Contents

Viuhka

(Finnish, in English 'Fan', in French 'Eventail')

1. Introduction

Viuhka is a PW user-library for creating complex, multi-layered musical textures. These can be used to produce scores for instrumental music and to provide material for sound synthesis. The end result of a Viuhka patch can either be displayed in a PW multiseq module or translated to a synthesis file, typically a Csound score file, or a midi-file.

The starting point is a group of bpfs (*Viuhka bpfs*) that constitute the pitch field (harmonic skeleton) of the result. Viuhka bpfs provide the main pitch material which can further be reacted to (or elaborated by) different sorts of *Viuhka ornaments*. Besides "typical" ornaments (like grace notes, trills, etc.), the Viuhka ornaments can also be runs, repetitions, chords, clusters, clouds or even complete Viuhka patches (i.e. a Viuhka patch can contain other Viuhka patches).

The user typically divides a Viuhka patch into musical time slices or *segments*. The duration of the segments are given as a list of delta-times (or time intervals). The structure of each segment (i.e. number of layers, rhythmic structure, time modification, ornaments and synthesis information) is normally given inside a PW abstraction. There the user defines the *Viuhka parameter values* by using v-params and cs-params boxes. The output of the v-params boxes are then given to the main Viuhka box (called mk-viuhka) that in turn calculates the final score.

In the following text we introduce the main components of Viuhka. We start with a section describing the Viuhka bpfs. Then the main Viuhka box is introduced. The

description of Viuhka parameter values and ornaments follows. The last section deals with Csound score files. The Appendix serves both as a reference for Viuhka boxes and functions and as a tutorial where we give several example patches.

2. Creating Viuhka BPFs

This section describes the two functions that are used to create Viuhka bpfs.

patch-work::make-viuhka-bpfs midics-lst mappings delta-times [function]
midics-lst list of lists of midics (= list of midic-chords)
mappings list of lists of mappings (*harm-places* = numbers pointing to individual midic-values within midics sublists)
delta-times list of delta-times.

make-viuhka-bpfs returns (as a list) a list of lists of x-values and a list of lists of y-values for bpfs. The number of the resulting bpfs is given by the length of each mapping sublist (the mapping sublists must be of equal length).

delta-times give the x-values for the resulting bpfs (delta-times are translated to absolute times starting from 0). All resulting bpfs share the same x-value list. The length of delta-times must always be one less than the lengths of the mapping sublists.

midics-lst, in turn, combined with mappings produces the y-value lists. Each midic-chord sublist is associated to each mapping sublist in the following way. The first y-value list (the first bpf) is obtained by picking the *first* harm-place-values from each mapping sublist. The resulting list, *harm-place-line*, is used then to pick midic-values so that the first harm-place-value is applied to the first midic sublist, the second harm-place-value to the second

midic sublist, etc. The second y-value list is obtained in a similar fashion using the second harm-place-line (= the *second* harm-place-values from each mapping sublist), and so on.

For instance, if midics-lst is ((6000 6400 6700) (5200 5600 6000) (5200 5600 6000)), mappings ((1 1 2 3) (1 2 3 3) (1 1 1 3)) and delta-times (100 200 100), then the result will contain four bpfs (= the length of each mapping sublist). The resulting x-value lists will all be (0 100 300 400).

The y-value list for the first bpf, (6000 5200 5200), is found by picking from the first harm-place of each mapping (1 1 1); the second list of y-values, (6000 5600 5200), is found by picking from the second harm-place of each mapping (1 2 1); the third list, (6400 6000 5200), is found by picking from the third harm-place of each mapping (2 3 1) and finally the fourth list, (6700 6000 6000), is found by picking from the fourth harm-place of each mapping (3 3 3).

Figure 1 below gives an example of the make-viuhka-bpfs box inside a patch. The user gives four chords that constitute the midics-lst input. The mappings and delta-times inputs are given with const-boxes. The resulting bpfs are shown in the lowest multi-bpf box:

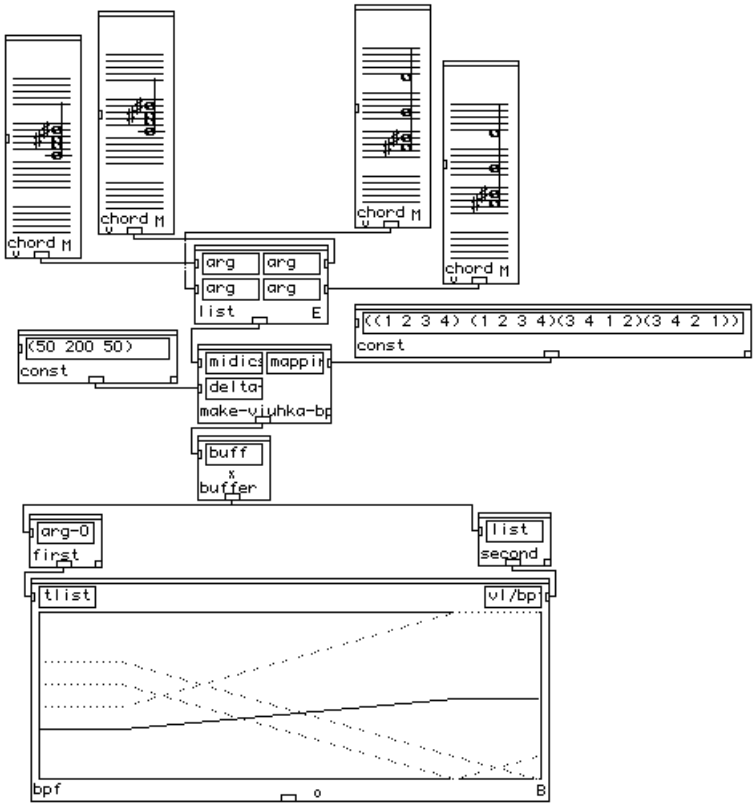


Fig.1: Creating bpfs with the make-viuhka-bpfs box.

patch-work::make-bpfs times+values
[function]

make-bpfs makes bpf objects out of times+values (a list consisting of list of lists of x-values and y-values). The output of this function is typically connected to the first input of the mk-viuhka box (explained below).

3. Mk-Viuhka

In this section we discuss the primary Viuhka function, called mk-viuhka. We describe its inputs and the way the Viuhka parameter values are prepared for the mk-viuhka function.

patch-work::mk-viuhka bpfs global-dtimes param-ls-ls &optional key-flags
[function]
bpfs list bpf objects

global-dtimes	list of delta-times
param-ls-ls	list of lists of Viuhka parameter values
key-flags	optional input for special flags defining the behaviour of mk-viuhka

mk-viuhka returns a list of chord-lines. This output is normally connected to a PW multiseq module or to a ->csound module (the ->csound box will be explained in section 5). The multiseq module allows also to save the result as a midi-file.

bpfs is a list of bpf objects defining the pitch field (harmonic skeleton) of the result. global-dtimes is a list of delta-times which give the durations of the segments to be produced. The length of global-dtimes has to match to the number of segments defined in the Viuhka parameters (explained below, section 4.2.6). param-ls-ls is a list of lists of Viuhka parameter value-groups (given by v-params boxes), where each sublist describes the structure of a segment. Finally, the optional input, key-flags, of special flags defines the behaviour of mk-viuhka.

The patch given in figure 2 shows the main window of a complete Viuhka patch. The patch is split into two parts (the split is indicated by the horizontal dotted line). The upper part shows how the Viuhka bpfs are defined (this part of the patch was explained already above in figure 1). The lower part, in turn, shows the mk-viuhka box together with its inputs: the Viuhka parameters (params abstraction) and the list of delta-times for the durations of the segments (global-dtimes). Below the mk-viuhka box there are two boxes, multiseq and ->csound, that are connected to the output of mk-viuhka.

As can be seen from figure 2, mk-viuhka will use four bpfs (the bpfs input) for the pitch skeleton. The result will have three segments lasting 300, 400 and 250 ticks[1] (the global-dtimes input) and thus the duration of the whole

[1] The time unit used in Viuhka is a PW tick (1/100th of a second).

result is 950 ticks (the sum of global-dtimes). Finally, the params abstraction defines the structure of each segment.

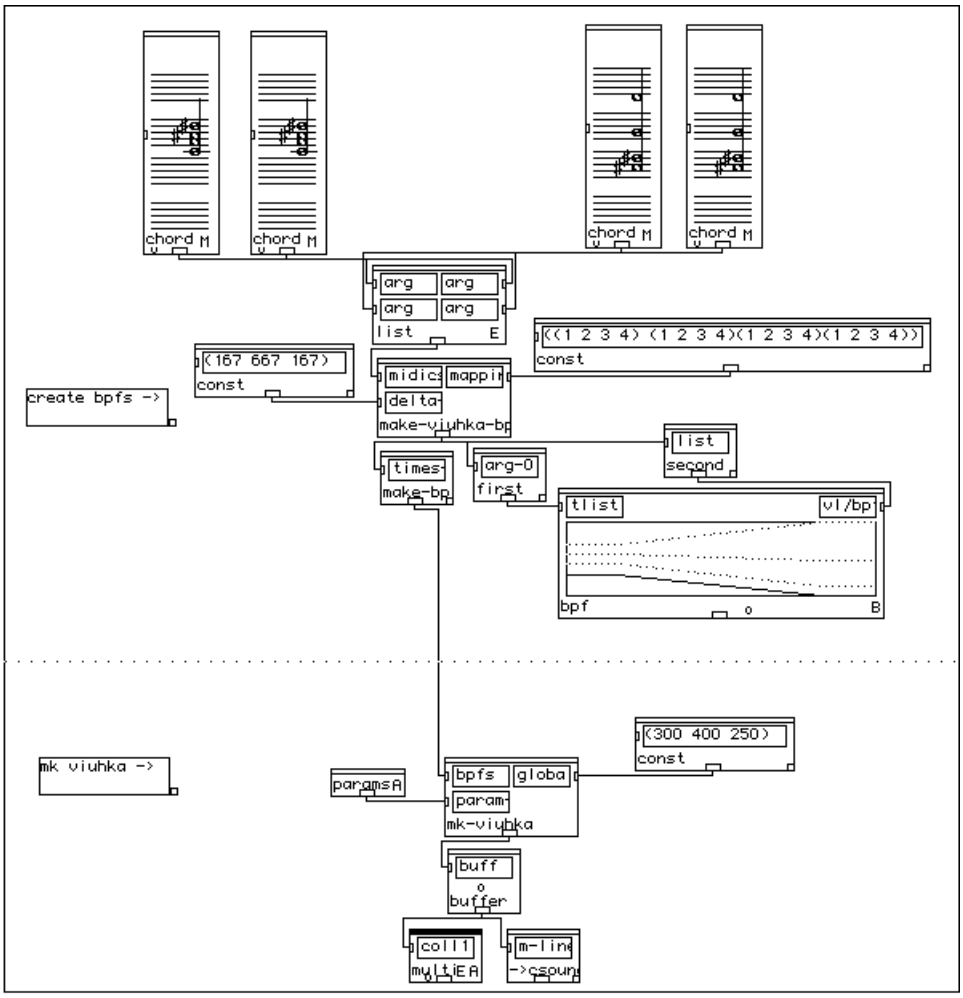


Fig.2: A complete Viuhka patch.

