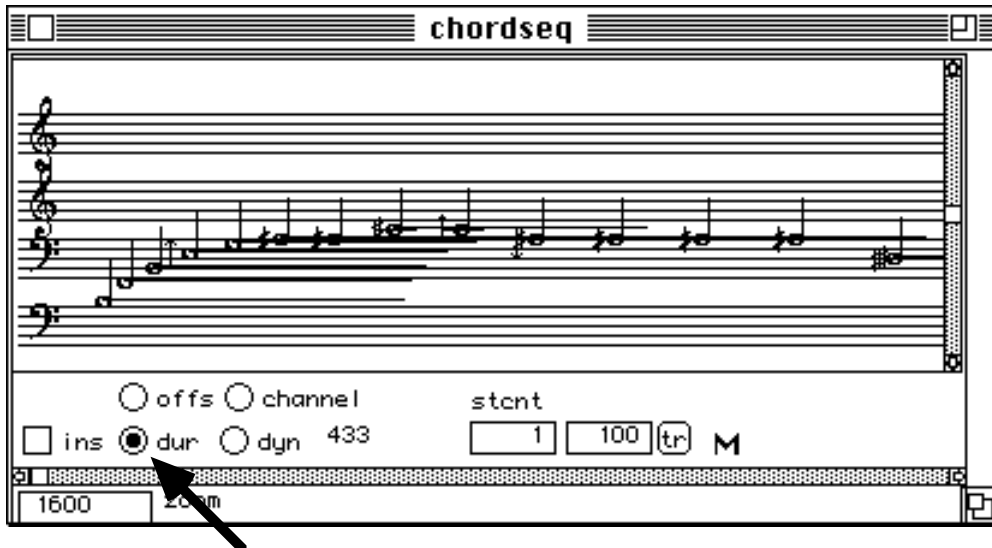
Open the abstraction by selecting "open" in its menu and observe the internal structure

Utilisation

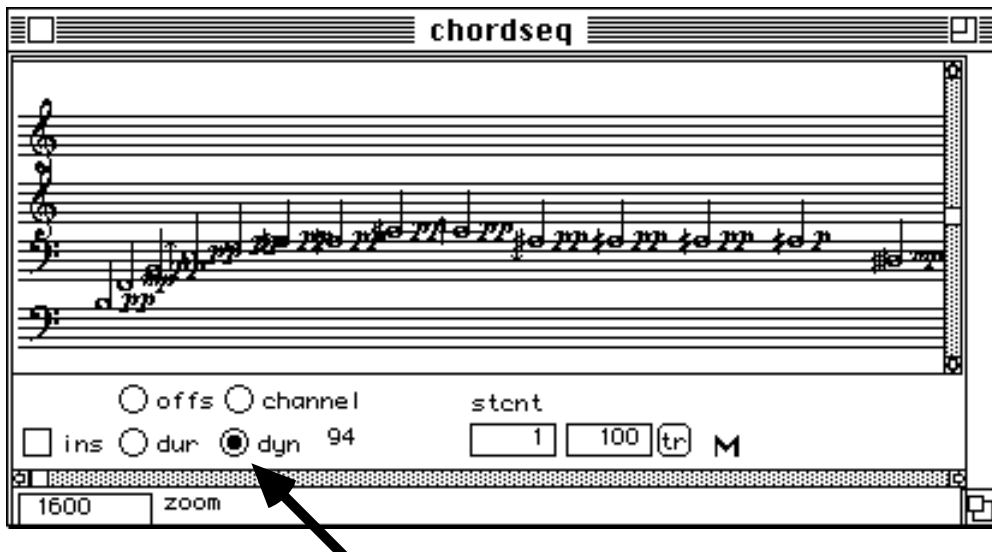
- A: The absin box (a) corresponds to the first input of the abstraction. It receives the break-point function coming out of the bpf box (C).
- B: The module in (b) receives the value coming out of the box out (A) (in the main patch).

- C: The bpf-sample samples the curve you have entered in bpf box (C). The number of samples corresponds to the value coming out of the in box (b), defined in numbox (A), in the main patch.
- D: The absin box (d) corresponds to the second input of the abstraction. The value entered here will correspond to the lower note of your sequence of notes.
- E: The absin box (e) corresponds to the third input of the abstraction. The value entered here will correspond to the upper note of your sequence of notes.
- F: The module g-scaling (f) scales the values coming out of bpf-sample (C) (in the main patch), between the values entered in the absin boxes (d) and (e).
- G: The absout box is the output of the abstraction.
- Back to the main patch...
- E: Open and edit a curve in the bpf box (E). This break-point function will be converted into a sequence of distances between each note by the abstraction (F). In PatchWork, distances are expressed in hundredths of a second: a value of 100 corresponds to one second between the attacks of each note. Thus, a value of 50 corresponds to half a second, 25 to a quarter of a second, etc...
- F: Abstraction. This is the same as abstraction (D). Enter the two values between which you want the curve (E) to be scaled. In the second input, enter the value that will be the minimum distance between any two notes. In the third input, enter a value corresponding to the maximum distance between any two notes.
- G: Open and edit a curve in the bpf box (G). This break-point function will be converted into a sequence of durations by the abstraction (H). As before, 100 corresponds to one second, 150 to a second and a half, 200 to two seconds, etc...
- H: Abstraction. This is the same as abstraction (D). This abstraction scales the durations of all of the notes between the two values entered in the second and third inputs.
- I: Open and edit a curve in the bpf box (I). This break-point function will be converted into a sequence of midi velocities by the abstraction (H). Value 1 corresponds to the minimal velocity and 127 to the maximal one.
- L: Abstraction. This is the same as abstraction (D). It scales the midi velocities between the two values entered in its second and third inputs.
- M: Evaluate and open the chordseq box to see the result.

If you want to see the durations press the button "durs" in the chordseq window.



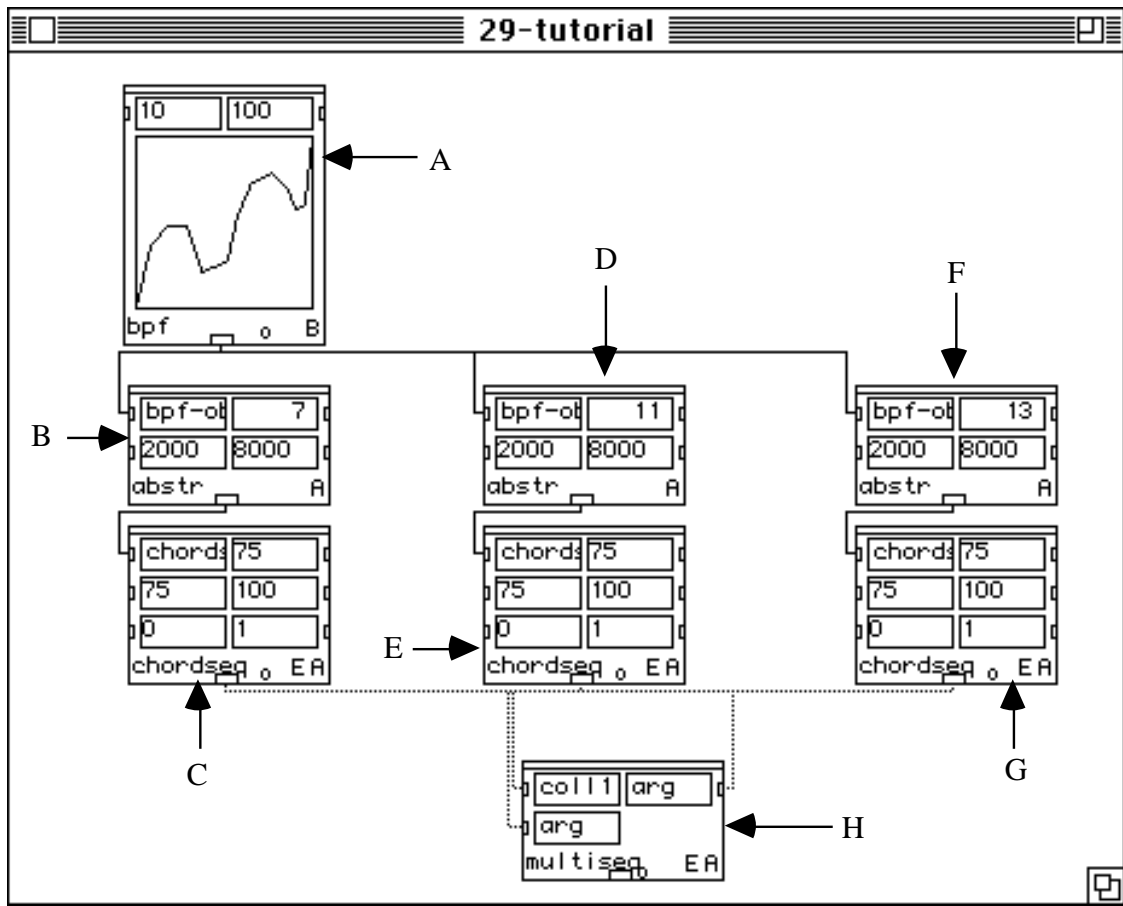
If you want to display the midi velocities, press the button "dyn" in the chordseq window.



Now, try changing either the break-point functions (C, E, G and I), the value in the numbox module (A) or the values of the abstractions inputs. Re-evaluate the chordseq box and look at the new results.

Tutorial 29

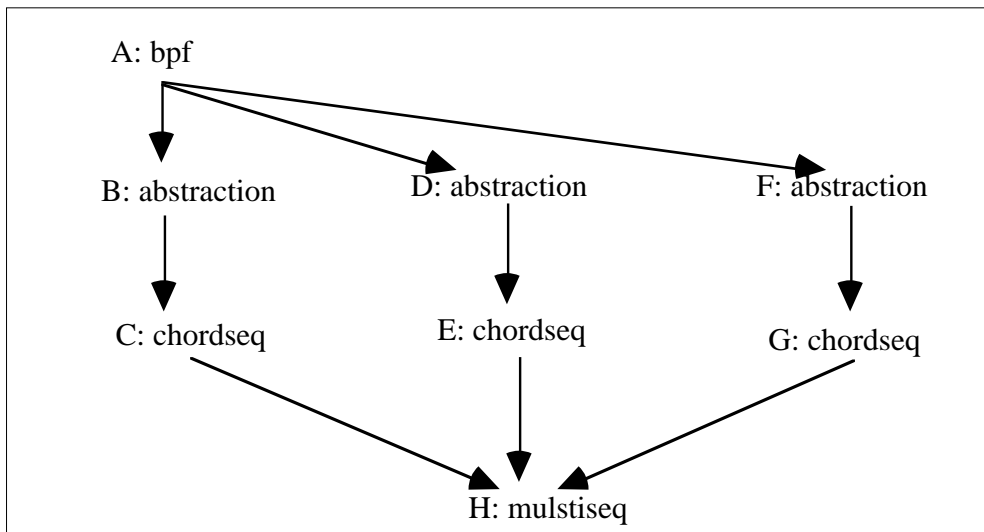
Musical notation: proportional notation of multiple note sequences.



Description

This patch transforms a break-point function (A) into three different sequences of notes and displays them in a single window using proportional musical notation.

Patch structure

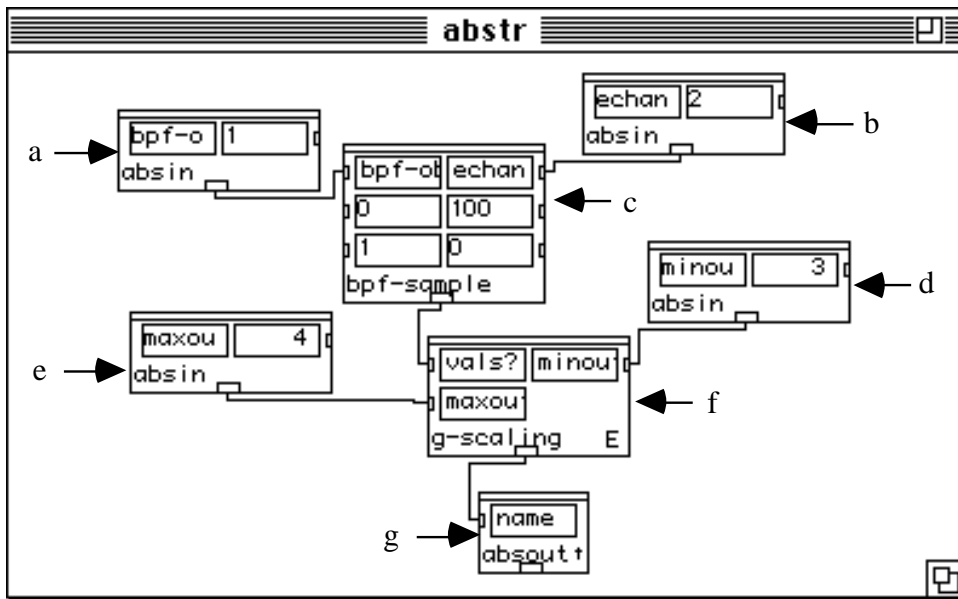


Boxes used

absin, absout, bpf, chordseq, bpf-sample, g-scaling.

Utilisation

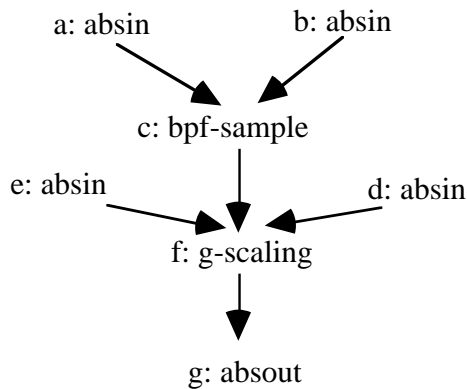
- A: Open the bpf box (A) and edit the curve. The break-point function you edit will be sampled into three different sequences of notes.
- B: Abstraction



Abstraction Description

This abstraction receives a break-point function in the first input. Then, it samples the curve into a given number of samples, defined in the second input. Finally it scales the samples between the two values entered in its third and fourth inputs.

