Ircam documentation

- Etudes et recherches
- Œuvres musicales
- Logiciels

PatchWork

PW-Functionals Library

r e f e r e n c e

Second Edition, April 1996

IRCAM Centre Georges Pompidou

This manual was written by Camilo Rueda, and produced under the editorial responsibility of Marc Battier - Marketing Office, Ircam.

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

2nd edition, April 1996.
This documentation corresponds to version 2.0 or higher of PatchWork.
Version 1.0 of the library.

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

IRCAM Users' group

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

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

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

Send comments or suggestions to the editor:

E-mail: bam@ircam.fr
Mail: Marc Battier,
Ircam, Département de la Valorisation
Place Stravinsky, F-75004 Paris

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

Contents

Résumé

La librairie de modules PW-Functionals pour PatchWork est utilisée pour

• Créer et transformer des fonctions

• Définir et manipuler des structures de données « infinies »

Ce manuel présente l'ensemble des modules de cette librairie.

1 Introduction

PW-Functionals is a library of PatchWork modules convenient for two purposes:

- 1. Creating and transforming functions.
- 2. Defining and manipulating 'infinite' data structures.

The idea is to allow a patch to be considered as some sort of object that can be used to parametrize the behaviour of a suitable PatchWork box. If we take the **g-oper** module, for example, we see that its first entry accepts a function (by default, the function '+'). Just by giving appropiate functions to this input the **g-oper** module can be made to behave as any one of the modules appearing in the **Arithmetic** menu of PatchWork. This feature represents thus a nice mechanism for abstracting into a single box many different functionalities. Unfortunately, the PatchWork kernel does not provide a way to define a function other than by typing its LISP code in the **make-num-fun** module. Some modules in the PW-Functionals library (see the **Function** submenu) on the other hand, serve to transform a normal patch into its equivalent LISP function thus allowing *graphical* function definitions. As a side effect, this also provides a mechanism for constructing recursive patches by including in the patch to be transformed into a function, a module calling that function. All this will be explained below in greater detail. Once functions can be suitably created by patches a different way of programming in PatchWork becomes apparent . In this alternative conception a patch is always a sequence beginning with the generation of a (possibly infinite!) data structure and followed by a sequence of filterings or functional transformations of that structure. Modules in the PW-Functionals library (see the **Series** submenu) help to create the data structure and to perform the desired transformations. applicable to musical contexts.

We describe below each module of the library.

2 Function

funarg

Converts a patch into an equivalent function

Inputs

patch a patch

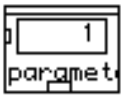
Optional inputs

name (symbol)

Returns a LISP function equivalent to the input patch. If the entry *patch* is not connected, the last constructed function is returned (if none, a function doing nothing is returned). *name* is an optional name for the function that can be used for calling it (see the **eval-fun** module). Parameters of the function are determined by the **parameter** modules found in the input patch.

parameter

Parameter of a constructed function



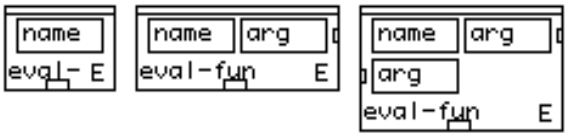
Inputs

order (number)

Defines a parameter in a patch connected to a **funarg** module. *order* must be unique among all parameters of the same **funarg**.

eval-fun

Evaluates a function defined by funarg



Inputs

name (symbol)

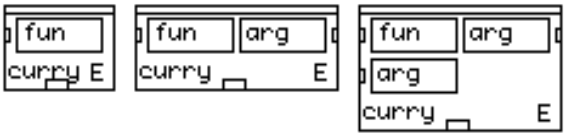
Optional inputs (any number of them)

arg (a patch)

Returns the result of calling the function *name*. This should be any function defined by the module **funarg**. **eval-fun** may appear within the patch defining the function *name* thus performing a recursive call. Each of the *arg* inputs (available by extending the box as needed) define a value for the corresponding parameter of the function *name* .

curry

Specializes a function



Inputs

fun (a function)

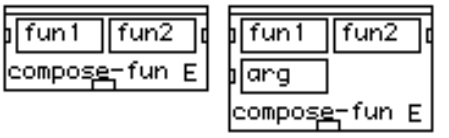
Optional inputs (any number of them)

arg (arbitrary value)