Figure 3 below gives the result of the Viuhka patch of figure 2 in a multiseq editor. The example consists of three segments (the segments are indicated by the vertical dotted lines). Each segment has three layers (this feature is defined inside the params abstraction).[2]

The example in figure 3 has ornaments like grace notes, glissando runs and clusters. In order to help to distinguish the ornaments from the *main notes* (i.e. notes

[2] The segments may have different number of active layers (see for details section 4.2.6).

that normally indicate the pitch skeleton given by the bpfs input), the multiseq editor is drawn in the *offs* mode (note that the offs radio button in the left-down corner is selected). The main notes are typically drawn with note stems whereas the ornament notes have no stems. Although figure 3 is quite heavily ornamented the pitch skeleton given by the Viuhka bpfs can been seen clearly (compare the chords and bpfs in figure 2 with the result in figure 3).

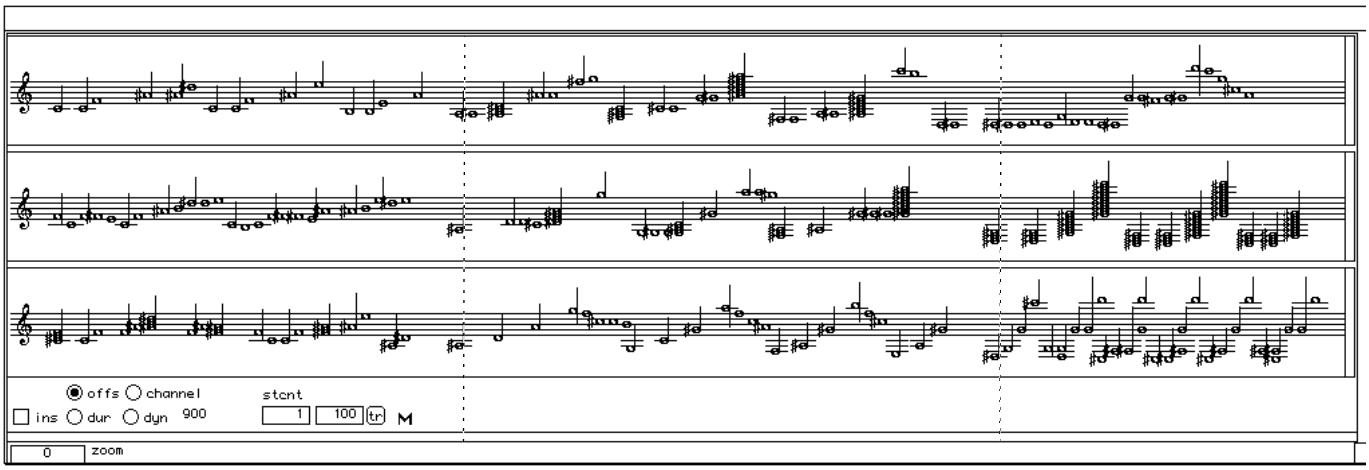


Fig.3: The result from the Viuhka patch in figure 2.

4. Viuhka Parameters

We start this segment by introducing a new data structure called *circ-list*. Next we give a short description of the global variables in Viuhka. After that we discuss the v-params box. It is used to define the structure (number of layers, ornaments, rhythm, etc.) of a Viuhka segment.

4.1 Circ-Lists

Several inputs discussed below accept as arguments circ-lists. Circ-lists are like ordinary Lisp lists except that the items are read in a circular fashion (i.e. after

reading the last item, the next item is accessed from the beginning of the list).[3] A circ-list can contain (1) constants (numbers, symbols, etc.) that simply return themselves, (2) *global variables* that are evaluated when they are being read or (3) Lisp expressions.[4] Also the latter ones are evaluated each time an expression is encountered while reading the circ-list.

A circ-list containing constants, like (1 2 a 4), returns (when read continuously) the sequence 1 2 a 4 1 2 a 4 1 2 a, etc., infinitely.

A circ-list containing global variables, like (1 gtime),
(gtime is defined as a Viuhka global variable returning the global absolute time), returns the sequence 1 <value of gtime> 1 <value of gtime> 1 <value of gtime>, etc., where <value of gtime> is the current value of the global variable gtime.

A circ-list containing Lisp expressions, like ((if (> mydur 25) 'g 'n) 66 (random 10)), returns sequences like: g 66 5 n 66 7 g 66 9, etc. Here the first Lisp expression (if (> mydur 25) 'g 'n) returns the constant g or n depending on the value of the global variable mydur. 66 is a constant returning always itself. Finally, the expression (random 10) returns a random number between 0 and 9.

Besides ordinary Lisp expressions a circ-list can also contain special Lisp functions that affect dynamically the length (and/or the content) of the circ-list. These special functions, *circ-repeat-functions*, all return lists that are flattened each time the circ-list is "recycled" (i.e. when reading the first element of the circ-list).[5] For instance, we can define a circ-list that (when read continuously) shrinks or expands its length. The following circ-list ((1 (rl 3 6 'g) 2) produces sequences like: 1 g g g 2 1 g g g g g 2 1

---

[3] PW contains a box called circ that behaves in a similar fashion than the Viuhka circ-list.
[4] Gloabal variables are discussed in the next section.
[5] We need special functions to accomplish this as normal functions returning lists are not flattened.

g g g 2, etc. We use here the special function rl that returns variable amount of the constant g. rl is defined as follows:

```
rl          cntl  cnth  elem
[function]
```

rl returns a list (containing elem) whose length varies randomly between cntl and cnth.

Circ-repeat-functions affecting dynamically the circ-list are stored in the global variable *circ-repeat-fns*. Besides rl, *circ-repeat-fns* contains currently the following functions:

```
rep-rn      n  low  high
[function]
```

rep-rn returns n repetitions of a random number. The arguments low and high define the lower and upper limits of this number.

```
rep-det     n  x
[function]
```

rep-det returns n repetitions of x.

```
rn-rn       cntl  cnth  low  high
[function]
```

rn-rn returns random values in the range low and high. The length of the result varies randomly between cntl and cnth.

circ-repeat-fns contains also the standard PW function pw::permut-random that reorders its argument (a list) randomly.

4.2 Global Variables

Viuhka contains a collection of global variables that give various sorts of information during the calculation of a Viuhka score. Global variables are typically used inside the input circ-lists and allow the user to specify expressions that react to the current state of the system (the current state refers to the global variables discussed below).

It is important to understand the order in which the calculation of a Viuhka score is performed. Inside a segment the calculation advances layer by layer (first we calculate the first layer, then the second one, etc.). Within each layer the calculation advances in time order by calculating information for the next main note (the *current note*). While calculating the current note, the system first determines the absolute start time and the delta-time. Then it chooses the *current bpf* (i.e. the Viuhka bpf giving the midic-value for the current note). After this the ornaments and synthesis information are calculated.

Expressions using global variables must follow this order in order to work properly. For instance an expression (given to an input that calculates ornaments) like:

```
(if (> mydur 50) 'g 'n)
```

is correct as `mydur` (explained below) is calculated before the ornaments. The following expression (given to an input that calculates delta-times), by contrast, is not coherent as delta-times are calculated before `mypitch` (also explained below):

```
(if (> mypitch 6000) 10 20).
```

Currently Viuhka contains the following global variables:

`gtime` the current global time in ticks.

`mydur` the duration of the current note in ticks.

`mydtime` the delta-time of the current note in ticks.

viuhka-bpfs returns all Viuhka bpfs.
myindex the index of the current Viuhka bpf.
mybpf the current Viuhka bpf.
myother-bpfs the current Viuhka bpfs except the current one.
mygrad the gradient of the current Viuhka bpf.
mytime-till-next-bp time in ticks to the next bp of the current bpf.
mypitch the pitch of the current note in midics.
mychord the current chord-midics list read from the Viuhka bpfs at current time.
mycsndparams the current Csound parameters.

4.3 V-Params Box

After this we are ready to discuss the v-params box (figure 4). It returns information of bpf indexes, delta-times, time modification, ornaments and data for a Csound orchestra. This information (usually coming from several v-params boxes) is then fed as a list to the mk-viuhka box discussed above (see the params input).

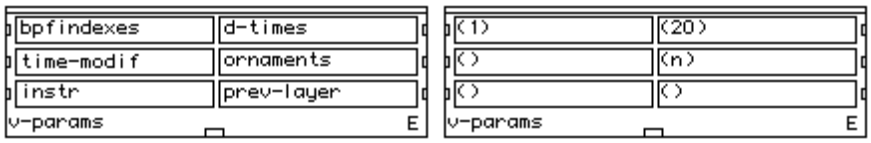


Fig.4: v-params box.

v-params has the following arguments:

v-params
[function]
bpfindexes circ-list of bpf indexes.
d-times circ-list of delta-times.
time-modif a bpf object defining the tempo-bpf or a list consisting of a tempo-bpf, focus-value and contrast-bpf.
ornaments circ-list of ornaments.
instr input from a cs-params module.

prev-layer input from another v-params box allowing to chain several v-params boxes.

&optional
ornament-vargs information from -varg boxes allowing to customise the behaviour of Viuhka ornaments.

The arguments for the v-params box are quite complex. Therefore we split the following discussion into subsections, where each subsection is dealing with one c-params argument.

4.2.1 Bpf Indexes

bpfindexes is a circ-list of indexes that are used to read Viuhka bpfs (given by the first input of the mk-viuhka box).[6] Besides numbers the list can also contain the symbol "*" meaning that the code picks randomly any Viuhka bpf. Also the user can give within the circ-list either 0 or nil. In these cases Viuhka will produce a rest instead of a note.