Abstraction structure



Open the abstraction by selecting "open" in its menu, and observe the internal structure

Utilisation

- A: The absin box (a) corresponds to the first input of the abstraction. It receives the break-point function coming out of the bpf box (A).
- B: The absin box (b) corresponds to the second input of the abstraction. In this input, enter the number of samples by which you wish to sample the break-point function.

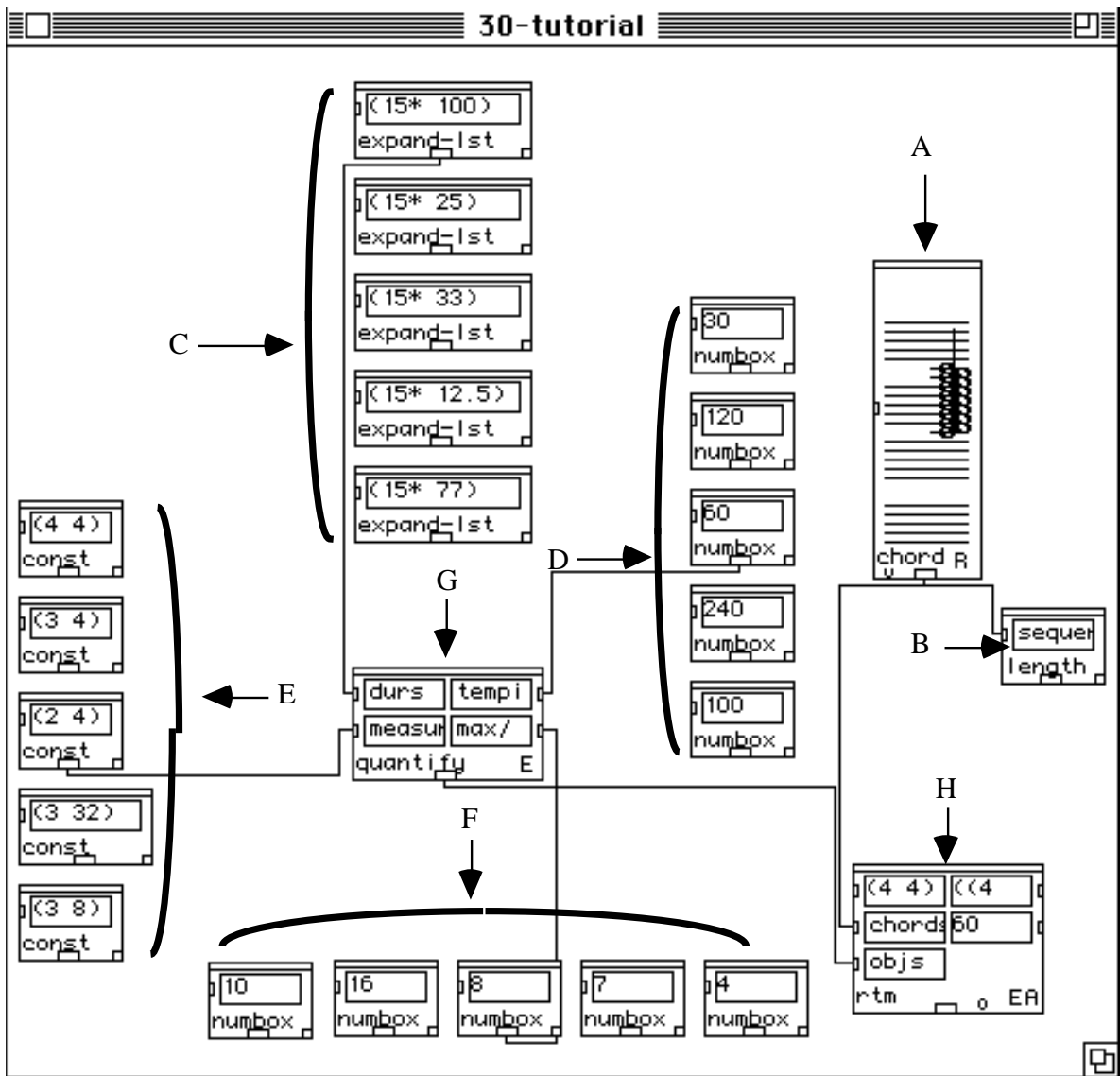
- C: The bpf-sample samples the curve you have entered in bpf box (C). The number of samples corresponds to the value coming out of the absin box (b).
- D: The absin box (d) corresponds to the third input of the abstraction. The value entered here will correspond to the lowest note in your sequence of notes.
- E: The absin box (e) corresponds to the fourth input of the abstraction. The value entered here will correspond to the highest note in your sequence of notes.
- F: The module g-scaling (f) scales the values coming out of bpf-sample (A), between the two values entered in the absin boxes (d) and (e).
- G: The absout box is the output of the abstraction.

Back to the main patch...

- C: Evaluate the chordseq (C) to see the first sequence of notes.
- D: Abstraction. This is the same as abstraction (B).
- E: Evaluate the chordseq (E) to see the second sequence of notes.
- F: Abstraction. This is the same as abstraction (B).
- G: Evaluate the chordseq (E) to see the third sequence of notes.
- H: Evaluate the multiseq (H) box and open it to see the three sequences of notes superimposed. Now close the multiseq window, change either the curve in the bpf box (A), the number of samples in the second input of each abstraction, or the two scaling values in the third and fourth inputs of each abstraction, then re-evaluate the multiseq and look at the new result.

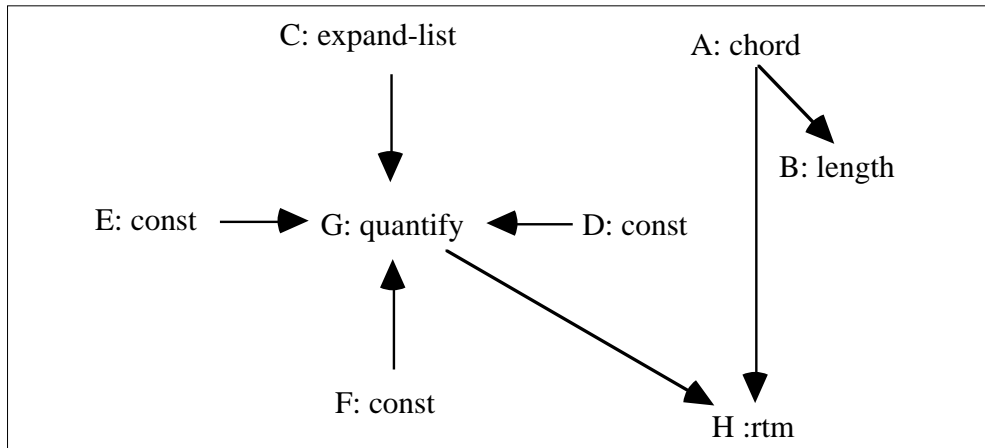
Tutorial 30

Musical notation: melodic and rhythmic notation.



Description

This patch displays a sequence of notes (A) using musical notation, for which you can define different durations, tempi, measures and rhythmic approximations.

**Boxes used**

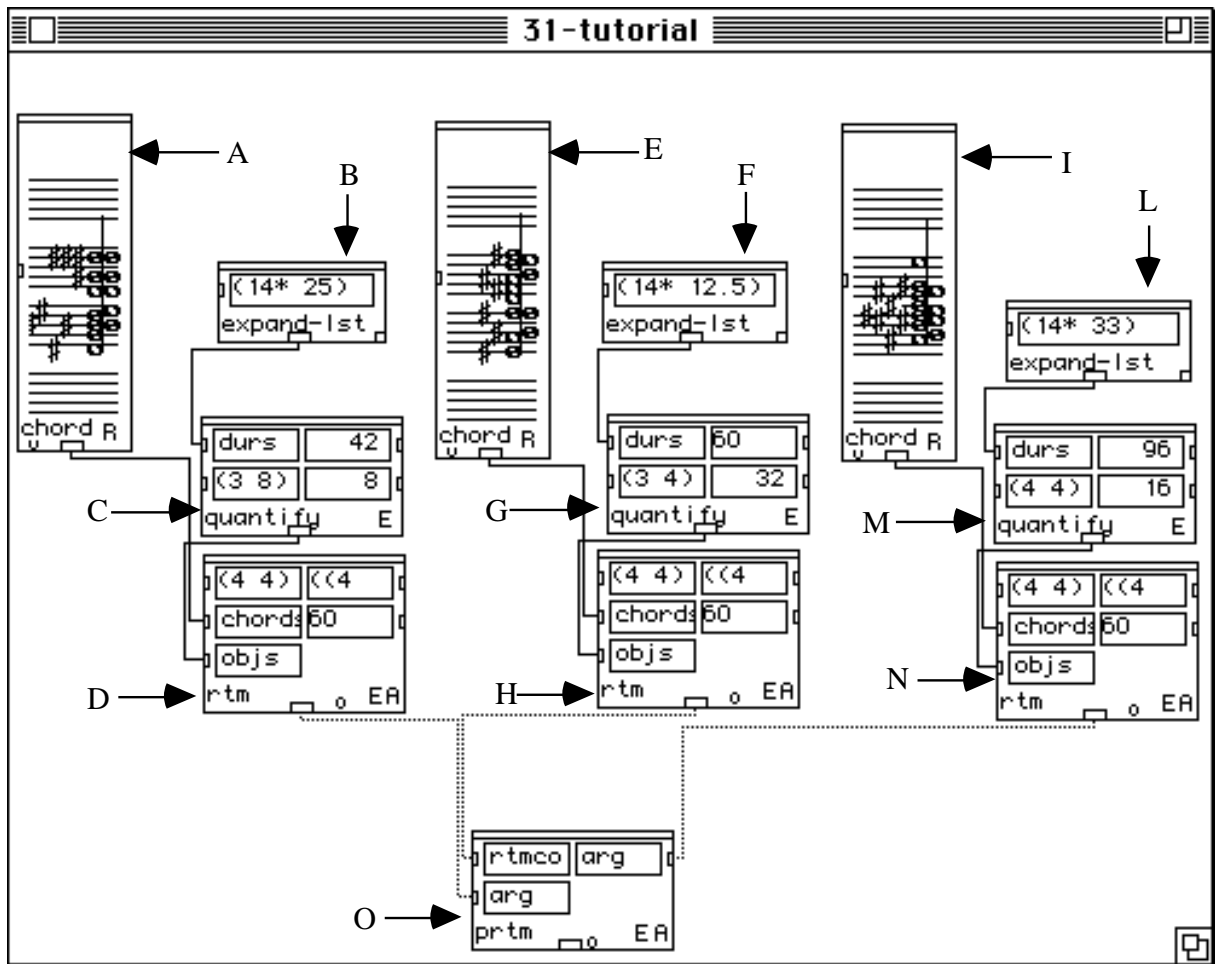
chord, const, expand-list, length, quantify, rtm.

Utilisation

- A: Open the chord (A) and edit a sequence of notes in the "arpeggio" mode. Then close the chord window and set it to the "Reorder" mode.
- B: Evaluate the length box (B) to see the length of the sequences you have entered.
- C: Choose one of these expand-list modules (C) and connect it to the "durs" input of the quantify box (G). In this example, the expand-list box (C) sends out the value 100 fifteen (15*) times.
- D: Choose one of these numbox modules (D) and connect it to the "tempi" input of the quantify box (G).
- E: Choose one of these const modules (E) and connect it to the "measures" input of the quantify box (G).
- F: Choose one of these numbox modules (E) and connect it to the "max/" input of the quantify box (G).
- G: The module quantify quantizes the list of values entered in the chosen expand-list module (C) into a given metric measure defined in the chosen const module (E), with a tempo entered in the chosen numbox module (D) and with a maximum beat division defined in the chosen numbox module (E).
- H: Evaluate the rtm box (H) in order to display the results. Now try changing the connections between the quantify box and the expand-list modules (C), the const modules (E), the numbox modules (D) and the numbox modules (E). Then, change the values in the different boxes to create your own rhythms, measures, tempi and rhythmic approximations, re-evaluate the rtm box and view the new results.

Tutorial 31

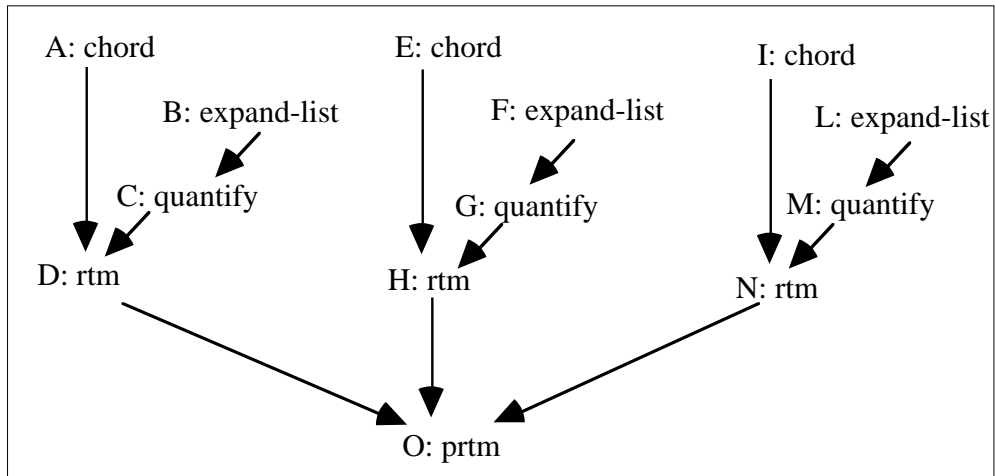
Musical notation: notation of multiple rhythmical sequences.



Description

This patch creates three different rhythmical structures with three different sequences of notes and displays them in a single window using rhythmical musical notation.

Patch structure



Boxes used

chord, expand-list, prtm, quantify, rtm.