Constructs a new function by fixing the value of some of the parameters of *fun* . This new function is thus a "specialization" of *fun* for the supplied values. For example, if *fun* is a function of three numeric inputs *a*, *b* , c , evaluating the module **curry** with only one value (say 3) for b, returns a function of two arguments *a* , *c* . Calling this new function with values, say 5 and 7, for its inputs *a* ,and *c* , is exactly the same as calling *fun* with values 5, 3 and 7. For this example then, the **curry** module is first extended THREE times (thus having input boxes for each possible parameter of *fun*) but only the second input is given a value , while the first and third are left with their default values (a mysterious negative integer). The module **curry** is named after the mathematician who first formally defined this operation.

compose-fun

Combines several functions



Inputs

fun1 (function)

fun2 (function)

Optional inputs (any number of them)

arg (function)

fun1, *fun2* and all the *arg*'*s* must be functions of only one argument. Let's say this argument is *x* . **compose-fun** returns a new function of one argument. Calling this function is equivalent to calling *fun1* (*fun2* (*arg*...(*x*))). For example, if *fun1* adds 5 to its argument, and *fun2* squares it then the result of evaluating **compose-fun** with this two functions is a function computing

(*X* + 5) * (*X* + 5)

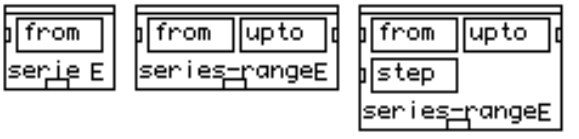
3 Series

Series are lists representing "infinite" data structures. Infinite structures are handled by what is called *delayed evaluation* . Basically only the first element of the structure is explicitely given while the computation of subsequent elements is deferred to the time where they are really needed. The boxes described below serve to construct and manipulate these data structures.

Construction

series-range

Constructs an arithmetic series



Inputs

from (number)

Optional inputs

upto (number)

step (number)

Constructs a (possibly infinite) arithmetic series starting at *from* . If *upto* is given, the last element of the series is not higher than it. *step* is the increment between elements (default 1).
The optional argument <type> allows the choice of whether the output is a list of chords ('seq') or a single chord ('chord') containing all the transpositions combined.

make-series

Constructs an arbitrary series

Inputs

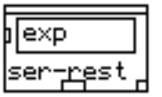
init	(a function)
next	(a function)
test	(a function)

Constructs a series defined by three functions. *init* must be a function with no arguments which gives the first element of the series. *next* must be a one argument function which, given an element of the series (its argument) returns the next element in the series. *test* must be a one argument function which, given an element of the series (its argument) returns t if it is the last one or NIL otherwise.

Selection

ser-rest

Removes the first element of a series



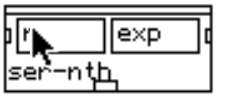
Inputs

exp (a series)

Returns a series equivalent to *exp* but without the first element.

ser-nth

Gives the nth element of a series



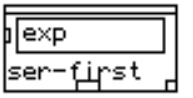
Inputs

n	(an integer)
exp	(a series)

Returns the element in position *n* in the series *exp* .

ser-first

Gives the first element of a series



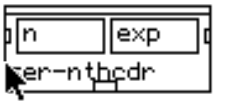
Inputs

exp (a series)

Returns the first element of the series *exp*.

ser-nthcdr

Removes elements from the head of a series



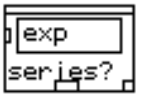
Inputs

n	(an integer)
exp	(a series)

Returns a series equivalent to *exp* without the first *n* elements.

series?

Tests if something is a series



Inputs

exp (anything)

Returns t if *exp* is a series. NIL otherwise.

Conversion

ser->list

Converts a series into a list

Inputs

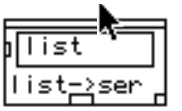
exp (a series)

Outputs a list with all the elements in the series *exp* .

warning : A series might be infinite or very big, so use this module only when you are sure about its size.

list->series

Converts a list into a series



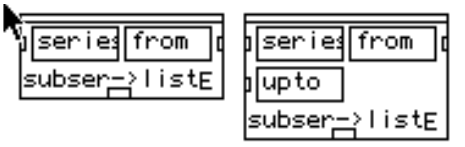
Inputs

list (a list)

Returns a series containing the elements in *list* .

subser->list

Puts selected elements of a series into a list



Inputs

series (a series)

from (an integer)

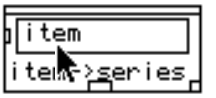
Optional inputs

upto (an integer)

Returns a list containing the elements of *series* starting at position *from* and (optionally) ending at position *upto* .

item->series

Converts an item into a series



Inputs

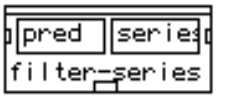
item (anything)

Returns an infinite series of *item* .