4.2.2 Delta-Times

d-times is a circ-list of delta-times (i.e. time intervals indicating when the next event will occur). These values define in conjunction with the time-modif input the rhythmic structure of the segment.

In order to demonstrate the effect of the first two arguments let us investigate two Viuhka examples. Figure 5 below gives a simple Viuhka patch. It consists of only one layer (lasting for 500 ticks, see the global-dtimes input of the mk-viuhka box) and has only one Viuhka bpf (see the ramp in the multi-bpf module at the top of the patch). Its y-values are scaled between 6000 and 8400 by the bpf-y-scaling module.

---

[6] The indexes start from 1. The numbering of bpfs is not checked by the system. Hence wrong indexes will produce an error.

The params input of the mk-viuhka box is defined by one v-params box. Its first input, bpfindexes, is (1), meaning that we always read the pitch information from the first and only Viuhka bpf. The second input, d-times = (20), controls the delta-times between the notes in the final result (in this case we get a new note every 20 ticks or 5 notes in a second).

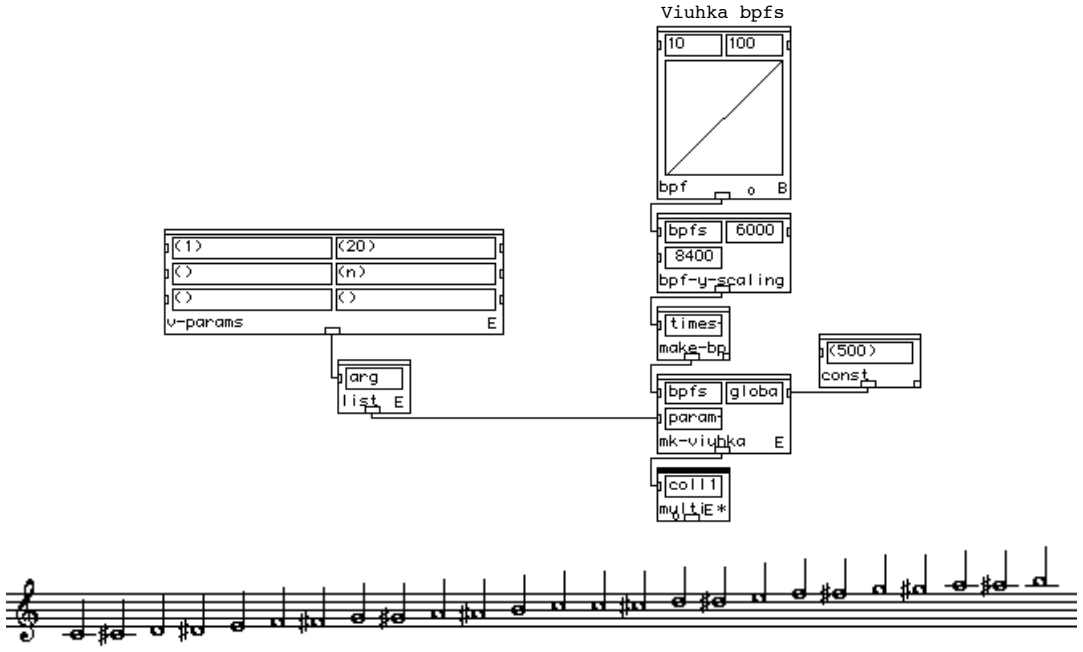


Fig.5: A simple Viuhka patch with its result.

Next we modify (figure 6) our example patch by introducing a new bpf to the Viuhka bpfs (see the two ramps in the multi-bpf module). We change the bpfindexes input to (1 2), which means that we alternate between the first and the second Viuhka bpf. Also the d-times input has changed to (20 10 15). This in turn implies that the first note gets the delta-time 20, the second 10, the third 15, the fourth 20, etc.

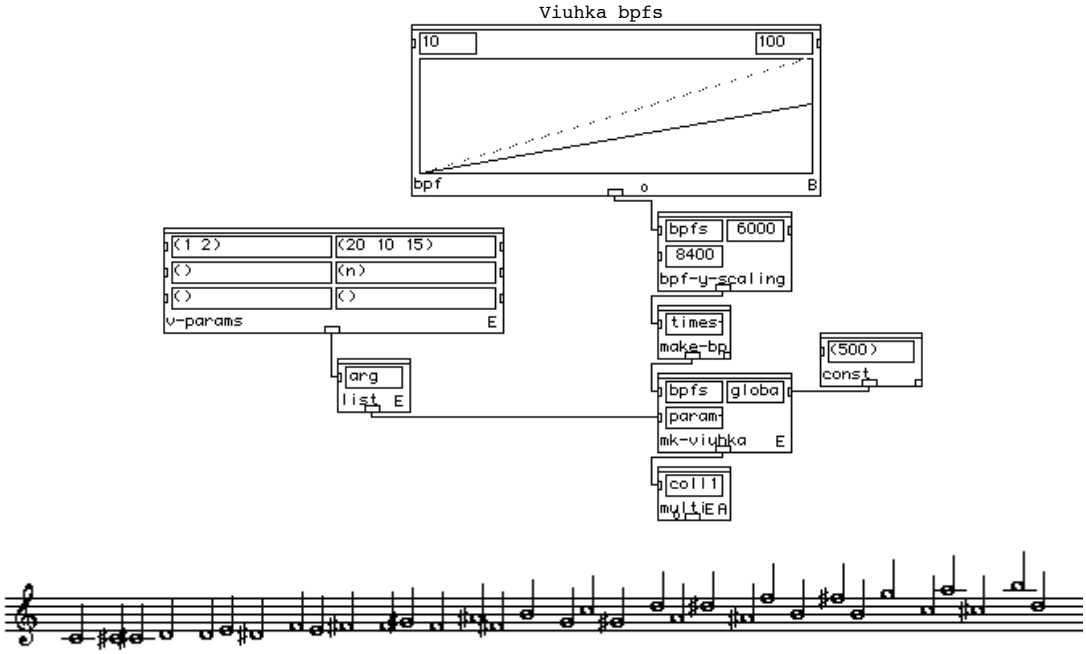


Fig.6: A modified Viuhka patch with its result.

4.2.3 Time-Modif

The time-modif input accepts two kinds of inputs and effectuates two types of time modification within a segment. If this input is not given (i.e. it is ()), then the delta-times coming from the d-times input are not modified. If, however, the time-modif input is given it must be either a single bpf object (i.e. a *tempo-bpf*, typically the output from a PW multi-bpf module) or a list consisting of a bpf object (tempo-bpf), a numeric value (*focus-value* given in ticks) and another bpf object (*contrast-bpf*).

4.2.3.1 Tempo-Bpf

When the input consists of a single bpf, the bpf acts as a tempo-bpf modifying the behaviour of delta-times. The y-values of the tempo-bpf are considered to be "metronome"

values (for instance, to double the speed one must double the current metronome value). The following conventions are used to modify delta-times: (1) each point in the tempo-bpf having the value 60 is considered to be "a tempo" value (i.e. delta-times are not modified); (2) points having values greater than 60 will shorten delta-times; (3) points lying below 60 will lengthen delta-times. Thus, to play delta-times twice as fast the y-values of the tempo-bpf should be 120, to play them at half speed the y-values should be 30, etc.

The tempo-bpf has some special properties as it allows to control the exact timings where a given tempo change will occur in the final result.

In a conventional score the performer has usually to realise the indicated accelerandi or ritardandi by playing *all* notes within the tempo change. Also one must gradually reach the desired end tempo when arriving at the last note within that group. The performer thus controls the tempo change (both the start tempo and end tempo) and the number of notes (normally the performer has to play all notes). This means that there is no direct control *when* the end point of the tempo change is reached. It is in a sense a "by-product" of making the accelerando or ritardando.

As in the conventional score, the tempo-bpf in Viuhka controls both the start and end tempo when realising a tempo change. There is, however, no direct control of the number of notes that will appear within the tempo change.[7] This in turn allows the tempo-bpf to control the exact time points for tempo changes.

To clarify this let us investigate a variation of the patch given in figure 6 above (figure 7). The second ramp of figure 6 is now modified so that it is merged with the first one for the first 250 ticks. After this it begins to

---

[7] To start with circularity (i.e. using circ-lists) brings unspecified, uncounting (quasi endless) repetition of elements.

descend rapidly and produces a gap between the ramps. The third input (time-modif) of the v-params box is connected to a multi-bpf module. The tempo-bpf makes an accelerando (from 60 to 120) for the first 250 ticks, after this it makes a ritardando (from 120 to 60). The metronome values (i.e. 60, 120 and 60) are given to the tempo-bpf by the const-box. The d-times input has been changed to (20) in order to clarify the tempo change of the result. The interesting point to note here is that the timing of the "split-point" of the Viuhka bpfs and the tempo change from accelerando to ritardando can be easily synchronised.