Utilisation

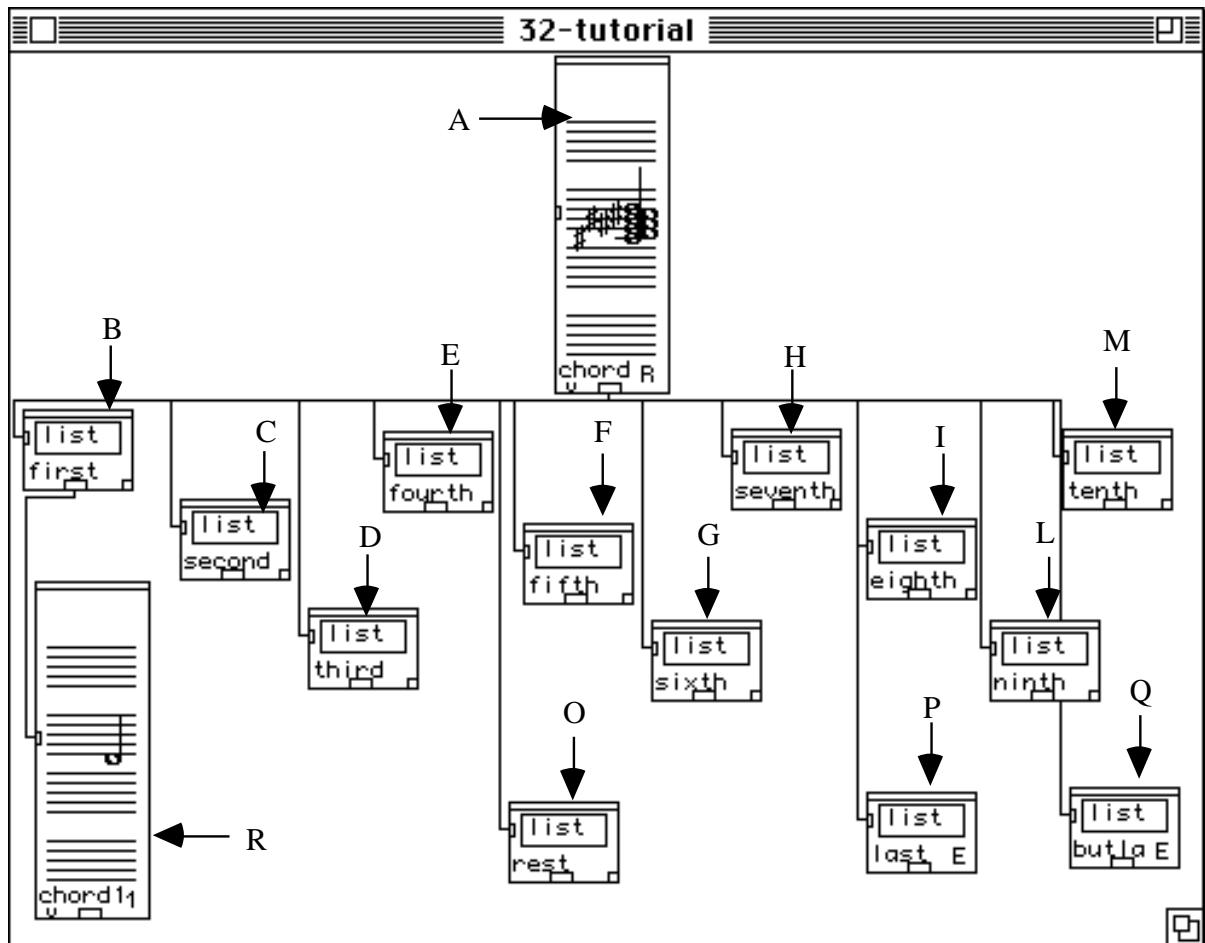
- A: Open the chord (A) and edit a sequence of notes in the "arpeggio" mode. Then close the chord window and set it to the "Reorder" mode.
- B: In the expand-list (B) enter a rhythmical structure of the same number of elements as the number of notes entered in the chord box (A).
- C: The module quantify (C) quantizes the list of values entered in the expand-list module (B) with a tempo entered in its second input, into a given metric measure defined in its third input, and with a maximum beat division defined in its fourth input.
- D: Evaluate the rtm box (D) in order to display the results of the first rhythmical sequence.
- E: Open the chord (E) and edit a sequence of notes as before.
- F: In the expand-list (F) enter a rhythmical structure as before.
- G: The module quantify (G) quantizes the list of values entered in the expand-list module (F) with a tempo entered in its second input, into a given metric measure defined in its third input, and with a maximum beat division defined in its fourth input.
- H: Evaluate the rtm box (H) in order to display the results of the first rhythmical sequence.
- I: Open the chord (I) and edit a sequence of notes as before.
- L: In the expand-list (L) enter a rhythmical structure as before.
- M: The module quantify (M) quantizes the list of values entered in the expand-list module (L) with a tempo entered in its second input, into a given metric measure defined in its third input, and with a maximum beat division defined in its fourth input.
- N: Evaluate the rtm box (N) in order to display the results of the first rhythmical sequence.
- O: Evaluate the prtm (poly-rhythm) (O) and open it to see the three different musical structures into a single window. Now change either the sequences of notes

(A, E and I), the rhythmical structures in the three expand-list boxes, or the values entered in the three quantify modules, then re-evaluate the prtm box to see the new result.

See also in your Reference manual: prtm.

Tutorial 32

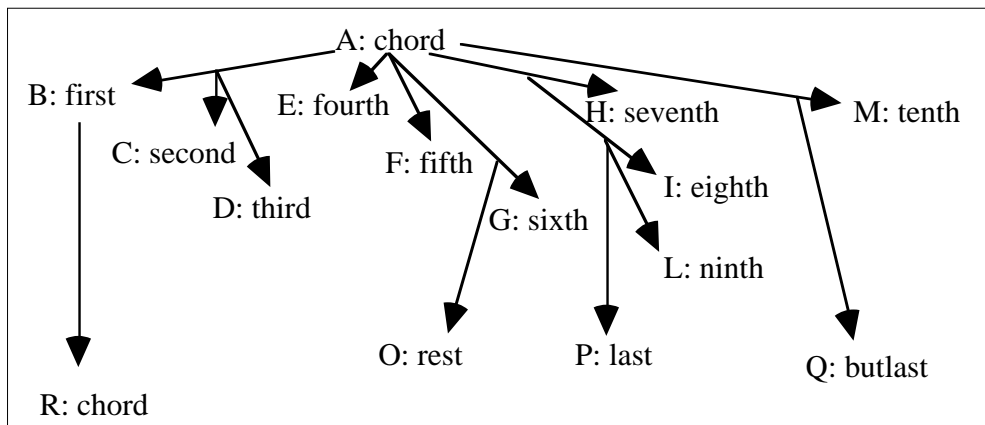
Using Lisp functions: extraction of one or more elements in a sequence of notes.



Description

This example shows how to invoke and use some Common Lisp functions in order to extract one or more elements of a sequence of notes.

Patch structure

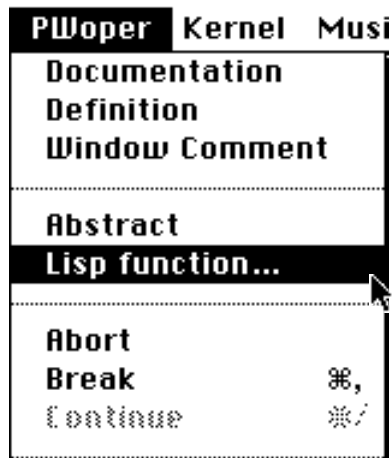


Boxes used

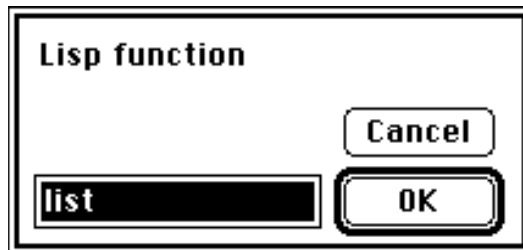
butlast, chord, eighth, first, fourth, fifth, last ninth, second, rest, sixth, seventh, tenth, third.

Utilisation

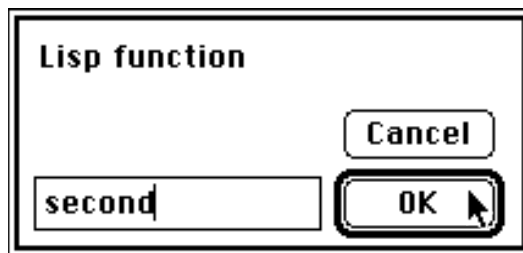
- A: Open the chord (A) and edit a sequence of ten notes.
- B: The first box (B) takes the first element of the entered notes in the chord box (A). Connect this module to the chord box (Q), then evaluate the module chord box (Q) to see the result.
- C: The second box (C) takes the second element of the entered notes in the chord box (A). This function doesn't belong to the Kernel menu or the Music menu. To invoke this box select Lisp function... in the PWoper menu as shown:



A dialog box will appear on your screen.



Now enter the name of the box you want to invoke and press "Return" on your keyboard.



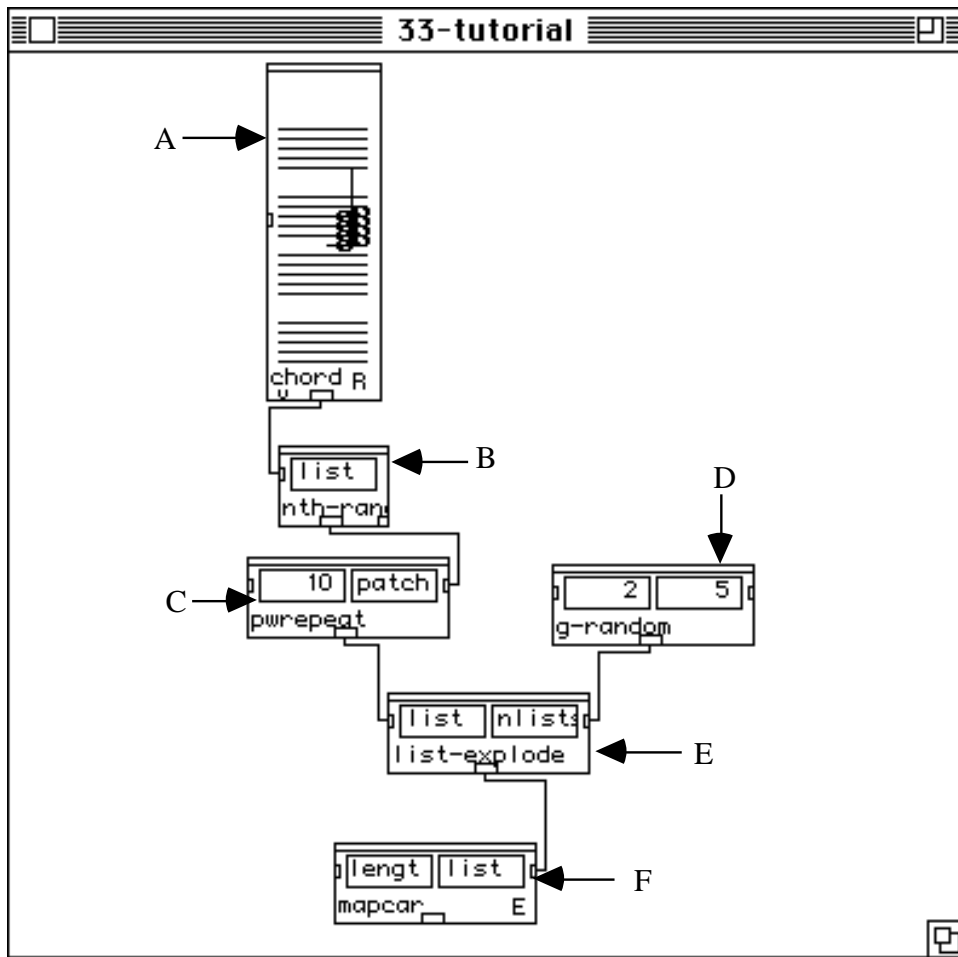
Now connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result.

- D: The third box (D) takes the third element of the entered notes in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).
- E: The fourth box (E) takes the fourth element of the entered notes in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).
- F: The fifth box (F) takes the fifth element of the entered notes in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).

- G: The sixth box (G) takes the sixth element of the entered notes in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).
- H: The seventh box (H) takes the seventh element of the entered notes in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).
- I: The eighth box (I) takes the eighth element of the entered notes in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).
- L: The ninth box (L) takes the ninth element of the entered notes in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).
- M: The tenth box (B) takes the tenth element of the entered notes in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).
- N: The rest box (N) takes all the elements of the sequence of notes entered in the chord box (A) except the first element. Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result.
- O: The last box (O) takes the last element of the sequence of notes entered in the chord box (A). Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result. If you want to invoke this module follow the same instructions as before (C).
- P: The butlast box (P) takes all the elements of the sequence of notes entered in the chord box (A) except the last element. Connect this module with the chord box (Q), then evaluate the module chord box (Q) to see the result.

Tutorial 33

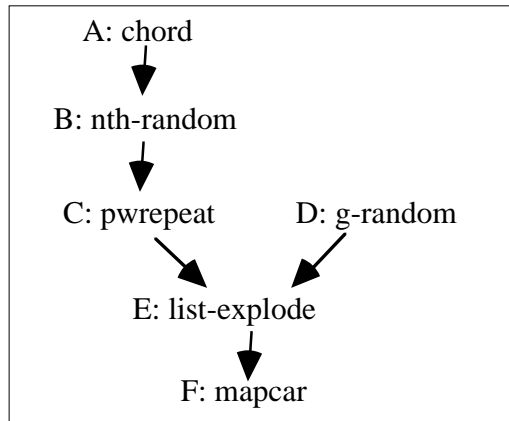
Using Lisp functions: applying a Common Lisp function to each element of a sequence.



Description

This example uses a Common Lisp function that is able to apply a defined function to all the elements in a sequence of notes.