Transformation

filter-series

Filters a series



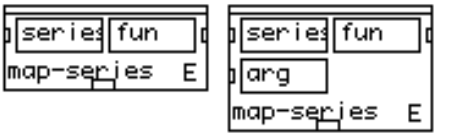
Inputs

pred	(a function)
series	(a series)

Filters *series* by eliminated all elements that do not satisfy the predicate *pred* which should be a one argument function returning t or NIL accordingly.

map-series

Applies a function to a series



Inputs

series (a series)

fun (a function)

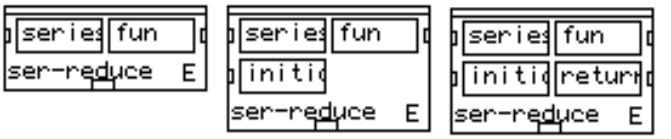
Optional inputs (any number)

arg (a series)

Returns the series resulting of applying *fun* to each corresponding element of *series* and each of the *arg's* (if any). The number of arguments of *fun* must be equal to the number of series in the inputs to **map-series**.

ser-reduce

Reduces a series by a function



Inputs

series (a series)

fun (a function)

Optional inputs (any number)

initial (any value)

returns (a menu)

Reduces *series* by the function *fun* . If "value' is chosen for *return* (a menu) the result is a single value. If 'series' is chosen the result is a series of all successive values in the reduction. *initial* is the initial value for the reduction (default 0). *fun* must be a two argument function. A reduction is an operation that goes as follows: First, *fun* is evaluated with arguments *initial* and the first element of *series*. . In the next step, *fun* is evaluated with arguments the result of the previous operation and the second element of *series* . In each successive step *fun* is evaluated with arguments the result of the previous operation and the next element of *series* . So the final result can be either the last value computed in this way (if *return* is 'value') or the series of values computed in each step of the process (if *return* is 'series'). Keep in mind that a *series* being a potentially very large data structure it is usually safer to choose the 'series' option for *return* (so there is no need to go through the whole input series for computing the output).

append-series

concatenates two or more series

Inputs

series1 (a series)

fseries2 (a series)

Optional inputs (any number)

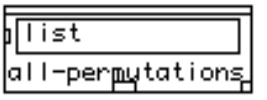
arg (a series)

Returns the series resulting of concatenating *series1* , *series2* and all *arg* series (if any) .

Utilities

all-permutations

Gets all permutations of a given list



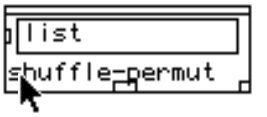
Inputs

list (a list of numbers)

Returns a series containing all different permutations of tbe elements in *list* . Permutations are given in lexicographical order starting from an ordering of *list* . Note that *list* must be a list of numbers (not necessarily unique).

shuffle-permutations

al l permutations of a given list



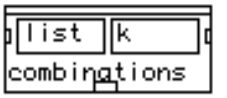
Inputs

list (a list of numbers)

Same as **all-permutations** but the first permutation is *list* . The numbers given in *list* must be unique.

combinations

al l selections from a given list



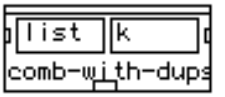
Inputs

list	(a list of numbers)
k	(a number)

Returns a series of all different selections of *k* elements in *list* .

comb-with-dups

Selections with repetitions



Inputs

list	(a list of numbers)
k	(a number)

Returns a series of all different selections of *k* elements in *list* . Each element of *list* is repeated up to *k* times in the output.

cartesian

Cartesian product of two series

Inputs

s1	(a series)
k	(a series)

Returns a series consisting of the cartesian product of *s1* and *s2* .

all-replacements

Replaces elements of a series

Inputs

list	(a list)
ftem	(any value)