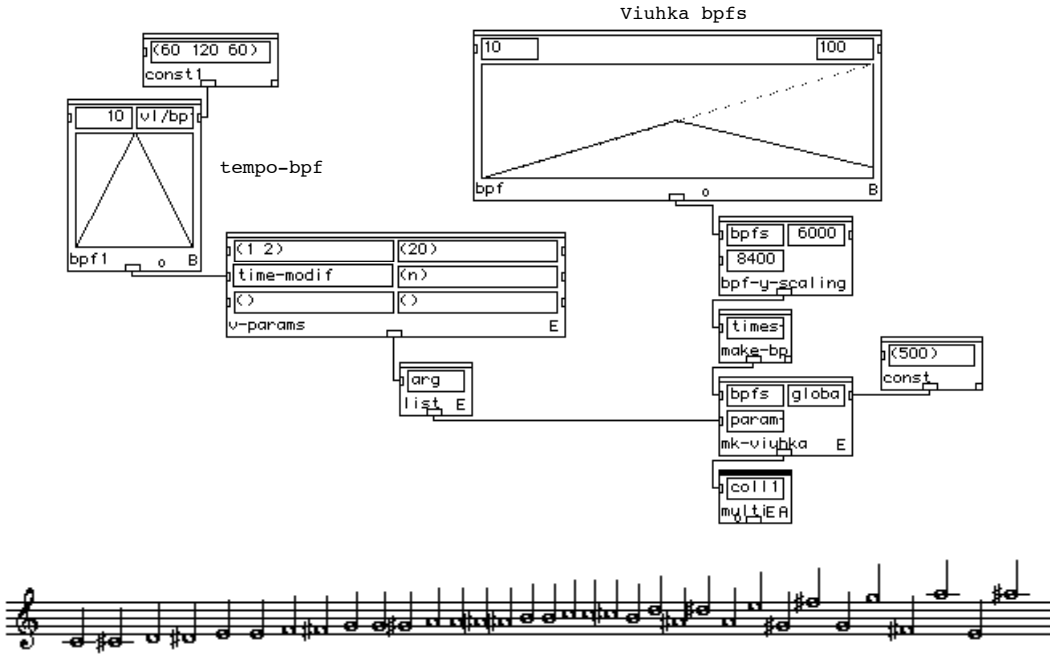


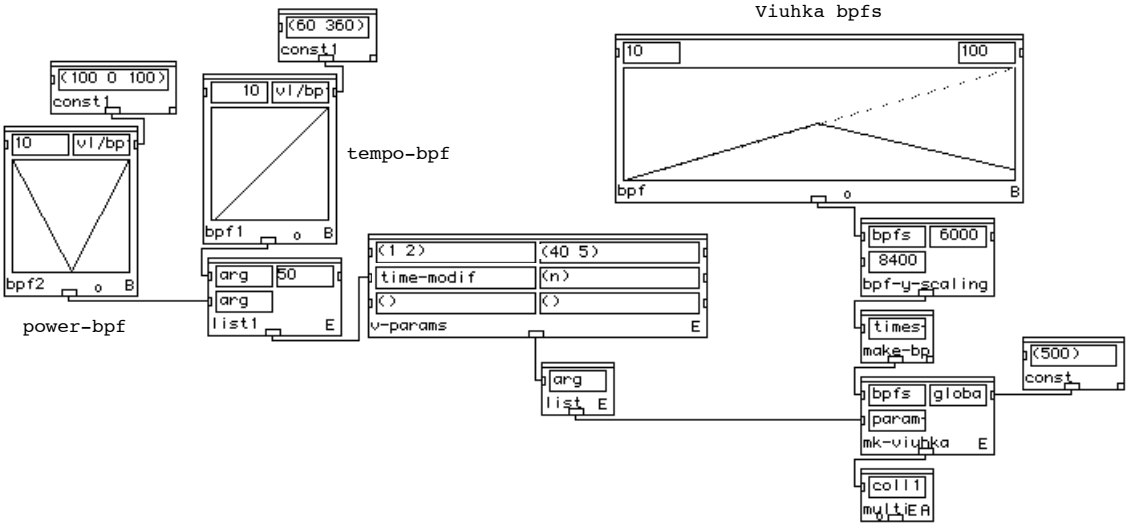
Fig.7: A tempo-bpf Viuhka patch with its result.

4.2.3.2 Contrast-Bpf

If the time-modif input is a list (consisting of a tempo-bpf, focus-value and contrast-bpf) the focus-value and contrast-bpf interact with the tempo-bpf in the following way. The

contrast-bpf[8] controls the mixture on the one hand of the delta-values modified by the tempo-bpf and on the other hand of the constant value given by the focus-value (given in ticks). If the contrast-bpf has a y-value equal to 100 then the output will be purely delta-times. If the y-value is 0 then the output will be the focus-value. Finally, y-values between 0 and 100 allow to interpolate between delta-times and the focus-value.

Figure 8 shows an example patch with a tempo-bpf, focus-value and contrast-bpf. Here the tempo-bpf makes an accelerando from 60 to 360. The contrast-bpf starts with 100 (purely delta-times), goes to 0 (only focus-value, 50 ticks) and back to 100 (purely delta-times). The result of the complex interaction of tempo-bpf and contrast-bpf can be seen in the result given below the patch. At the beginning the delta-times, (40 5), are more strongly present. In the middle part both the accelerando effect and delta-times are cancelled out by the relatively slow focus-value. In the end part the accelerando effect is very strong.



[8] Note that the y-values of the contrast-bpf are normally within the range 0 and 100.

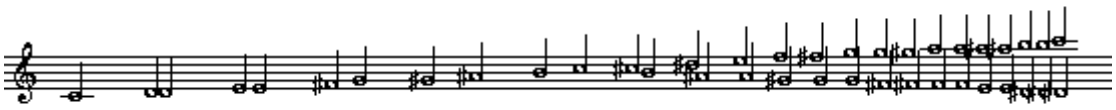


Fig.8: A Viuhka patch with a tempo-bpf, focus-value and contrast-bpf.

4.2.3.3 Normalising D-Times

The basic time modification scheme in Viuhka does not control directly the number of notes in the result. Sometimes, however, it is desirable to "normalise" or to match the d-times circ-list repetition periodicity with the duration of a segment (i.e. the result will contain *N* notes where *N* is always an exact multiple of the length of the d-times circ-list). This means that the system guarantees that the d-times circ-list is always exhausted when finishing a segment. For instance, if the length of the d-times circ-list is 4, then the result can contain 4, 8, 12, 16, 20, etc. notes. This normalising is accomplished by slightly adjusting the tempo-bpf so that the time points for tempo changes (given by the input tempo-bpf) are preserved in the result.

Figure 9 contains a patch where the d-times circ-list consists of a longer value and three shorter ones (i.e. (30 10 10 10), see the second input of the v-params box). The fourth key-flags input of the mk-viuhka box is connected to a const-box that in turn contains the list (:norm-dtime-clists t). This input forces the system to normalise (to round) the number of the notes of a segment so as to be an exact multiple of the length of the d-times circ-list. The result in the first staff below the patch shows how the d-times circ-list has been recycled exactly 11 times. For the sake of comparison the second staff contains a result without normalising d-times.

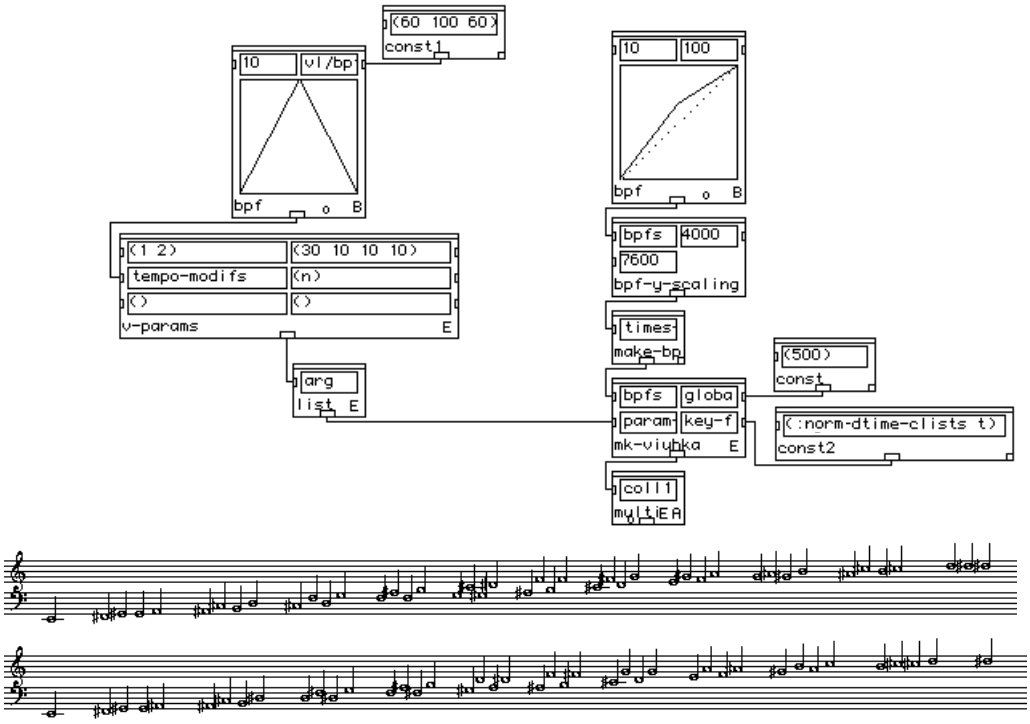


Fig.9: A Viuhka patch normalising the d-times circ-list length.

4.2.4 Ornaments

The ornaments input is a circ-list defining the ornaments to be used when building the score. There are two basic ways of indicating the type of ornaments. The first way is to use short symbols (like n, g, rn, c, etc.) that refer to relatively simple ornaments in a predefined *ornament library*. The other way is to give a pointer to a multiseq module. This module in turn evaluates its input (typically a patch) and returns its contents. This scheme is very powerful as it allows to evaluate dynamically any patch of arbitrary complexity. These kind of nested patches will be referred henceforth as *super-notes*. Often the patch to be triggered is actually also a Viuhka patch which means that any Viuhka patch can contain other Viuhka patches.

4.2.4.1 Ornaments Library

The ornament library contains the following ornaments:[9]

`n` simple-note
returns simply the unmodified main note.

`rn` rnd-note
returns the main note whose pitch is modified by a random value (by default +- 100 cents). The value of the random range can also be given by a patch (see section 4.2.7).

`g` grace-note
returns two notes, where the pitch of the first note is read from a bpf that is closest to the current bpf (or *nearest bpf*) at current global-time. The pitch of the second note is read from the current bpf. The latter note is given a time-offset value of 10 ticks (i.e. it is played 10 ticks later than the first one). The amplitude of the first note (the grace note) is 20 units lower than the amplitude of the second note.

`g+1` grace-note+1
like `g`, except the pitch of the first note is 100 cents above the pitch of the second note.

`g-1` grace-note-1
like `g`, except the pitch of the first note is 100 cents below the pitch of the second note.