Patch structure



Boxes used

chord, g-random, list-explode, mapcar, nth-random, pwrepeat.

Utilisation

- A: Open the chord (A) and edit a sequence of notes.
- B: The nth-random box (B) takes randomly a note of the sequence entered in the chord box (A).
- C: The pwrepeat box (C) repeats the random extraction 10 times.
- D: The g-random box (D) calculates a single random value between two limiting values (in this example between 2 and 5).
- E: The module list-explode box (E) divides the list of notes coming out of the pwrepeat box (C) into a number of groups coming out of g-random box (D).
- F: The module mapcar (F) represents a Common Lisp function. This box applies the function entered in its left input to each element in its right input. In this example, in the left input there is the function length and in the right one there are the subgroups generated by the list-explode box (E). That is to say that the mapcar box (F) will calculate the length of each subgroup generated by the list-explode (E). Evaluate this box many times and see the result in the Listener window.



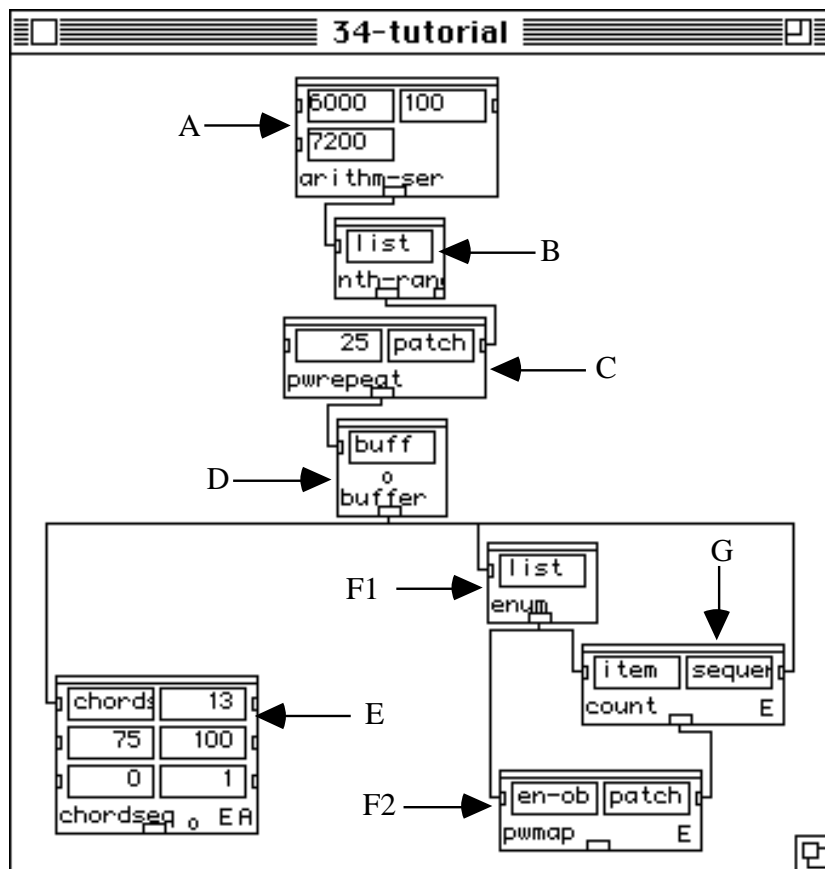
Now, enter the function list in the left input of the module mapcar.



This time, the mapcar module will add a pair of parenthesis to any list of elements coming out of the list-explode box (E). Re-evaluate the mapcar box many times to see the result which will always appear in the Listener window.

Tutorial 34

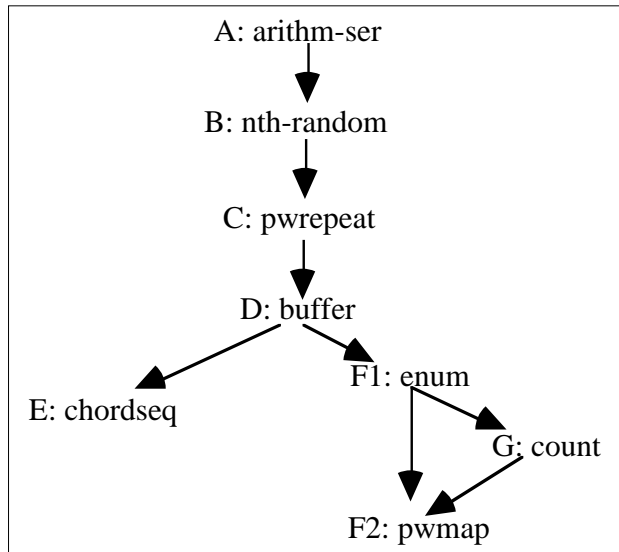
Using Lisp functions: counting notes



Description

This patch uses a Common Lisp function `count` to count how many times a note appears in a sequence of notes.

Patch structure



Boxes used

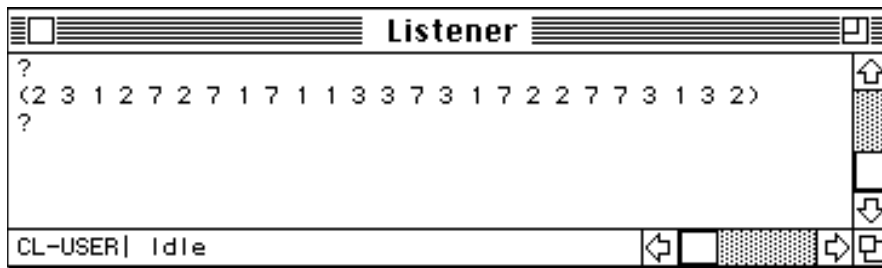
arithm-ser, buffer, chordseq, count, nth-random, pwmap (enum), pwrepeat.

Utilisation

- A: The module arithm-ser (A) creates an arithmetic series starting from value 4800, ending with the value 7500, with a step of 100 (a semitone).
- B: The nth-random box (B) takes randomly an element of the sequence created by the arithm-ser box (A).
- C: The pwrepeat box (C) makes the nth-random (B) repeat the same process many times (in this example 25 times).
- D: The buffer box (D) stores the result of this part of the patch's calculation. Evaluate the buffer box and then close the buffer by clicking on the "o" in the centre of this box.

In this state, the buffer box has stored the data coming out of the pwrepeat box (C).

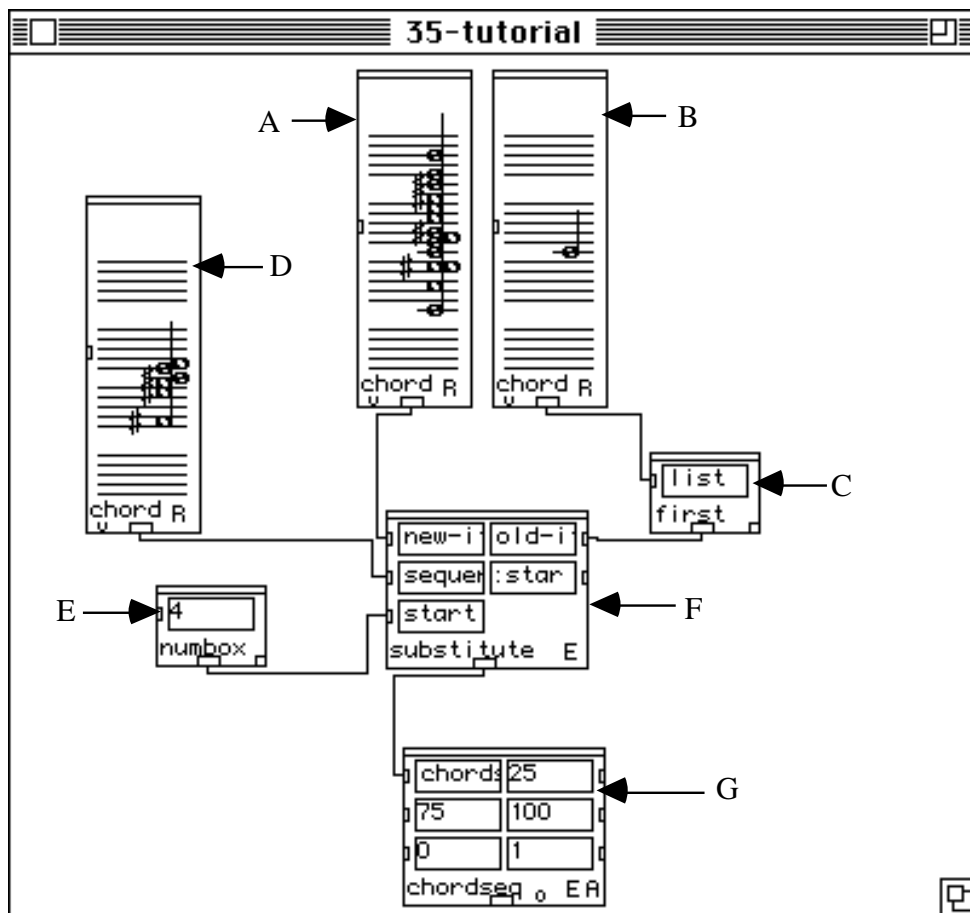
- E: Evaluate the chordseq box (E) to see the result of this part of the patch.
- F1: The module enum (F1), connected with the module pwmap (F2), takes the elements that come out of the buffer box (D), one by one.
- G: To invoke the Common Lisp function count select Lisp function... in the PWoper menu. This function counts how many times each element of the random series appears in the same series.
- F2: The module pwmap (F2) allows us to repeat the same process for all the elements coming out of the buffer box (D). Evaluate this box and see how many times a note appears in the hole sequence.



Now open, re-evaluate and close the buffer box (D), then re-evaluate the chordseq to see the new random sequence of notes and re-evaluate the pwmap box (F2) to see the new result.

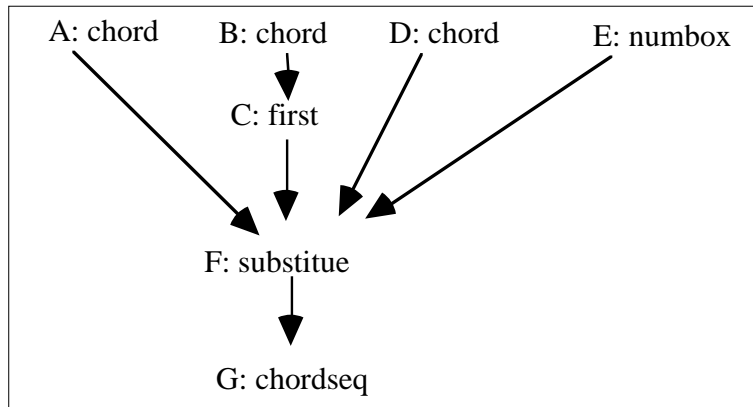
Tutorial 35

Using Lisp functions: substituting a chord for a note in a sequence of notes.



Description

This patch substitutes a defined note with a chord in a series of notes.



Boxes used

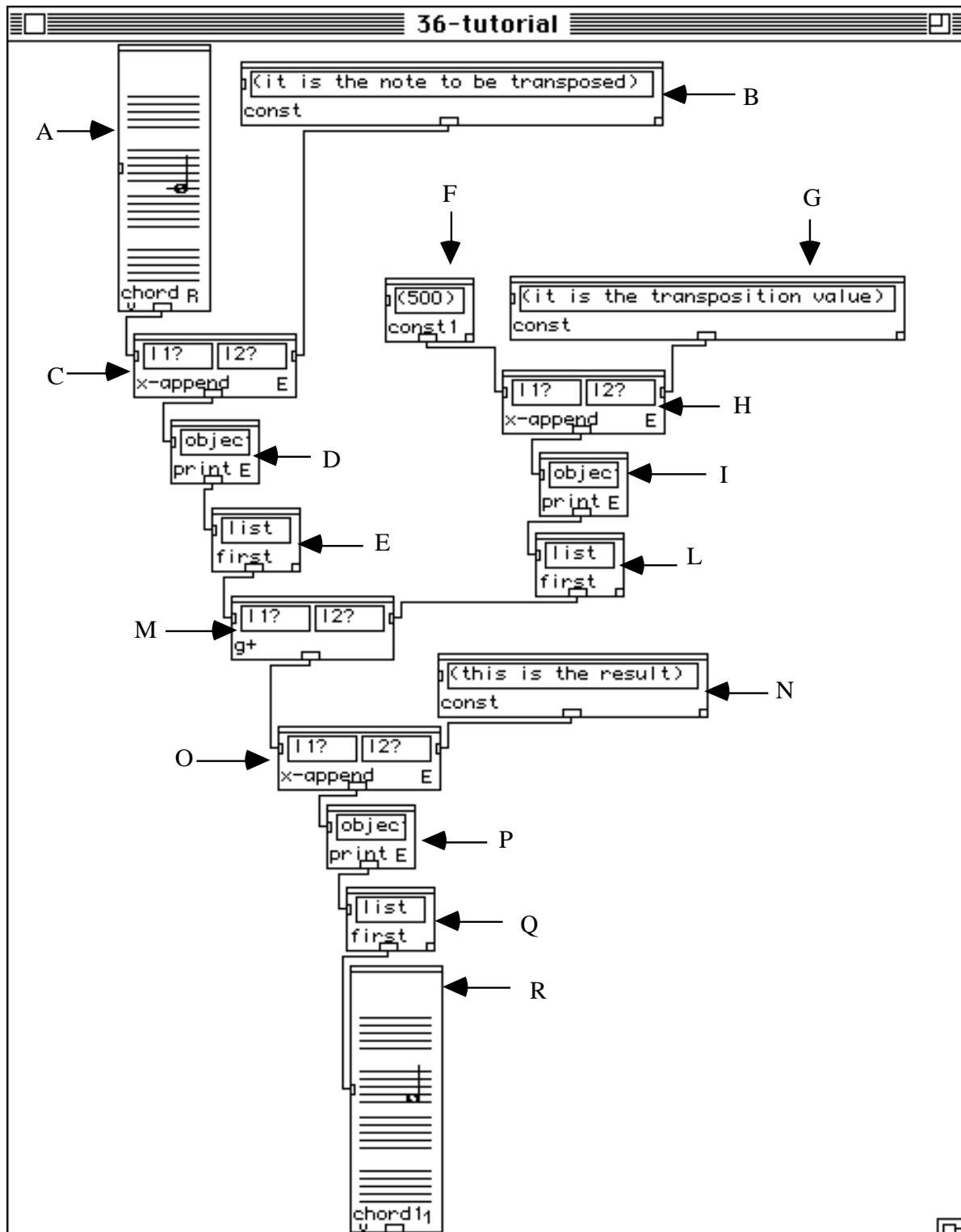
chord, chordseq, first, substitute, numbox.