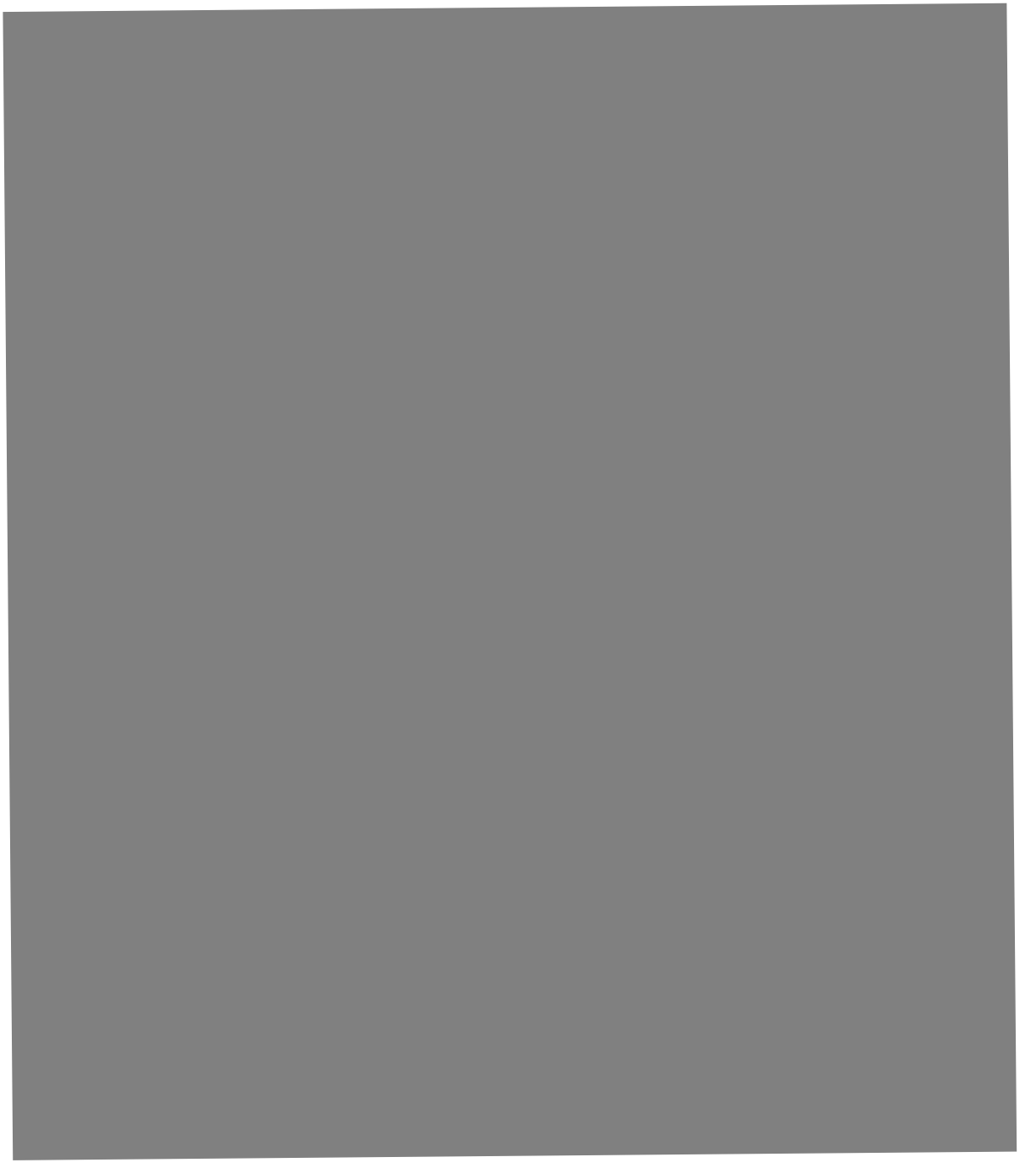
Optional inputs

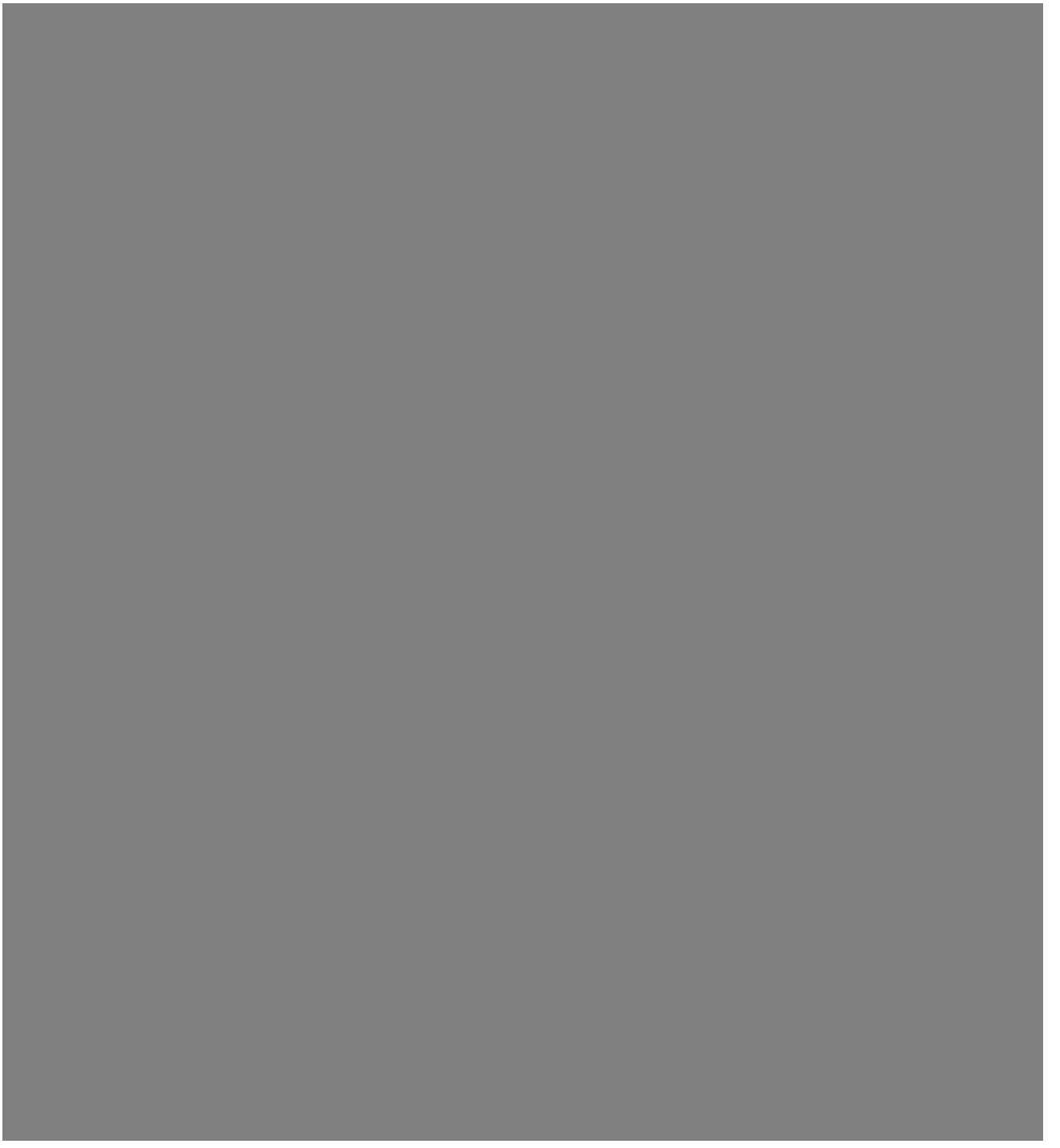
repet	(a non negative number)

Returns the series resulting of all possible ways to replace *item* by elements of *list* . The input *repet* gives the maximum number of allowed repetitions of *item* in each of the elements of the output series (default is the length of *list*).

Examples

The next two examples are intended to give a general idea of how to design patches using the modules described above. In the first example (figure 1) a particular chord of 7 notes is given and the aim is to build a melodic line using sequences of notes that are permutations of the basic chord. Only sequences containing an interval of a major seventh and an interval of a minor sixth are retained. No two sequences are allowed to have the same two notes at the beginning (in any order). The second example (figure 2) is more elaborate. Given a spectrum and a chord the purpose is to find those sets of frequencies (notes) in the spectrum that best approximates the chord. There are two measures for defining the approximation. One is a standard euclidean distance between the spectrum subset and the chord: The square root of the sum of the squares of the distance between each corresponding note of the spectrum and the chord. The second measure is the comparison of the difference between maximum and minimum intervals in both the spectrum subset and the chord (this is called "homogeneity" in the example). Notes distance measure must be less than 10 and homogeneity difference must be less than or equal to 2. From all the subsets satisfying these conditions, only 24 are displayed, in increasing distance order.





Index