`sn` simple-snake

---

[9] We give first the short symbol (usually consisting of only one or two letters) used in the circ-list followed by a longer ornament name. After this a short explanation follows describing the ornament.

returns a sequence of "snake-notes" that fills the time-span of the main note. The duration of each snake-note is 10 ticks. The pitches are chosen randomly from a scale centred around the pitch of the main note. The scale step is defined as 50 cents and the range (or ambitus) of the scale is +- 300 cents. All notes (except the note having the current midic-value) have an amplitude that is 20 units lower than that of the main note.

sg gliss-snake

like sn, except the scale is centred around a ramp starting from the current bpf and ending at the nearest bpf. The scale step is defined here as 100 cents and the range (or ambitus) of the scale is +- 400 cents.

ch chord-note

returns a chord whose pitches are derived by reading the bpfindexes circ-list *N* times where *N* is the chord density (number of notes). Note that ch is an exceptional ornament as it affects the state of the bpfindexes circ-list. The chord density is by default 2. The density value can also be given by a patch (see section 4.2.7).

c cluster

returns a cluster whose pitches are derived by filling the range from the current midic-value to the midic-value given by the nearest bpf. All cluster notes (except the note having the current midic-value) have an amplitude that is 20 units lower than that of the main note.

up upside-note

returns the main note whose pitch is transposed up by 100 cents.

dw downside-note

returns the main note whose pitch is transposed down by 100 cents.

grr graced-repeat
returns a sequence of notes filling the time-span of the main note. The pitch of the first note is 100 cents higher than the main note. The pitches of the remaining notes are equal to the pitch of the main note. The duration of each note in the sequence is 10 ticks. The amplitude of the first note is 20 units lower than that of the main note.

dv dev-note
returns the main note whose pitch is transposed by a range value (by default 100 cents). The range can also be given by a patch (see section 4.2.7).

Figure 10 gives a Viuhka patch containing all ornaments from the current ornaments library. The upper staff shows the result with ornaments. The lower staff (containing only the main notes) is given below the upper one in order to clarify how the ornaments where calculated by the system.[10]

[10] The example has one exception. In the lower staff at the main note position of the chord-note ornament we give the ornament instead of the main note. This was done in order to synchronise the bpf-indexes of the two staves (note that the chord-note ornament consumes an index for each note in the resulting chord).

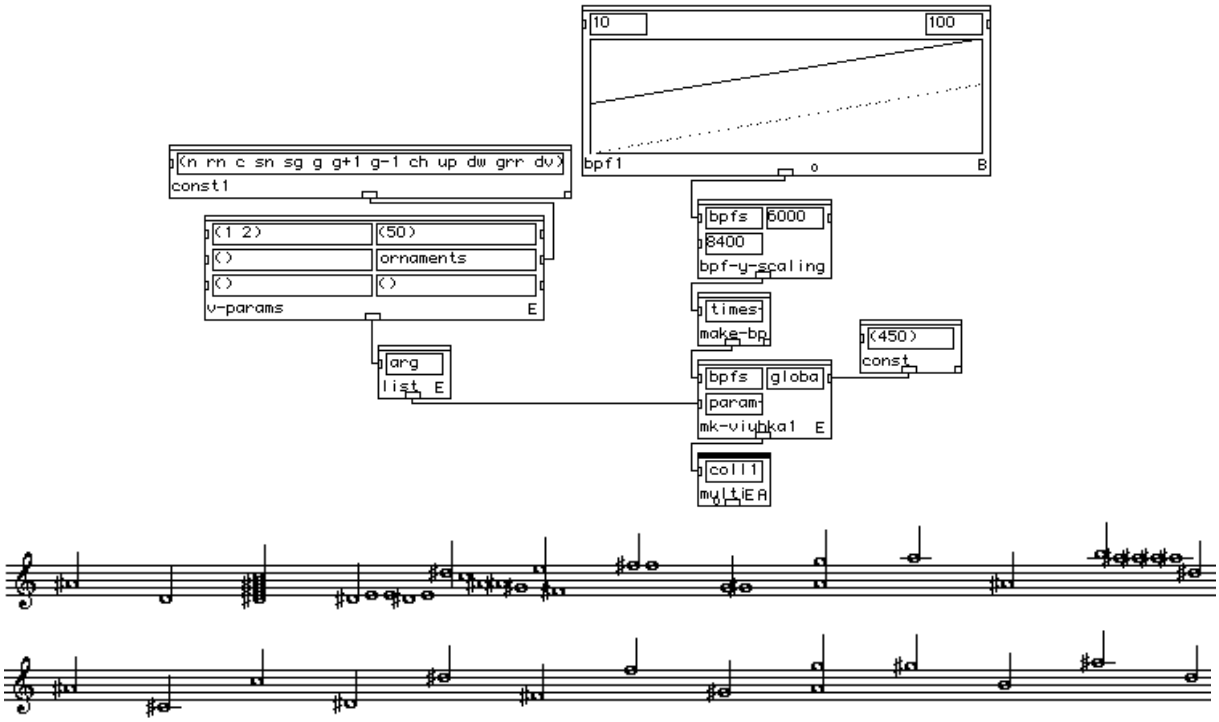


Fig.10: A Viuhka patch with the ornaments from the ornaments library.

4.2.4.2 Super-Notes

Super-notes (i.e. ornaments that are defined by a patch) are interfaced with the Viuhka main patch by a box called box-ptr (= "box pointer" box). It accepts one input which is in our case always a multiseq module.[11] It returns as a value the Clos pointer to the multiseq editor. The pointer is used in turn to retrieve the chord-line information that has been produced by the super-note patch. This means that the circ-list, defining the ornaments input of the v-params box, accepts besides symbols (indicating an ornament from the ornament library), also pointers to multiseq modules.

Typically a super-note patch needs to access information of the global state of the main note (like

[11] Actually the box-ptr box is more general as it can accept any PW box as input. We will, however, use this box here only to make Viuhka super-notes and will therefore not use it for other purposes.

pitch, duration, etc.). Therefore a super-note patch often contains evconst boxes (PW boxes that evaluate their inputs) that in turn contain names of Viuhka global variables.[12]

To clarify this let use investigate a super-note example. As our example will be more complex than the previous ones we split it into two parts: the main patch (figure 11) and the abstraction (figure 12) containing the v-params box and the super-note definition. The main patch is very similar to the ones in the previous examples, except that the params input is defined within an abstraction (see the params abstraction in figure 11):

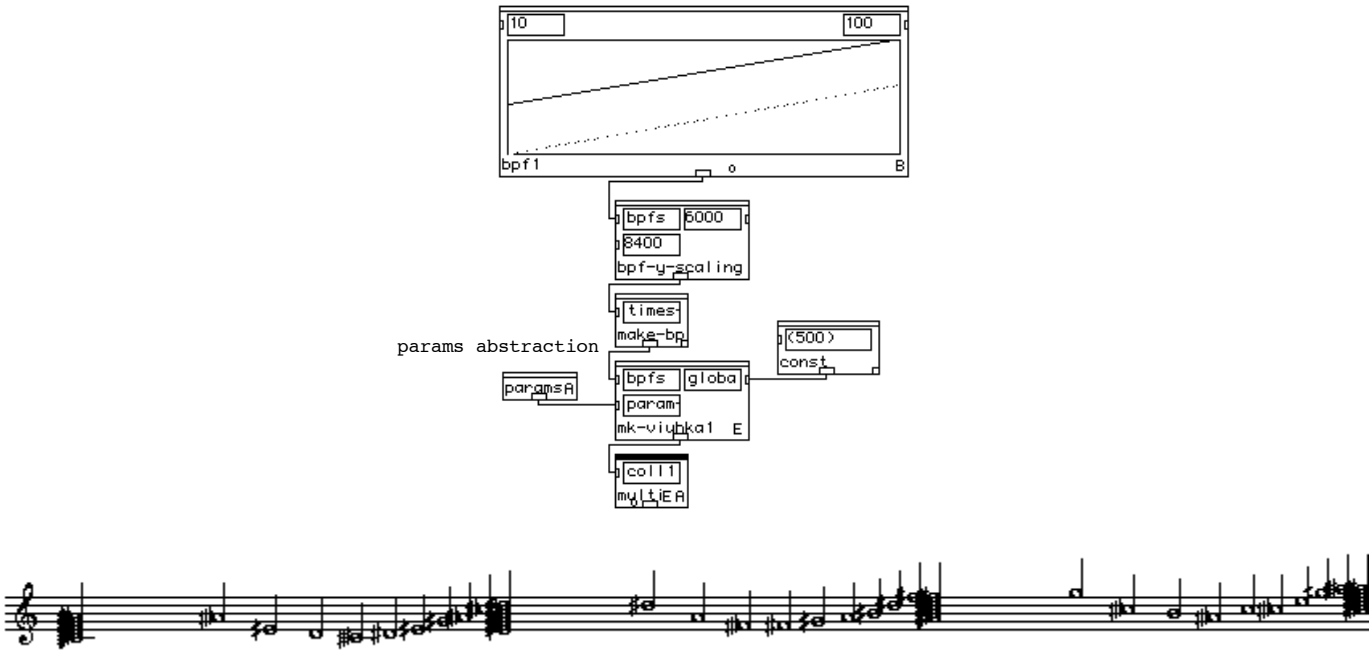


Fig.11: A super-note main patch with its result.

Figure 12 below gives the contents of the params abstraction of figure 11. Figure 12 is split into two sections (see the horizontal dotted line). The lower part contains the v-params box of the main patch. Its ornament input is connected to a list-box. Its first input, in turn, contains the symbol "c". The second input is connected to

[12] Global variables were discussed in section 4.2.

the box-ptr box of the upper part of figure 12. This means that the resulting Viuhka score (see figure 11) will alternate between a cluster (given by "c") and a super-note ornament defined in the upper-part of figure 12.

The super-note contains a mk-viuhka box whose output is connected to the box-ptr box. The inputs of the mk-viuhka box (bpfs, global-dtimes and param-ls-ls, for more information see section 3) define the resulting super-note information. In our example the bpfs input (defining the pitches of the super-note) has a bpf whose y-values are offset (with a bpf+offset box) by the global variable mypitch. The global-dtimes input (defining the duration of the super-note) is given by the global variable mydur. Finally, the param-ls-ls input is connected to a v-params box that defines the contents of the super-note. We have only "ordinary" notes (i.e. the ornament input is (n)). The actual delta-times are defined by the second input, (20), and the time-modif input (in our example we make an accelerando, see the result in figure 11).