Utilisation

- A: Open the chord (A) and edit a sequence of notes in the "arpeggio" mode. Then close the chord window and set it to the "Reorder" mode.
- B: Open the chord box (B) and enter the note you want to substitute.
- C: The first box (C) takes the first element of the list coming out of chord box (B).
- D: Open the chord box (C) and enter the chord you want to replace in the same place of the note defined in the chord box (B).
- E: Enter in the numbox module (E) where you want the substitution to start in your sequence. That is to say that if you enter the number 1, the substitution starts from the first element of the sequence, if 2, from the second, if 3, from the third, etc.
- F: The substitute box (F) is a Common Lisp function and to invoke it you have to use the PWoper menu, as before. This module substitutes the note defined in the chord box (B) with the chord entered in the chord box (C), starting from the index entered in the numbox (E).
- G: Evaluate this chordseq box (G) to see the result. Then try changing the value in the numbox (E) and re-evaluate the chordseq box (G) and look at the new result.

Tutorial 36

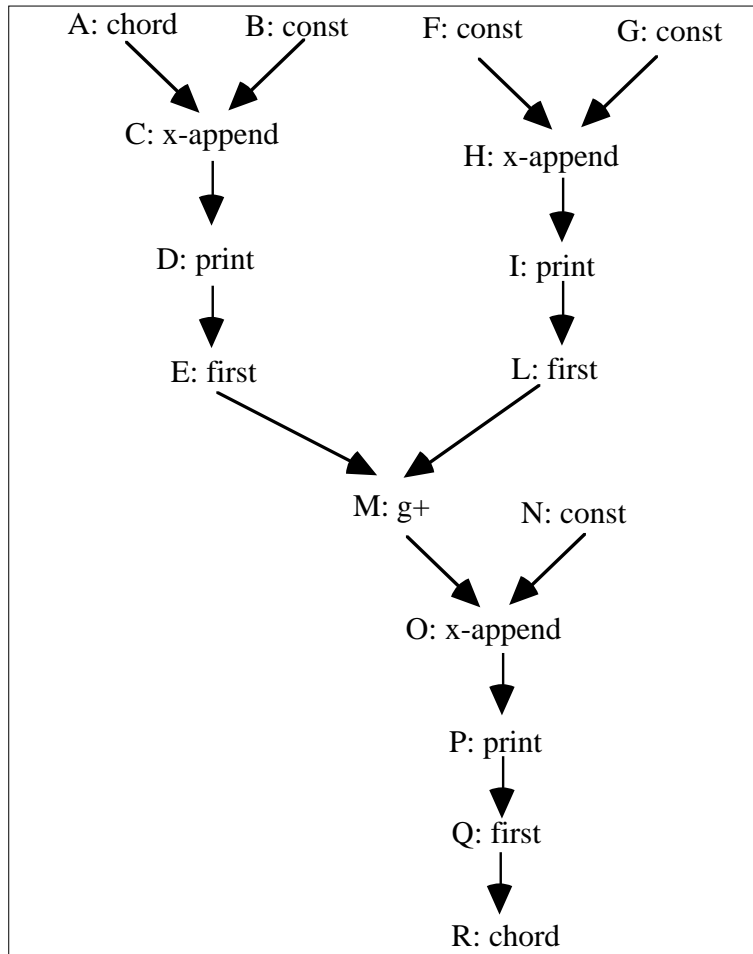
Using Lisp functions: auto-documented transposition.



Description

This example makes a transposition of one note and it displays previously prepared documentation in the Listener window.

Patch structure



Boxes used

chord, const, first, g+, print, x-append.

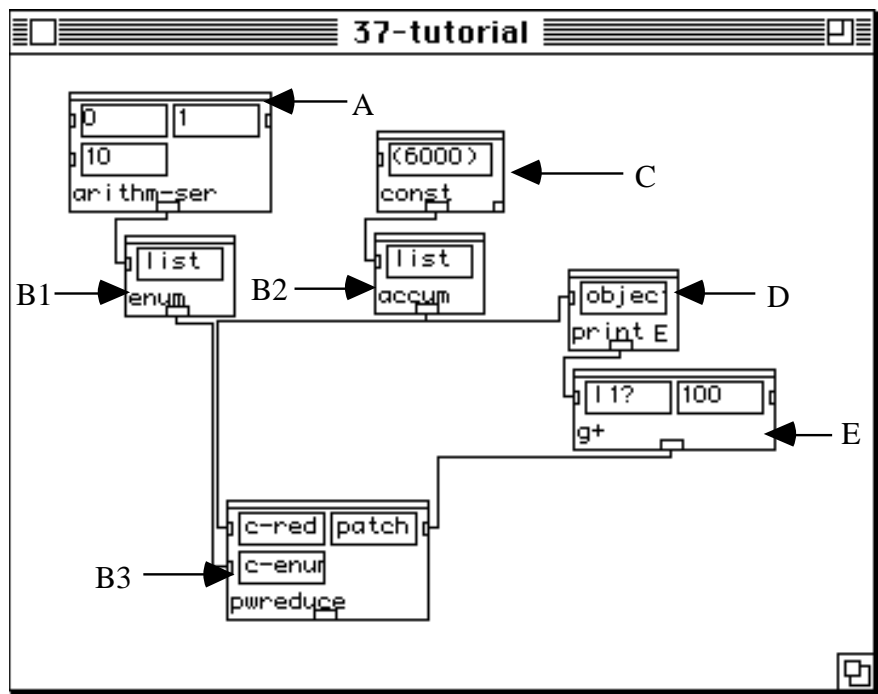
Utilisation

- A: Open the chord (A) and edit a sequence of notes.
- B: This const box (B) contains the first message that will be printed in the Listener window.
- C: The x-append box (C) puts together the value entered in the cord box (A) and the message written in the const box (B).
- D: The print box (D) is a Common Lisp function. To invoke it use the Lisp function... in the PWoper menu. This box prints out all elements entering in its input, in the Listener window.
- E: The first box (E) takes the first element coming out of the print box (D).
- F: Enter in the const box (F) the transposition value.

- G: This const box (B) contains the second message that will be printed in the Listener window.
- H: The x-append box (H) puts together the value entered in the cord box (F) and the message written in the const box (G).
- I: The print box (I) prints out all elements (entering in its input) in the Listener window.
- L: The first box (E) takes the first element coming out of the print box (D).
- M: The g+ module (M) makes the sum between the note entered in the chord box (A) and the transposition value defined in the const box (F).
- N: This const box (N) contains the third message that will be printed in the Listener window.
- O: The x-append box (O) puts together the value coming out of the g+ module (M) and the message written in the const box (N).
- P: The print box (I) prints out all elements entering in its input, in the Listener window.
- Q: The first box (Q) takes the first element coming out of the print box (P).
- R: Evaluate this chord box (R) and see the commented result in the Listener window.

Tutorial 37

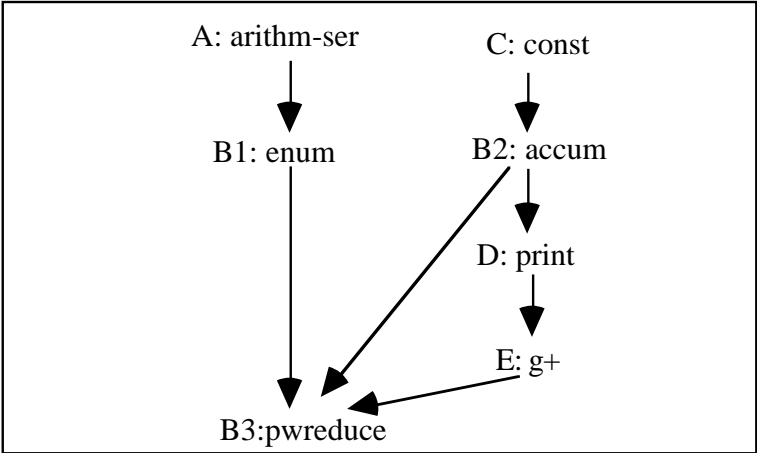
Recursion: explaining recursion.



Description

This example creates a series of notes using a recursive function.

Patch structure



Boxes used

arithm-ser, const, print, g+, pwreduce (enum, accum).

What is recursion?

A recursive function is a particular kind of function that takes as input its own output.

That is to say, a recursive function needs of two things: a number that indicates how many times it must repeat the recursive process on its result and the first value from which it must start. For example, let's consider a recursive function that adds 1 to its input and repeats this operation 10 times. If the first input that it receives is equal to 0, this function will produce:

$$0 + 1 = 1.$$

Then it takes this result as the new input and adds 1 to it. That is to say that the function will produce:

$$1 \text{ (the first result)} + 1 = 2.$$

Then it takes again the new result as the new input and adds it the value 1:

$$2 \text{ (the second result)} + 1 = 3.$$

This function repeats this process ten times and the last evaluation will return the value 10.

Utilisation

- A: The module arithm-ser (A) creates an arithmetic series starting from value 0, ending with the value 11, with a step of 1. (The result of this series will be: 0 1 2 3 4 5 6 7 8 9 10, and the number of elements is equal to 11).
- B1: The module enum is always connected with the module pwreduce. When you invoke the module pwreduce it appears, in the patch window, already connected with a second module enum and a third module accum. These three modules together are able to make a recursive function. As we have already seen for the pwmap box, enum box (B1) takes the elements one by one in its input. In this example the values entering in the enum box correspond to how many times the function must repeat the recursive process.
- C: Enter in the const box (C) the value that will initialise the accum box to which it is connected. That is to say, this value will be the value from which the recursive function will start.
- B2: The accum box (B2) accumulates the results coming out of a evaluation to utilise it for the next evaluation.
- D: The print box (D) is a Common Lisp function. To invoke it you have to select Lisp function... in the PWoper menu¹. This function prints in the Listener window the different steps of the recursive function process.
- E: The addition box, g+ (E), transposes the value stored in the accum box (B2) by adding 100.
- B3: The module pwreduce (B3) is used to close the recursive process. That is to say, the enum box (B1) gives the number of times the operation must repeat the process, the accum box (B2) accumulates the previous result and utilises it, as input, to calculate the next one. The pwreduce box (B3) allows the enum box (B1) and the accum box (B2) to repeat the same process for the number of time corresponding to the number of elements entering in the enum box (B1). In this patch the enum box (B1) receives the value coming out of the module arithm-ser (A). The accum box (B2) it's initialised to the value 6000. In this case the recursive function adds 100 to the first value (6000) stored in the accum box and repeats this process 11 times (11 is the number of values coming out of the module arithm-ser (A)). Evaluate the pwreduce box (B3) to see all the steps of the

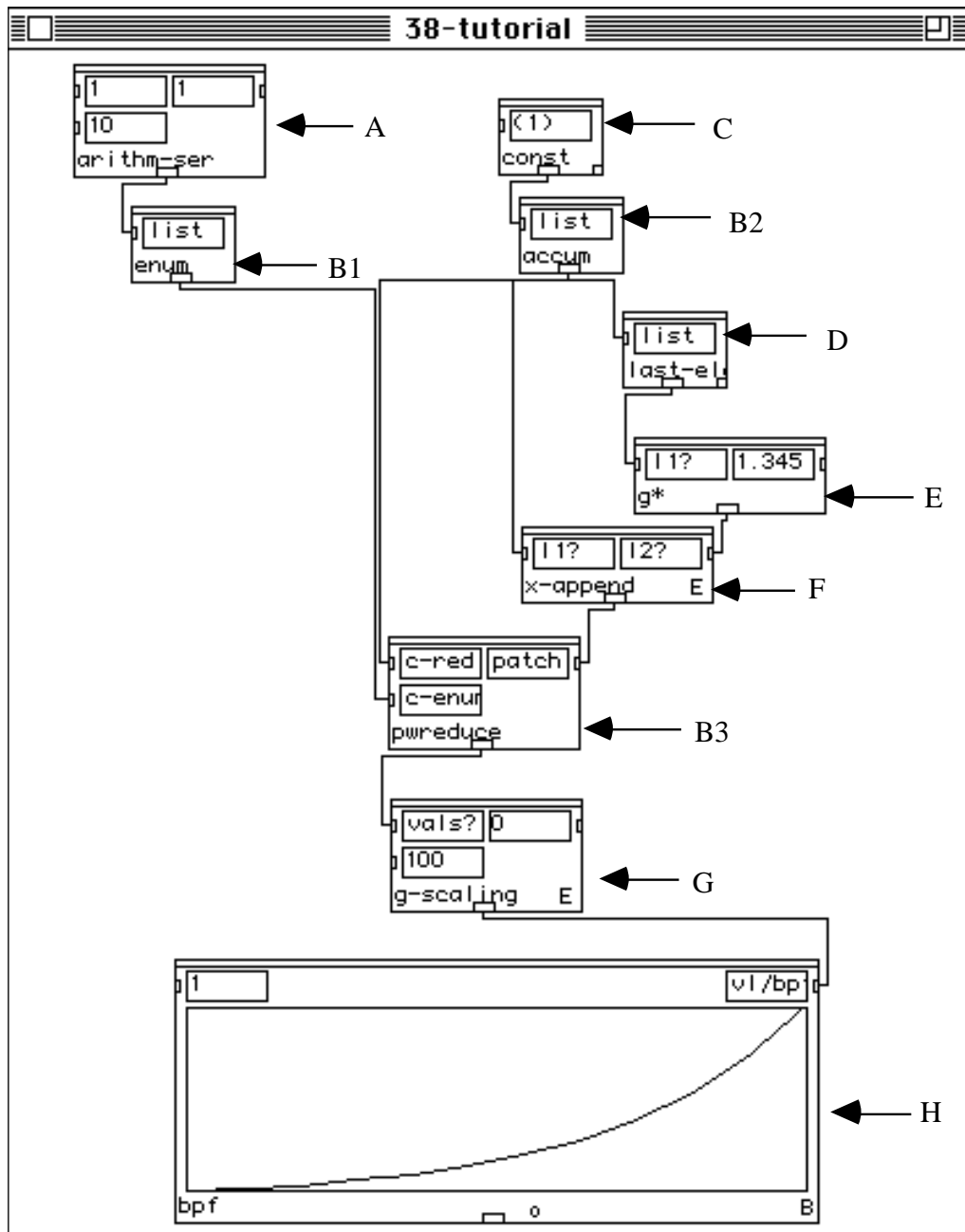
1. See tutorial 32, paragraph C.

recursion, in the Listener window. Then try changing the value in the const box (C) and the "end" value of the arithm-ser box (A), re-evaluate the pwreduce box (B3) and see again the result.

See also pwreduce in your Reference manual.

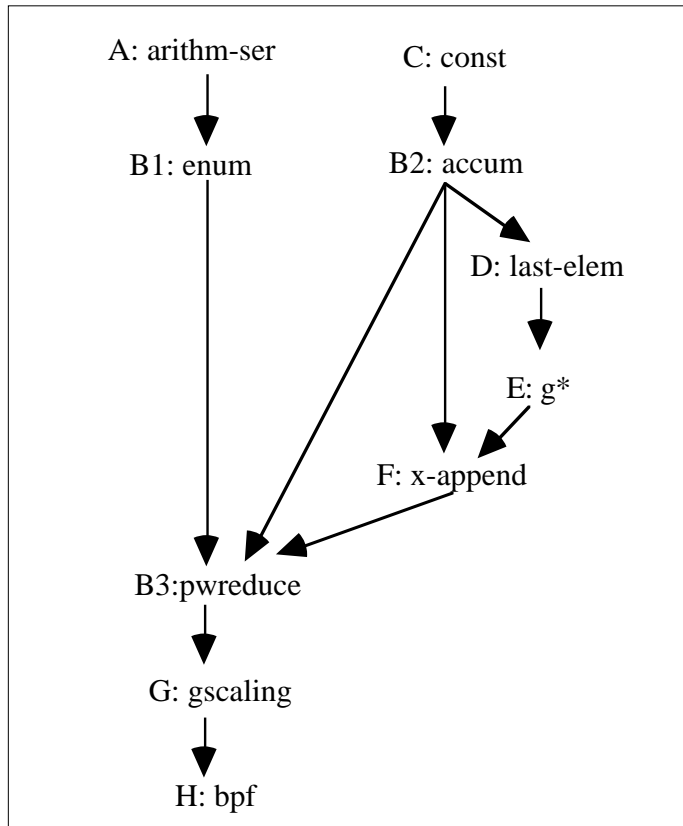
Tutorial 38

Recursion: creating a geometric curve.



Description

This example creates a break-point function using a recursive function.

**Boxes used**

arithm-ser, bpf, const, g*, g-scaling, last-elem, pwreduce (enum, accum).

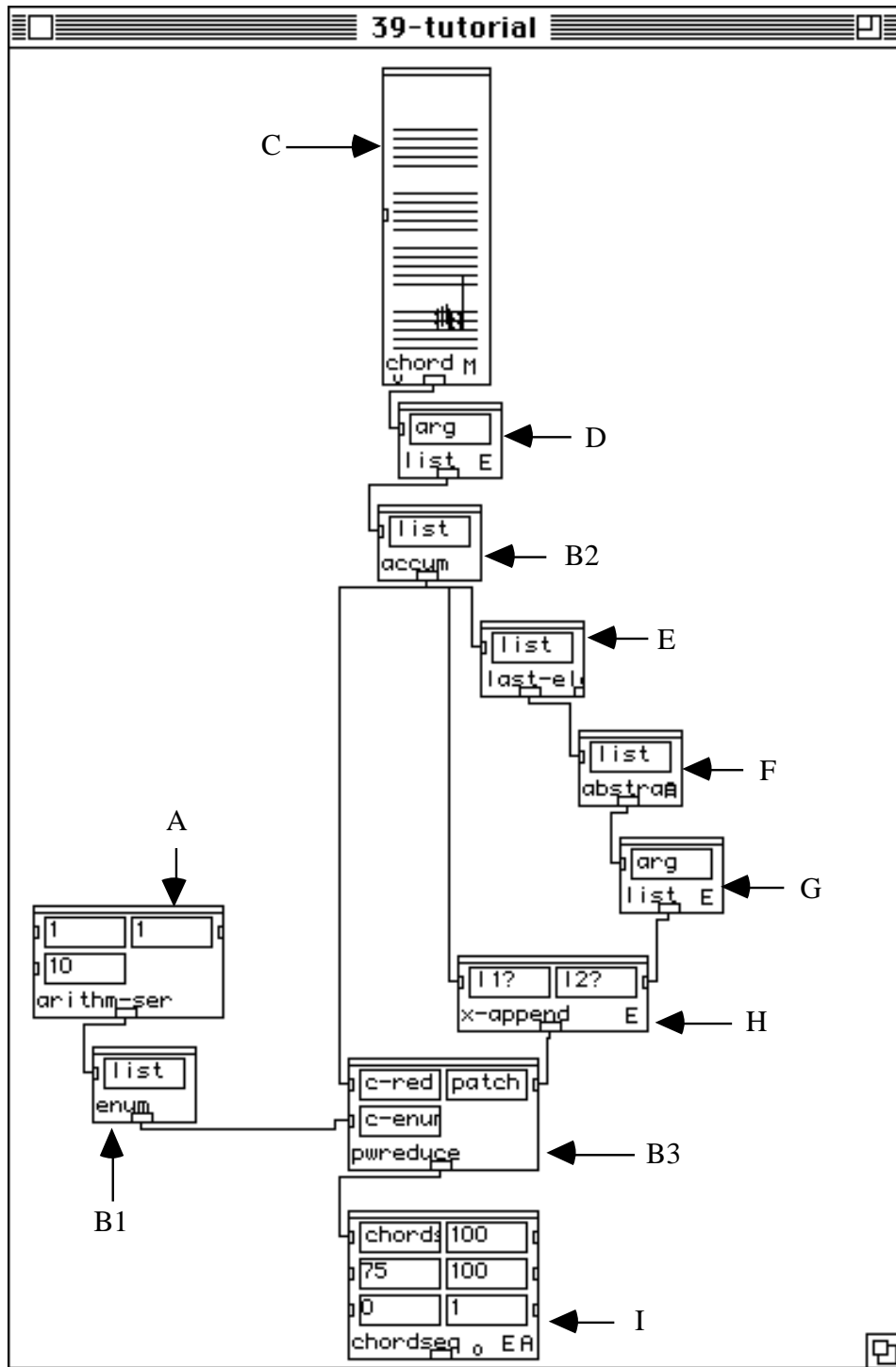
Utilisation

- A: The module arithm-ser (A) creates an arithmetic series starting from value 0, ending with the value 10, with a step of 1. (The result of this series will be: 1 2 3 4 5 6 7 8 9 10, and the number of elements is equal to 10)
- B1: The module enum (B1) is connected with the module pwreduce. It takes the elements coming out of the arithm-ser box (A), one by one.
- C: Enter in the const box (C) the value that will initialise the accum box to which it is connected. That is to say, this number will be the value from which the recursive function will start.
- B2: The accum box (B2) accumulates the results coming out of the previous evaluation to use it for the next one.
- D: The last-elem box (D) takes the last value stored in the accum box (B2).
- E: The g* box (E) makes the multiplication between the value coming out of the last-elem box (D) and the value entered in its right input.
- F: The x-append box appends together the values entering in its inputs. In this example, it appends the list stored in the accum box (B2) to the value coming out of the g* box (E). In this way, the next iteration of the recursive function will take this last element as the new input.
- B3: The module pwreduce (B3) closes the recursive process.

- G: In this patch, the module g-scaling scales the values coming out of the pwreduce box (B3) between 0 and 100.
- H: Evaluate the bpf box (H) and look at the obtained curve. Then change the value of the multiplication, in the right input of the g^* box (G), re-evaluate the bpf box (H) too see the new result.
- See also in your Reference manual: last-elem.

Tutorial 39

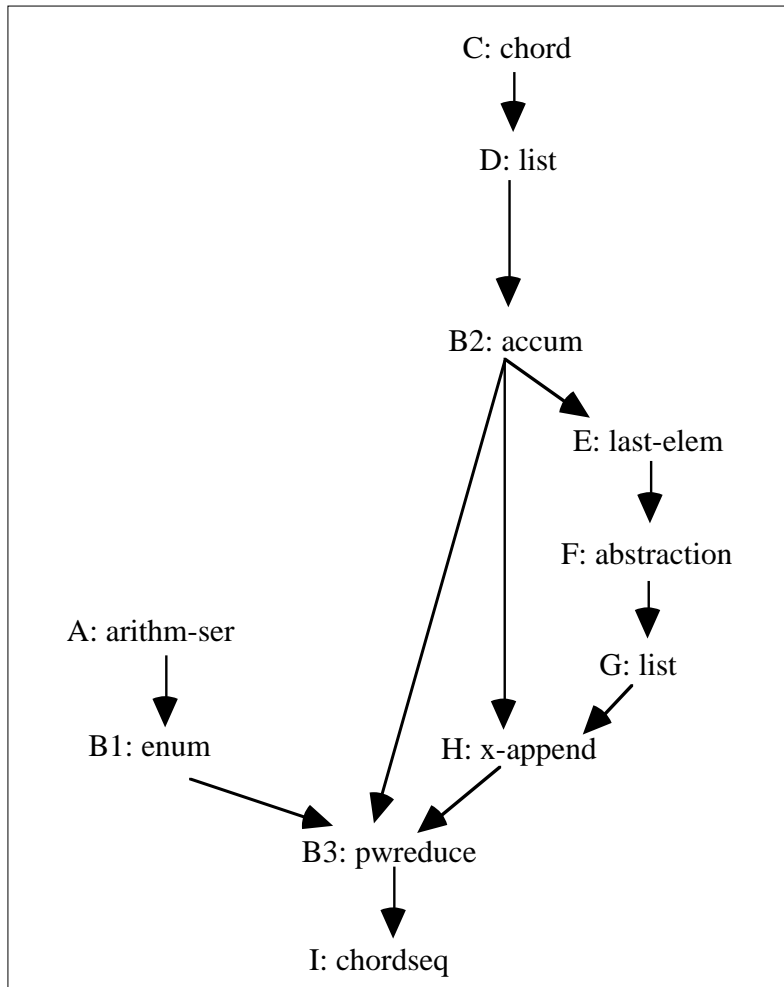
Using recursion, iteration and abstraction to create a series of chords.



Description

This patch creates a series of chords starting from a diad using both a recursive function and a iterating function.

Patch structure

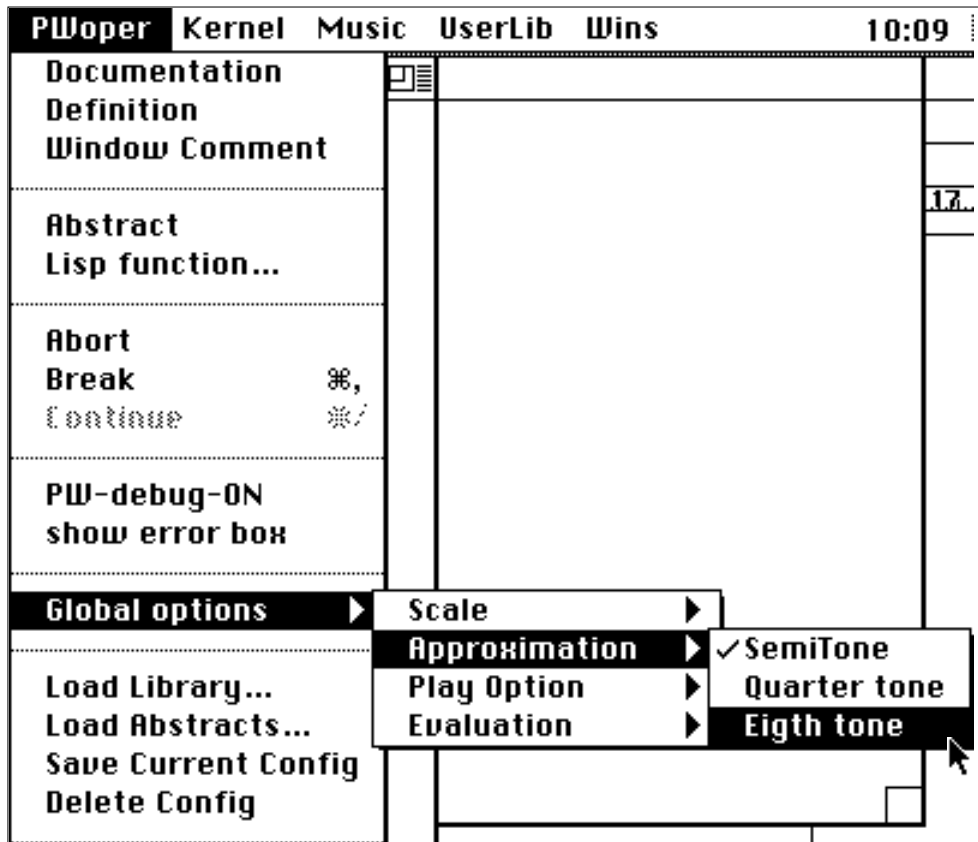


Boxes used

arithm-ser, chord, chordseq, flat, last-elem, list, pwreduce (enum, accum), rem-dups, sort-list