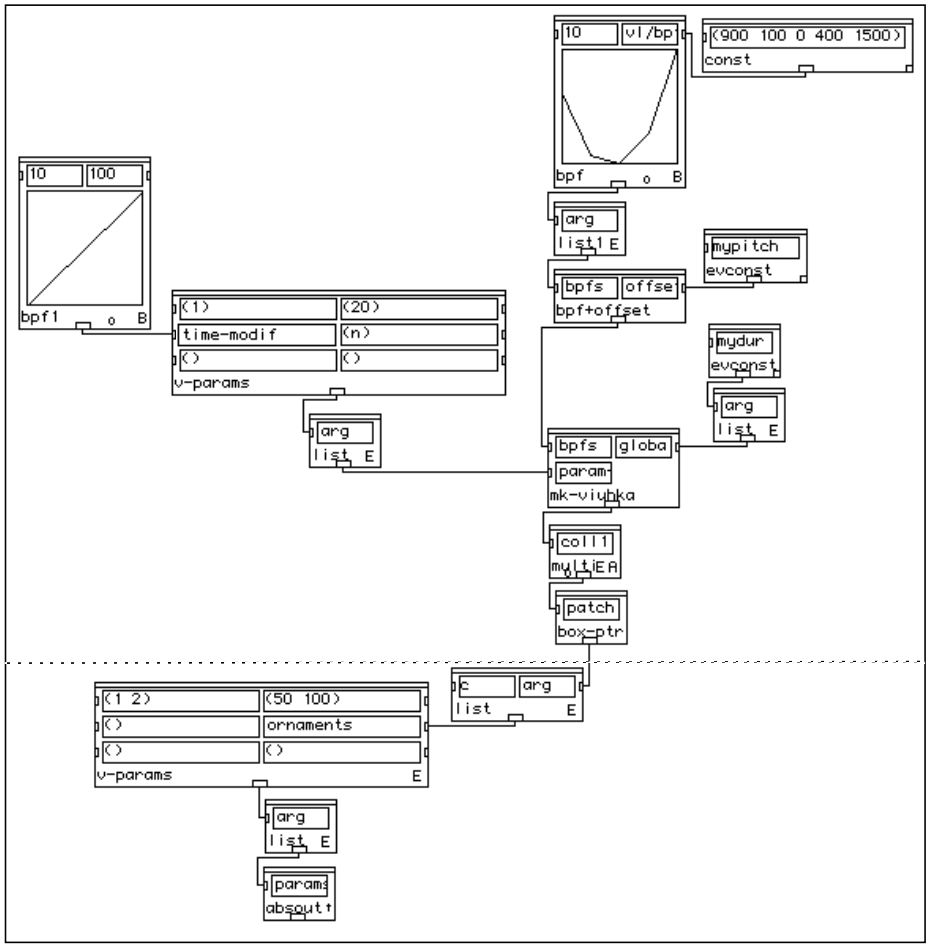


Fig.12: The super-note abstraction.

4.2.5 Csound Instrument Data

The fifth input of the v-params box, called instr, defines the "instrumental" or synthesis data to be used for a Csound score-file. The instr input is connected to a cs-params box (figure 13 below).

The cs-params box has three required inputs. The user can add inputs with an option-click (i.e. cs-params is an extended box). All cs-params inputs are circ-lists. The first three inputs (ins, artic and amp) give: (1) the instrument number for a Csound instrument; (2) the articulation of the main note; (3) the amplitude of the main note. The

instrument number must be a positive integer. The articulation, in turn, can either be an integer or a floating point value. In the former case the articulation input is considered to be a percentage value of the delta-time of the main note. In the latter case it indicates the absolute duration of the main note (given in ticks). Finally, the amplitude is given in MIDI-velocities (in the range 0 - 127).

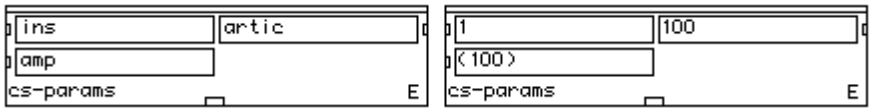


Fig.13: The cs-params box.

When writing a Csound score-file Viuhka always reserves the first five Csound instrument p-fields (P1 = instrument number, P2 start time, P3 = duration, P4 = amplitude, P5 = frequency). If the user adds inputs to the cs-params box the values of these are written as extra p-fields after the reserved ones. The meaning of these p-fields is defined by the user when designing the Csound instruments that are used in conjunction with the score-file produced by Viuhka.

In order to help the user to give time varying information the cs-params inputs accept besides numbers, circ-lists and expressions also bpf objects. For instance to make a crescendo within a segment we can simply make the following patch (figure 14, note that the amplitude input is connected directly to a bpf):

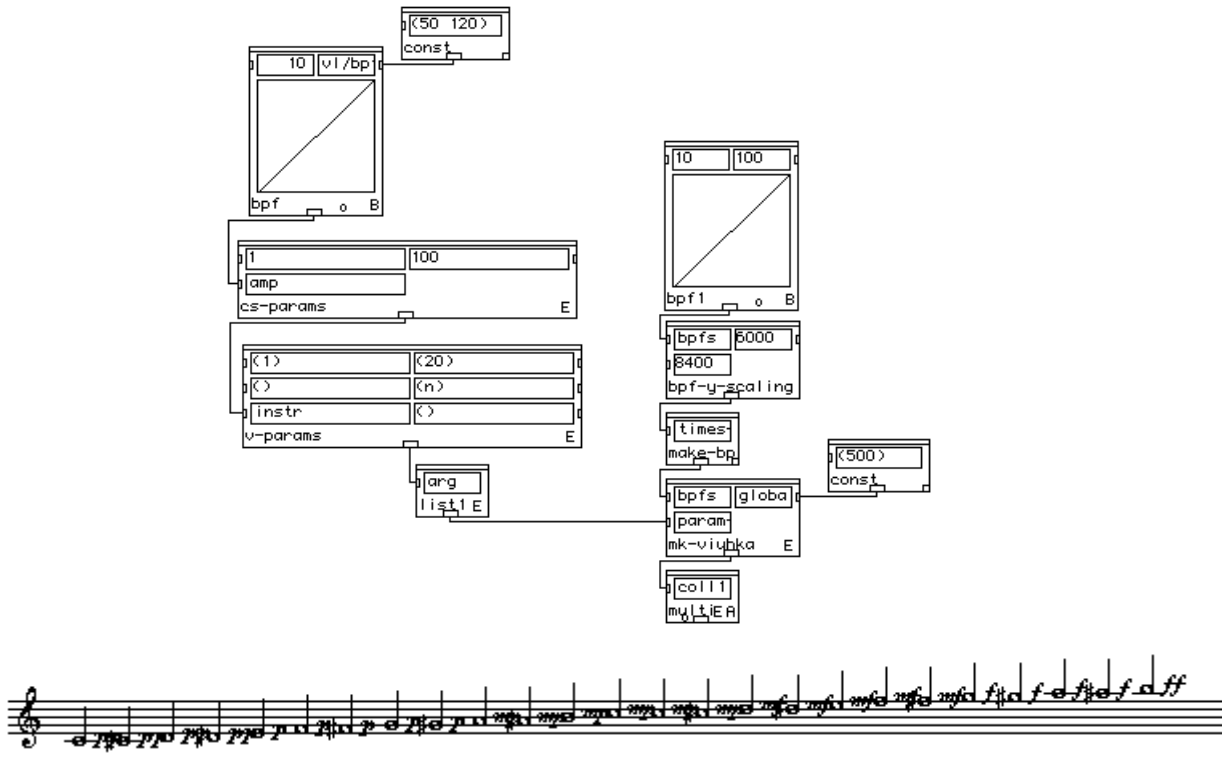


Fig.14: A crescendo cs-params patch.

Bpf objects can also be mixed freely with constants, expressions, circ-lists, etc. with the help of a box called ins-arithm (figure 15). ins-arithm has three inputs. The first one, function, must be a Lisp function (the input must be a binary function, i.e. a function that accepts two arguments). The second and third inputs, item1 and item2, can either be constants, expressions, circ-lists or bpf objects.

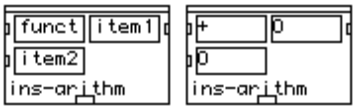


Fig.15: An ins-arithm box.

Figure 16 below shows a modified version of the crescendo patch given in figure 14. It contains an ins-arithm box that is connected to the amp input of the cs-params box. The first input of the ins-arithm box is the Lisp function +,

the second input is a ramp bpf and the third one is a sine wave bpf. The resulting amplitude evolution (the sum of the ramp and sine wave bpfs) is shown in the multi-bpf box to he right:

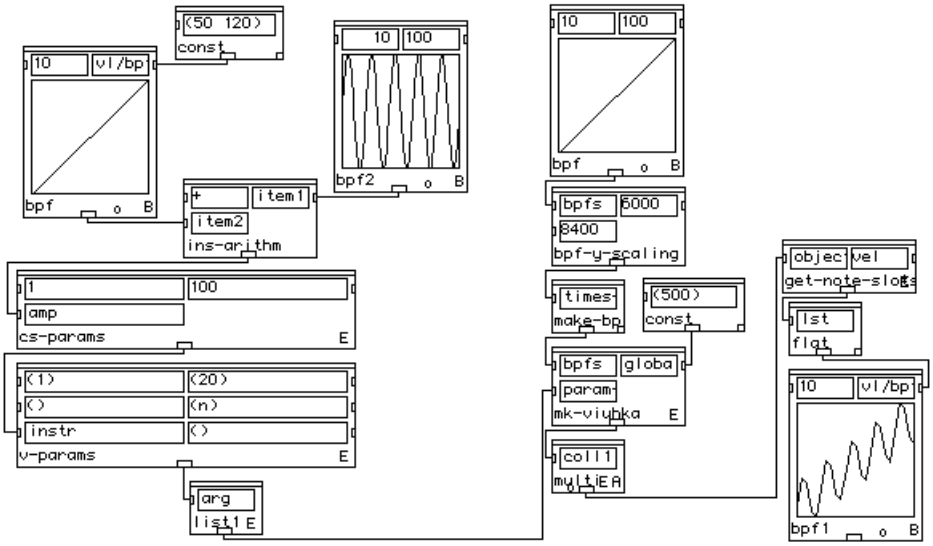


Fig.16: The crescendo patch with an ins-arithm box.

Bpf objects given to the cs-params box are by default scaled so that their x-values (time-values) match the length of the segment to which they belong to. Besides this default behaviour Viuhka allows to share cs-params information between several adjacent segments. In this case the cs-params box is connected only to the first segment (i.e. to the instr input of a v-params box). The inputs of the following adjacent segments contain the character "*" meaning that the cs-params information is shared with a previous segment connected to a cs-params box.

Figure 17 shows a two segment example where the crescendo bpf is shared between two adjacent segments (i.e. the x-values of the crescendo bpf are scaled to match the duration of the two segments). Note that the instr input of the second v-params box contains the character "*".