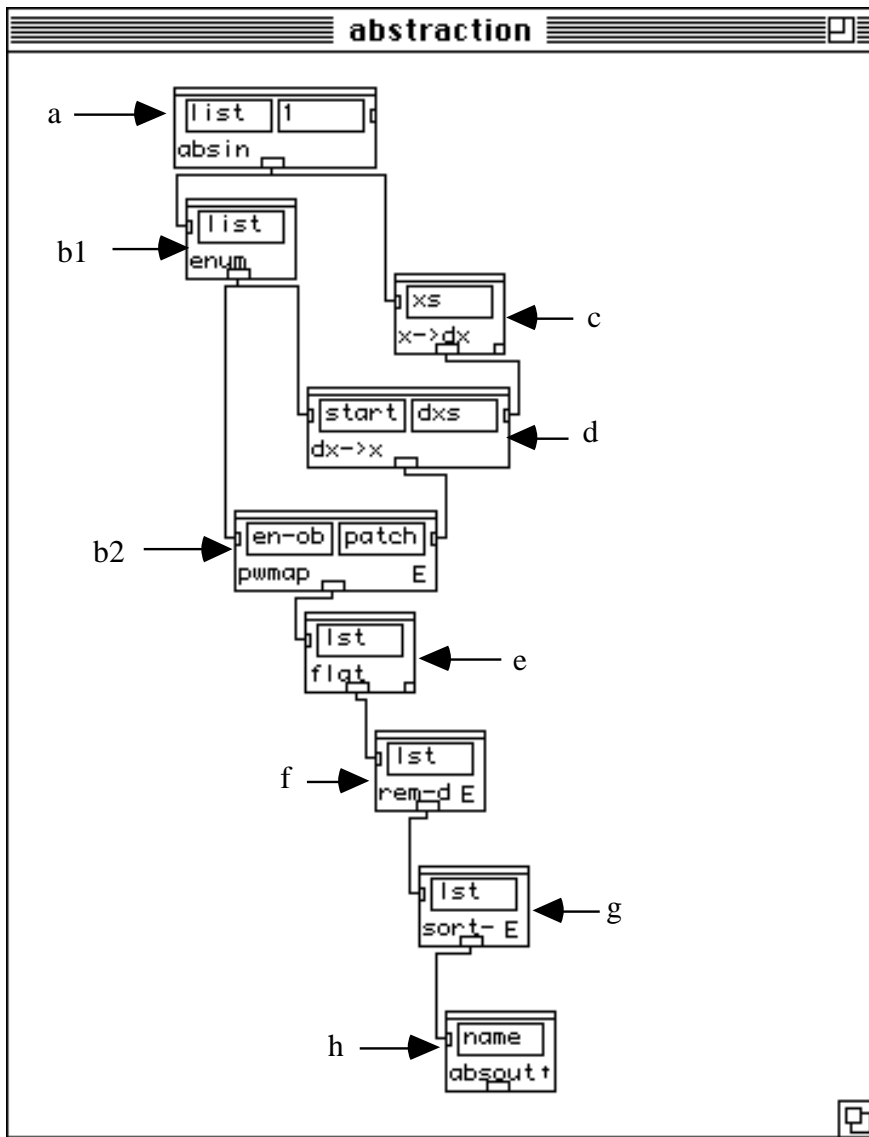
Utilisation

- A: The module arithm-ser (A) creates an arithmetic series starting from value 1, ending with the value 10, with a step of 1.
- B1: The module enum (B1) is connected with the module pwreduce. (B3) It takes the elements coming out of the arithm-ser box (A), one by one.
- C: Set the "Approximation" in the "Global option" to "Eighth tone".



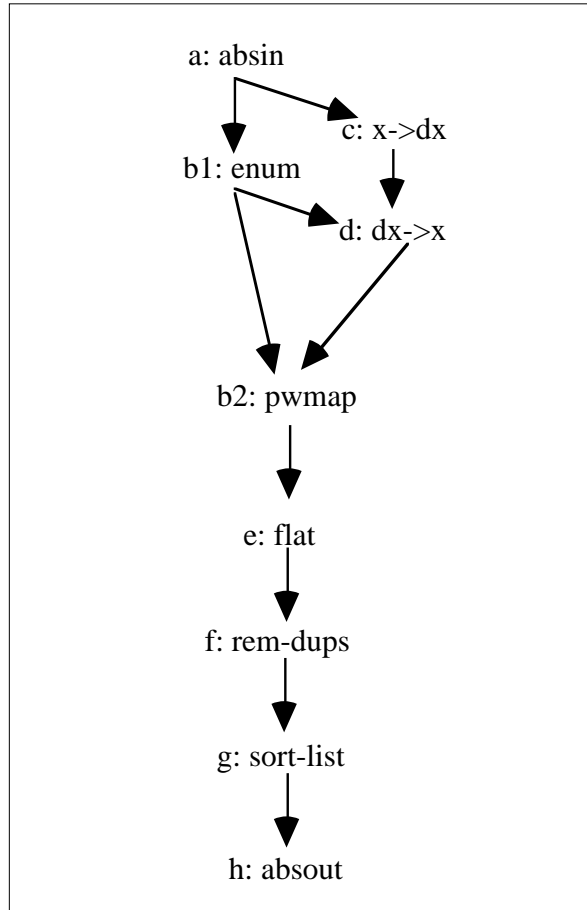
Then open and edit a diad in the chord box (C). The interval of the diad you enter will be reproduced recursively ten times.

- D: The list box (D) adds two parentheses to the diad enter in the chord box (C).
- B2: The accum box (B2) accumulates the results coming out of the previous evaluation to use it for the next evaluation.
- E: The last-elem box (D) takes the last value stored in the accum box (B2).
- F: Abstraction.



Abstraction Description

This abstraction creates many chords, starting from the notes that enter the absin box (a), and based on a series of intervals calculated in the x->dx box (c).



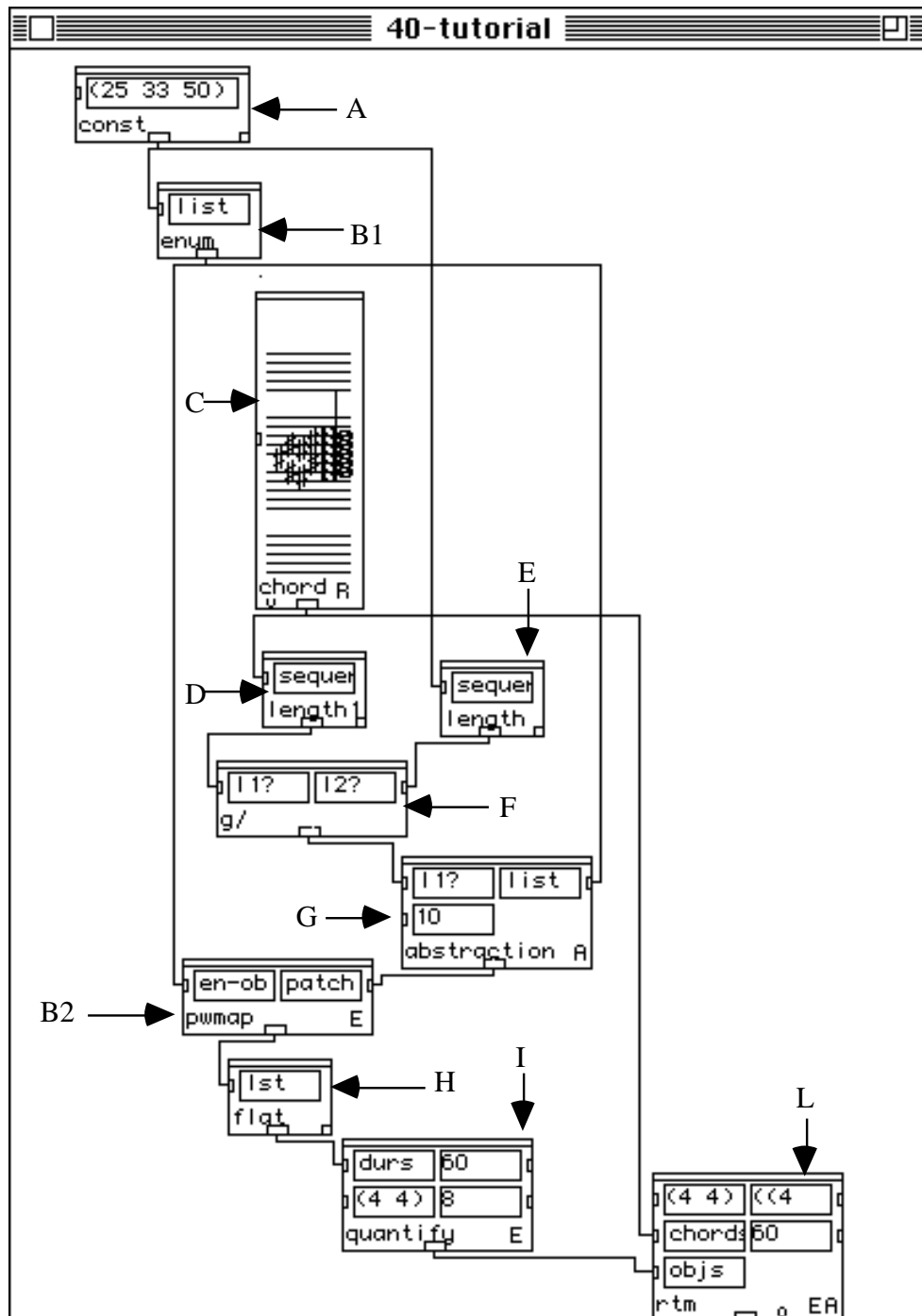
Open the abstraction by selecting "open" in its menu (or by double-clicking in its lower-right-hand corner) to see the internal structure.

- A: The absin box (a) is the input to the abstraction and it takes the result of the accum box (B2).
 - b1: The module enum is connected with the module pwwmap (b2). enum takes the elements that come out of the absin box (a), one by one.
 - C: The x->dx box (c) calculates the intervals between the notes in the sequence (a).
 - D: The dx->x module (d) makes new sequences starting with the values coming from enum box (b1), and are constructed using the intervals derived from the x->dx box (c).
 - b2: The module pwwmap (b2) allows the repetition of the same process for all the elements coming out of absin (a).
 - E: The flat box (e) sends out a list with all elements on the same level.
 - F: The rem-dups box (f) removes all duplicates.
 - G: The sort-list box (g) orders the elements coming out of the rem-dups box (f) from largest to smallest.
 - H: The absout box (h) is the output of the abstraction.
- Back to the main patch...

- G: The list box (D) adds two parenthesis to the diad enter in the chord box (C).
- H: The x-append box (G) appends together the values entering in its inputs. In this example, it appends the list stored in the accum box (B2) to the value coming out of the list box (D). In this way, the next iteration of the recursive function will take this last element as the new input.
- B3: The module pwreduce (B3) closes the recursive process.
- I: Evaluate the chordseq box (I) to view the obtained chords. As you can see the structure of each chord is based on the interval of the diad entered in the chord box (C). Change the diad, then the "end" of the arithm-ser box (A), re-evaluate the chordseq box (I) and look at the new series of chords.

Tutorial 40

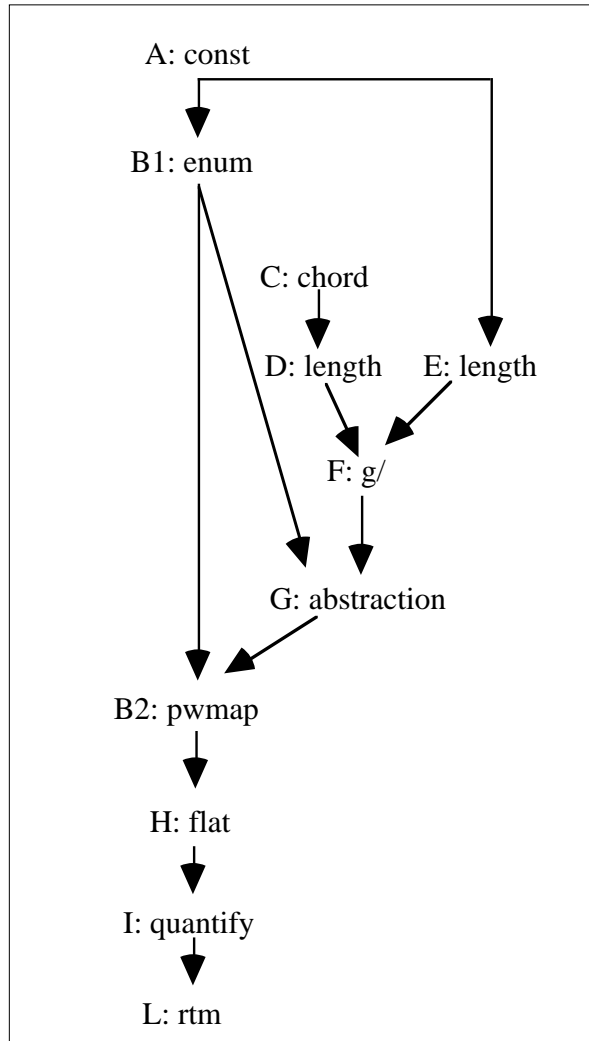
Using recursion, iteration and abstraction: creation of a rhythmic structure.



Description

This patch creates a rhythmical structure using both a recursive function and a function that iterates many times the recursive one.