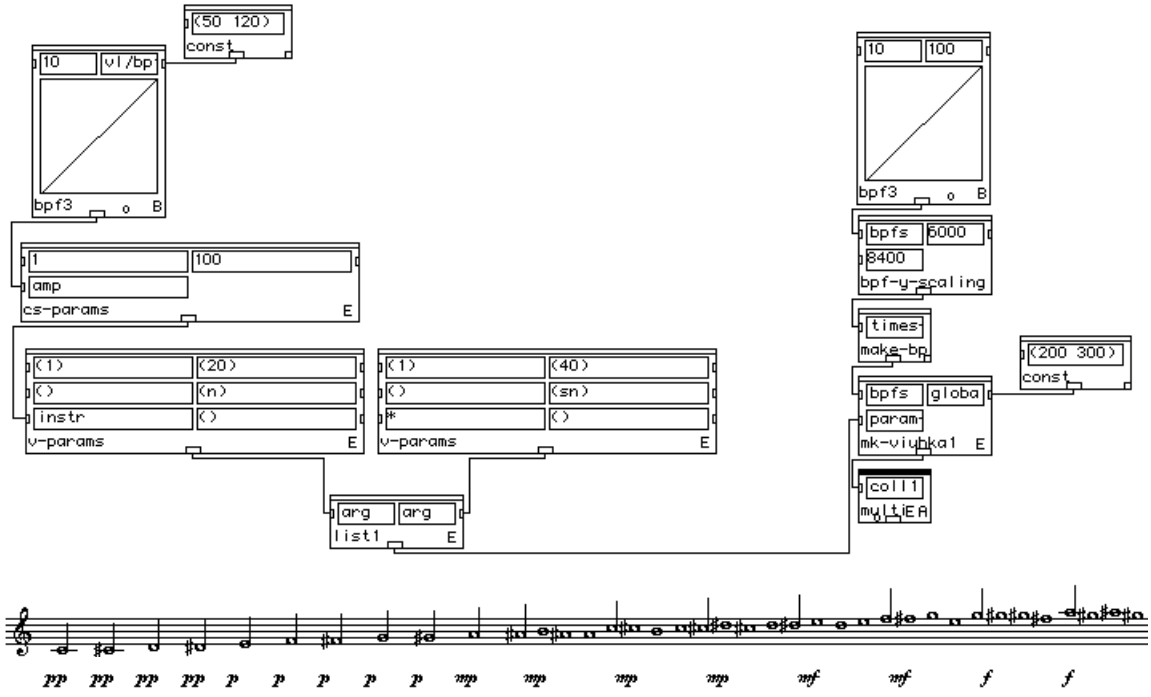


Fig.17: A two segment crescendo patch with a shared amplitude ramp bpf.

4.2.6 Previous Layer

The prev-layer input of the v-params box allows to chain several v-params boxes. Each v-params box in the chain defines a layer within a segment. The "highest" layer (the topmost staff in the score in the multiseq module) in the result is given by a box that has no connection at the prev-layer input (i.e. it is the first "link" of the chain). The box defining the second layer is connected to the prev-layer input of the first box of the chain, a box defining the third layer is connected to the second one, etc. The output of the last box in a chain (giving the lowest layer of the score) is finally connected to a list-box.[13]

For instance the patch in figure 18 below shows a chain of three v-params boxes that will thus produce three layers.

[13] The list-box is in turn connected to the param-ls-ls input of a mk-viuhka box (see section 3).

The upper box defines the first layer, the middle box the second layer and the lowest box the third layer. Finally the lowest box is connected to a list-box

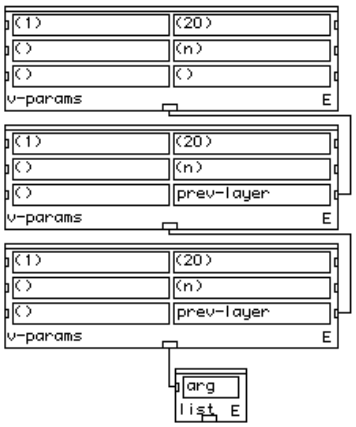


Fig.18: A three layer v-params chain.

The output of last box of a v-params chain defines a segment. All segments are collected to a list-box where the first input defines the first segment, the second input the second segment, etc.

Figure 19 contains a patch that has two segments (the list-box has two inputs). Each segment, in turn, contains three layers:

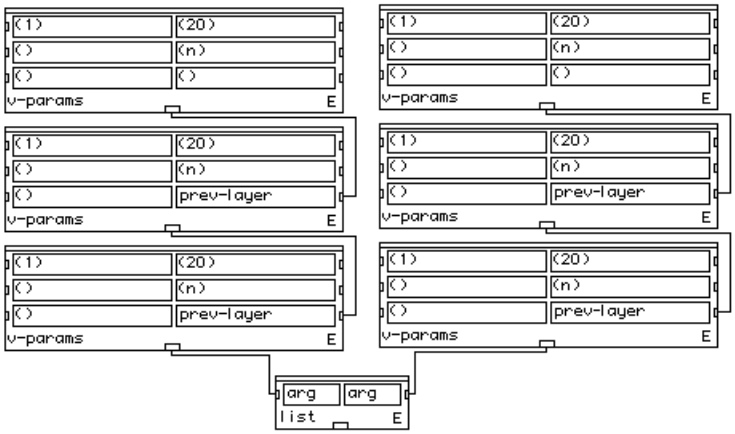


Fig.19: A patch with two segments and three layers.

The patch defining the segment and layer structure of a Viuhka patch forms a kind of a "matrix" (or *segment/layer matrix*). All segments must have an equal number of layers. This does, however, not mean that all layers within a segment must be "active" during the calculation. A layer that does not participate in the final result will be referred to as an *empty layer*.

Figure 20 gives a similar patch to the previous one except it contains two v-empty boxes. Each v-empty box (i.e. the second layer of the first segment and the third layer of the second segment) act only as a "place-holder" in the segment/layer matrix. In the final result these layers will be empty (i.e. they will contain no notes).

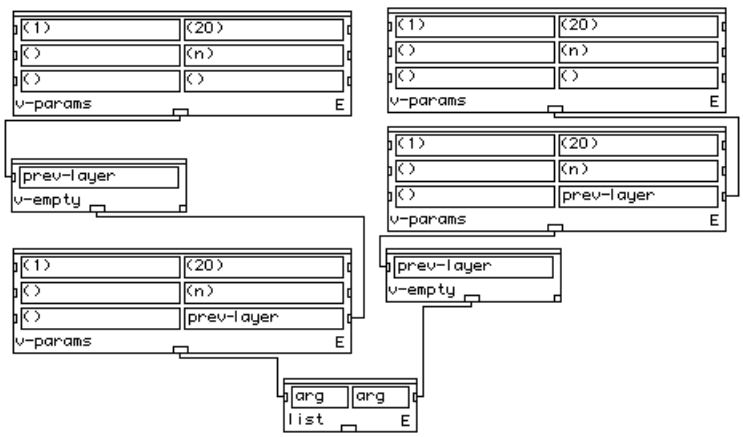


Fig.20: A patch with two segments and three layers containing v-empty boxes.

4.2.7 Ornament-Vargs

The last optional input of the v-params box, ornament-vargs, allows the user to control the behaviour of some of the predefined ornaments in the ornament library.[14] If this input is not given, i.e. it is (), the ornaments will be

[14] Note that v-params is an extended box which means that the optional inputs are not shown unless the user extends (with an alt-click) the box.

calculated according to default parameters (see section 4.2.4).

Currently the ornament library has three ornaments (rn rnd-note, ch chord-note and dv dev-note) that accept ornament-vargs information. This information will override the default behaviour of a given ornament.

The ornament-vargs information of rn (rnd-note) defines the range of the random offset (the default range is +- 100 cents) from the midic-value of the main note. In order to override the default range the user prepares a patch that contains a special *-vargs* box called rn-vargs (we use here a naming convention where the ornament name is appended to the symbol "-vargs"). This box has one input, range, that accepts circ-lists, bpfs, etc. (like the inputs of the cs-params box, see section 4.2.5). The output of the rn-args box is connected then to the ornament-vargs input.

Figure 21 gives an example where we alternate between the ornaments n (normal-note) and rn. The delta-times are modified by a tempo-bpf that produces an accelerando. The range of the rn ornament is controlled by a ramp that grows from 0 to 2400 (see the left-most bpf of figure 21):

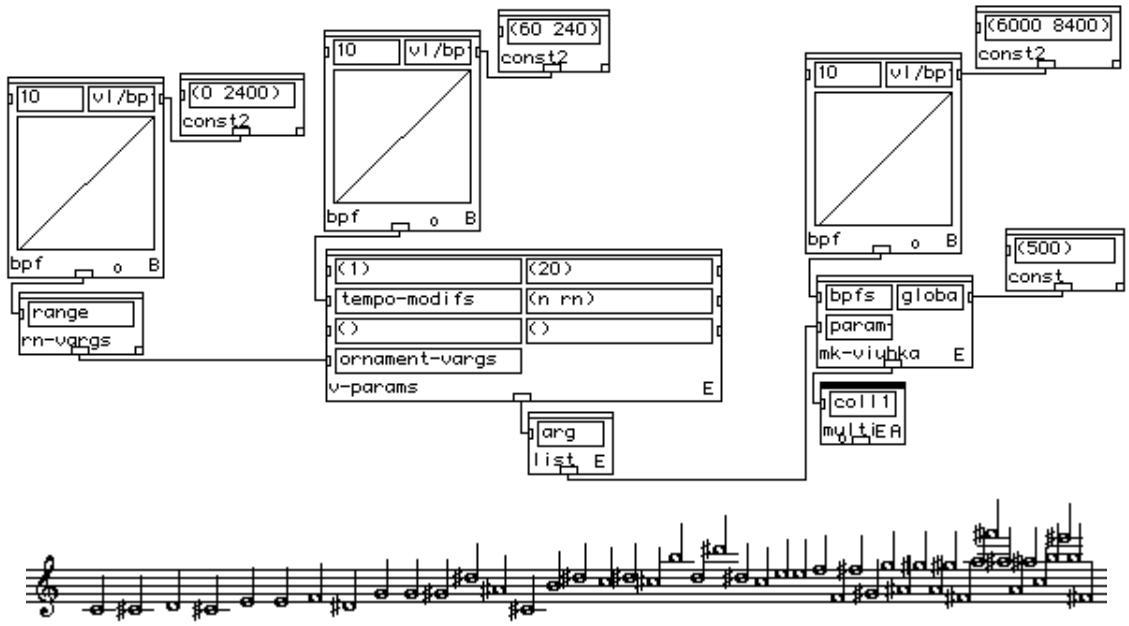


Fig.21: A patch controlling the range of the rn ornament.

The ornament-vargs information of ch (chord-note), in turn, is given by a -vargs box called ch-vargs. It has one input, density (i.e. number of notes), that accepts like the rn-vargs box bpfs, circ-lists, etc. Figure 22 below shows a patch where the density parameter is controlled by the left-most bpf:

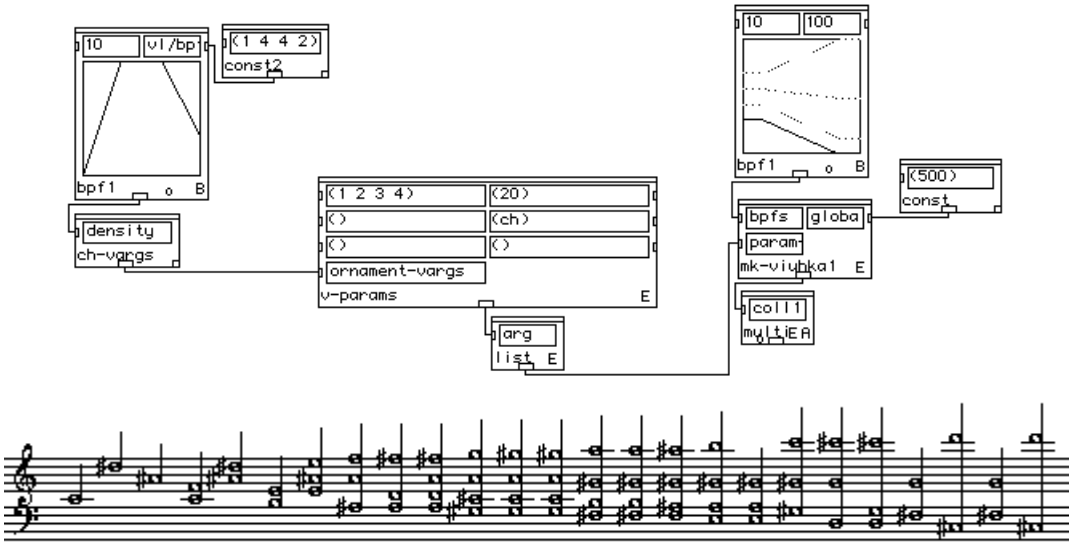


Fig.22: A patch controlling the density of the ch ornament.

Finally, the ornament-vargs information of dv (dev-note), is given by a -vargs box called dv-vargs. It has one input, range, accepting circ-lists, bpfs, etc. Figure 23 gives a patch that alternates between n and dv. The range of the dv ornaments is controlled by the left-most bpf:

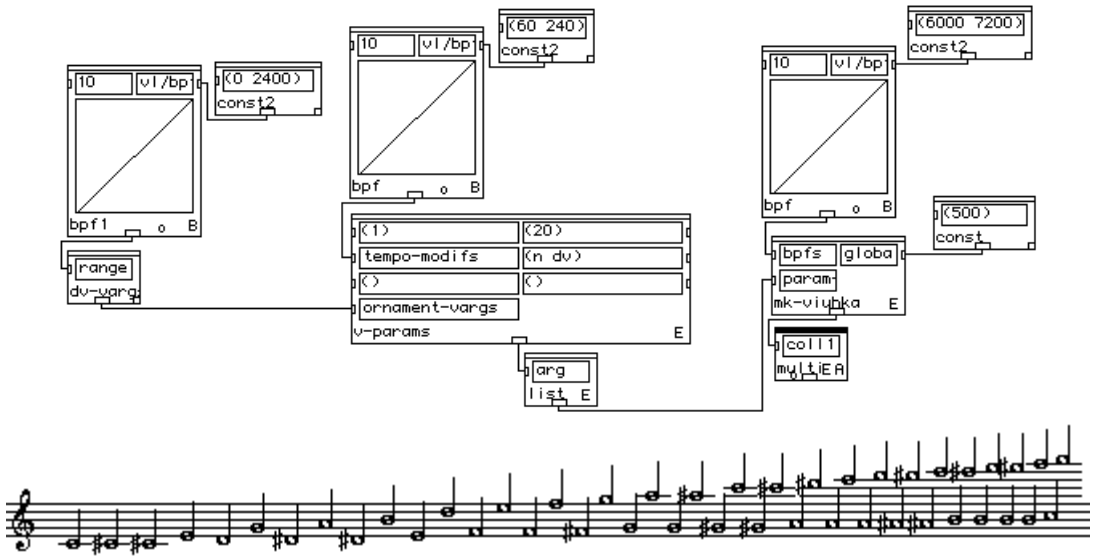


Fig.23: A patch controlling the range of the dv ornament.

The ornament-vargs input accepts also information from several -vargs boxes. In this case all -vargs boxes are collected to a list-box. The output of the list-box is then connected to the ornament-vargs input.

5. Csound Score File

Viuhka allows to create a Csound score file from a result given by a mk-viuhka box. This is done with a box called ->csound. It has one required argument, ch-lines, that accepts a list of chord-lines. The optional argument, f-header, allows the user to define information for Csound gen routines or any other information. The latter input must be a list of strings where each string represents a Csound gen routine or other score line. As section 4.2.5 already contains a detailed discussion of how Viuhka calculates synthesis information we give here only one example patch to demonstrate how the ->csound box works.

Figure 24 shows a patch that is similar to the one in figure 16 (see section 4.2.5). The patch below has, however, two additions.

First, the user has extended the cs-params box. This means that this input will be written in the score file after the reserved instrument p-fields (from p6 onwards, see for more details see section 4.2.5). The new input is connected to a patch that contains a triangle-shaped bpf which is scaled by a ins-arithm box.[15]

Second, the patch contains also a ->csound box. The first input is connected to a mk-viuhka box. mk-viuhka returns a list of chord-lines which provides the information for the instrument p-fields. The second input, f-header, is connected to a list-box having two inputs. These are in turn connected to two const-boxes which contain information for the Csound gen routines.

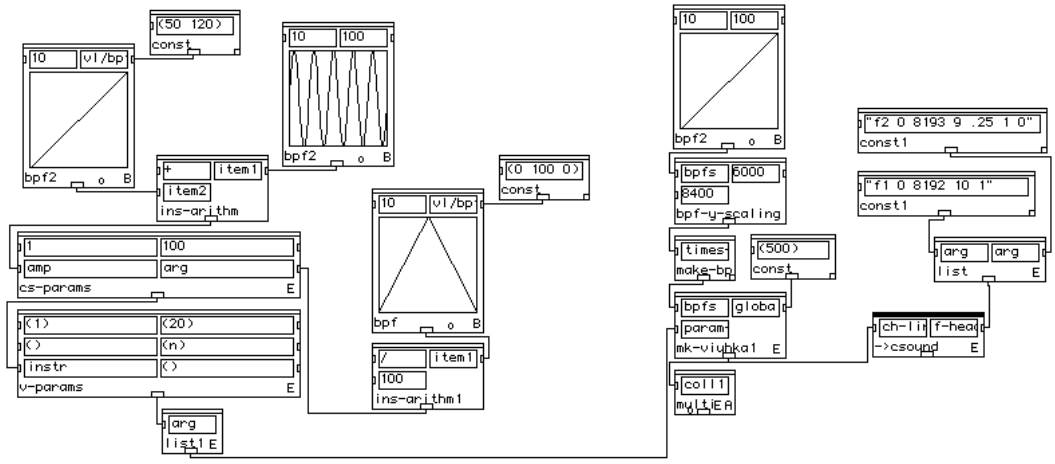


Fig.24: A patch creating a Csound score file.

When evaluated the ->csound box makes the following score file (we show due to space limitations only the beginning of the file):

```
f1 0 8192 10 1
f2 0 8193 9 .25 1 0
```

[15] The meaning of this parameter is determined in the Csound instrument definition.

```
; ins start dur amp freq
; rest

i1  0.0  0.2  50 261.63
0.0
i1  0.2  0.2  72 276.54
0.08
i1  0.4  0.2  67 292.31
0.16

.....................
```

Appendix (Reference, Tutorial, etc.)

In this section we discuss the Viuhka menu and give the documentation strings for boxes that were not discussed in the main text.

Figure 25 shows the Viuhka menu. It is split into three parts (the parts are indicated by dotted lines). The first part contains some general functions for creating and manipulating bpfs. The second part consists of the main Viuhka boxes that were discussed in detail above. Finally the third part contains two boxes. The first one, `->csound`, was discussed already in the main text (section 5). The second one, `mn-piano-roll`, is a general tool allowing to display information from a multiseq module in a "piano roll" style. This box will be discussed below.

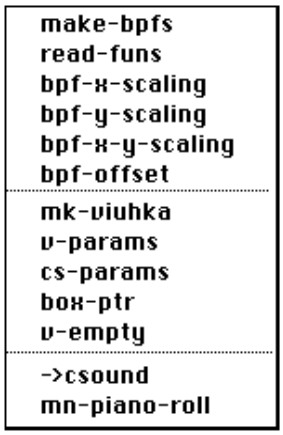


Fig.25: The Viuhka menu.

The Viuhka menu contains the following bpf functions:

patch-work::make-bpfs times+values
[function]
makes bpfs out of times+values (a list of list of lists of x-values and y-values).

patch-work::read-funs bpf-ob time
[function]
reads bpf-ob (a break-point-function) at time.

patch-work::bpf-x-scaling bpfs low high
[function]
scales bpfs (a breakpoint-function or a list of breakpoint-functions) in x-dimension between low and high. Returns a list of lists of x-points and y-points.

patch-work::bpf-y-scaling bpfs low high
[function]
scales bpfs (a breakpoint-function or a list of breakpoint-functions) in y-dimension between low and high. Returns a list of lists of x-points and y-points.

patch-work::bpf-x-y-scaling bpfs xlow xhigh ylow yhigh
[function]
scales bpfs (a breakpoint-function or a list of breakpoint-functions) both in x-dimension and y-dimension between xlow and xhigh and between ylow and yhigh. Returns a list of lists of x-points and y-points.

patch-work::bpf+offset bpfs offset
[function]
adds a constant (offset) to y-values of bpfs. Returns a bpf object.