Patch structure



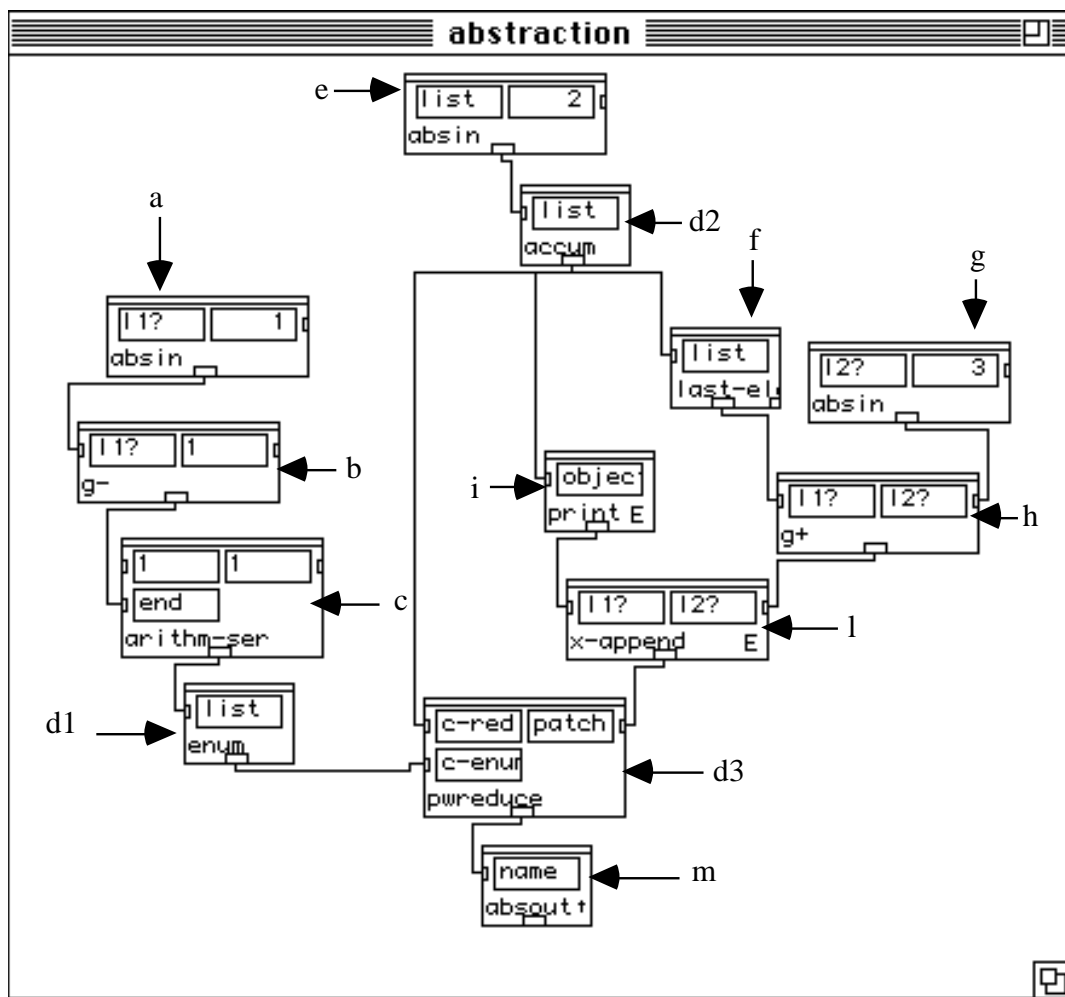
Boxes used

arithm-ser, chord, chordseq, flat, last-elem, list, pwreduce (enum, accum), rem-dups, sort-list

Utilisation

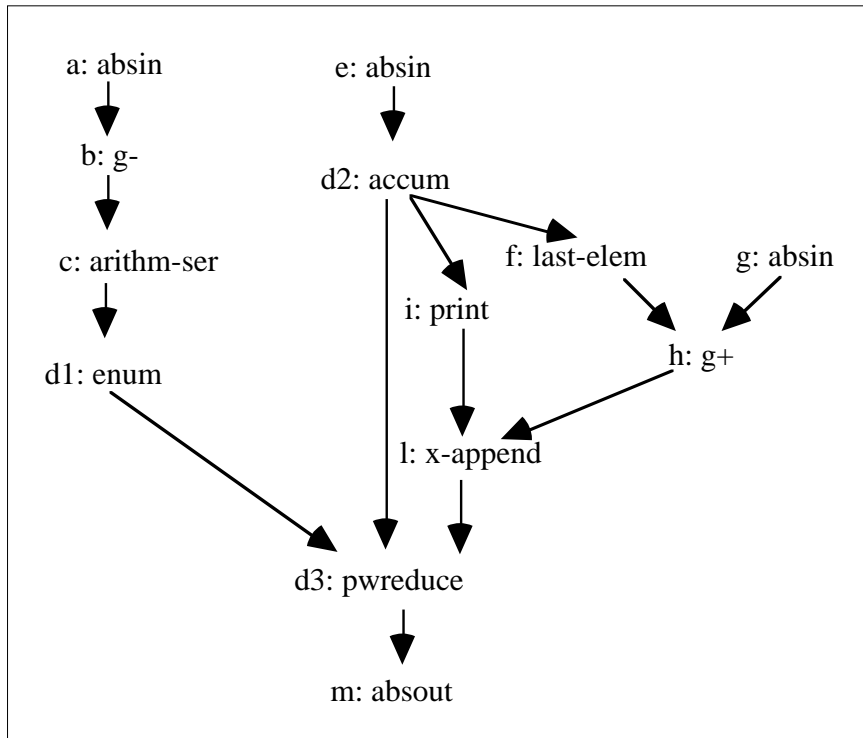
- A: Enter in the const box (A) the values to be used for generating the rhythmical structure. The number of values you enter will correspond to how many times the recursion its repeated. Each value will be correspond to the initialisation value for each recursion.
- B1: The module enum is automatically connected to the module pwmap. The enum box (B1) takes the elements coming out of the const box (A) one by one.

- C: In this chord box (C) enter the sequence of notes to which you want to associate the rhythmical structure.
- D: The length box (D) calculates how many notes you have entered in the chord box (C).
- E: The length box (E) counts how many values you have entered in the const box (A).
- F: The g/ box divides the number of notes entered in the chord box (C) by the number of elements entered in the const box (A).
- G: Abstraction.



Abstraction Description

This abstraction creates a rhythmical structure using a recursive function. In the first input it takes the value that indicates the length of each recursion. In the second input it takes the value that initialises each recursion. In the third input it receives the value that will increase each value of the recursion.



Open the abstraction by selecting "open" in its menu (or by double-clicking in its lower-right-hand corner) to see the internal structure.

- A: The absin box (a) is the first input of the abstraction and it takes the result of the g/ box (F).
- B: The module g- (b) subtracts 1 to the result coming from the g/ box (F).
- C: The module arithm-ser (c) creates an arithmetic series starting from value 1, ending with the value coming out of the module g- (b).
- d1: The module enum (B1) is connected with the module pwreduce. It takes the elements coming out of the arithm-ser box (A), one by one.
- E: The absin box (e) is the second input of the abstraction and it takes the value coming out of the const box (A).
- d2: The accum box (B2) accumulates the results coming out of the previous evaluation to utilise it for the next evaluation. The initialising value of the accum box (d2) come from the cost box (A).
- F: The last-elem box (f) takes the last value stored in the accum box (d2).
- G: In this third absin, (that corresponds to the third input of the abstraction), enter the value that will increase with each recursion step.
- H: The g+ box (g) makes the sum between the value coming out of the last-elem box (D) and the value entered in its right input.
- I: The print box (i) is a Common Lisp function. It prints all the steps of the recursion in the Listener window.
- L: The x-append box (l) appends together the values entering in its inputs. In this example, it appends the list stored in the accum box (d2) to the value coming out

of the g+ box (e). In this way, the next iteration of the recursive function will take this last element as the new input.

d3: The module pwreduce (B3) closes the recursive process.

M: The absout box (h) is the output of the abstraction.

Back to the main patch...

B2: The module pwmap (B2) allows the repetition of the same process for all the elements coming out of const box (A).

H: The flat box (H) sends out list of value coming out of the pwmap (B2) with all elements on the same level.

I: The quantify box (F) quantizes the list of values coming out of the recursive process into a given metric measure (4/4), with a tempo (quarter = 60) and with a maximum beat division (8).

L: Evaluate rtm box (L) to see the result. Now change the value in the const box, then the value that will increase each recursion step, in the third input of the abstraction. Re-evaluate the rtm box in order to see the new results.

Index

A

absin 86, 87, 90, 92, 93, 94, 96, 97, 98, 110, 111, 112,
115, 116, 117, 147, 148, 153
absout 86, 87, 90, 92, 94, 96, 98, 110, 112, 115, 117, 148,
154
Abstraction 85, 144
Approximation 104, 106
approx-m 105, 107, 108
Apps 17
arithm-ser 96, 97, 98, 100, 103, 131, 138, 139, 140, 142,
145, 149, 151, 153
Atom 29
Atoms 29

B

BPF 52, 54, 109
bpf 53, 55, 57, 59, 61, 63, 64, 73, 110, 112, 115, 117, 142
bpf-sample 55, 57, 59, 61, 63, 64, 73, 110, 112, 115, 117
buffer 92, 94, 100, 101, 103, 105, 107, 108, 131
butlast 124, 126

C

Chant 8
chord 36, 37, 38, 39, 40, 41, 42, 43, 44, 46, 48, 50, 53, 54,
55, 57, 58, 59, 61, 64, 66, 68, 69, 71, 73, 75, 76, 77,
78, 80, 81, 83, 84, 85, 86, 92, 96, 98, 119, 121, 124,
125, 126, 128, 133, 134, 136, 137, 145, 146, 149,
151, 152
chordseq 42, 44, 46, 48, 50, 51, 53, 55, 57, 59, 63, 64, 71,
73, 74, 78, 80, 81, 83, 84, 86, 92, 94, 96, 98, 100, 101,
103, 105, 107, 108, 112, 113, 115, 117, 131, 132,
134, 145, 149, 151
Common Lisp 7, 8, 13, 15, 17, 18, 29, 30, 31, 33, 123,
127, 130
Compression 47
Connect 24
const 21, 23, 24, 25, 43, 44, 52, 53, 66, 68, 69, 71, 107,
108, 119, 136, 137, 138, 139, 140, 142, 151, 152,
153, 154
Conversion 99, 102
 frequencies to midicents 102
 notes to symbols 99
 symbols to notes 102
count 130, 131
Counting notes 130
create-list 66, 68, 69
Creating a geometric curve 141
Creation of a rhythmical structure 150
Csound 8

D

d 10
Documentation 11
Duthen J. 2
dx->x 44, 48, 50, 51, 86, 92, 94, 148
dx-x 44, 46

E

Edit 20, 21
Evaluate 27
expand-list 119, 121, 122
Expansion 47
explode 89
Extraction 123

F

f->mc 103, 105, 107, 108
fibonacci-ser 68, 69
Filtering notes 79
first 44, 50, 51, 57, 80, 83, 124, 134, 136, 137
flat 145

G

g- 29, 40, 44, 57, 61, 63, 64, 68, 69, 73, 83, 92, 94, 96, 98,
103, 105, 107, 108, 110, 112, 115, 117, 128, 142,
143, 153
g* 47, 48, 50, 51, 59, 61, 92, 103, 142, 143
g+ 24, 25, 27, 28, 37, 38, 40, 42, 50, 51, 78, 83, 84, 136,
137, 138, 139, 153, 154
g/ 100, 103, 153
Generation of chords 91, 95
Generation of list 67
Grouping lists 70, 72, 75
group-list 71, 73, 107, 108
g-scaling/sum 59

H

High-pass filter 79

I

Images 13
in 63, 64, 68, 73, 110
int->symb 100, 101
Interpolation 33
Inversion 39, 49
Iteration 79, 82, 144, 150

K

Kernel 21, 33, 124

L

last-elem 142, 143, 145, 146, 151, 153

Laurson M. 2

length 59, 61, 66, 68, 73, 100, 119, 152

List 29, 30, 65, 67, 72, 75

list 63, 64, 151

List of lists 29, 30

Listener 14, 17, 30, 129

list-explode 76, 100, 128, 129

Low-pass filter 79

M

mapcar 128, 129

mat-trans 63, 64

mc->n 100, 101

Melodic and rhythmic notation 118

Midi 33

Multiplication of chords 85, 91

Music

notation 33

N

n->mc 103

Notation 33, 109, 114, 118

Notation of multiple rhythmical sequences 120

nth-random 96, 98, 103, 128, 131

numbox 36, 37, 38, 46, 48, 54, 55, 57, 59, 61, 63, 64, 68, 69, 73, 74, 78, 96, 98, 110, 111, 112, 113, 119, 134

O

Object 29

On-line documentation 11

out 63, 64, 68, 73, 110

P

permut-random 100, 101

posn-match 76

Proportional notation 109, 114

prtm 121, 122

PW 16

pwmap 78, 80, 81, 83, 84, 86, 92, 94, 96, 98, 100, 101, 107, 108, 131, 132, 139, 148, 151, 154

PW-Music.image 13, 14

pwreduce 138, 139, 140, 142, 143, 145, 149, 151, 153, 154

pwrepeat 92, 94, 96, 98, 103, 105, 107, 128, 131

Q

quantify 59, 61, 66, 68, 119, 121, 154

R

Recursion 138, 141, 144, 150

rem-dups 92, 94, 145, 148, 151

remove 80, 81

rest 124, 126

Retrograde 49

Retrograde of the inversion. 49
Retrograde transposition 41
reverse 42, 50, 51
rtm 59, 61, 66, 68, 69, 119, 121, 154
Rueda C. 2

S

Scale 43
sort-list 68, 69, 145, 148, 151
Sub-list 70
substitute 134
Substituting a chord for a note 133
Synthesis 8

T

t 12
test 80, 81, 83
Transposition 36, 45, 49, 77, 135
trigger 76
Tutorial 36
Twelve-tone row 49

X

x->dx 46, 47, 48, 50, 51, 86, 92, 93, 94, 97, 147, 148
x-append 40, 136, 137, 142, 149, 153
x-dx 46, 51