OMTimePack Reference

for version 2.5, August 2005

Table of Contents

OMTimePack Reference

for version 2.5, August 2005

Introduction

OMTimePack (OMTP) is an OpenMusic library for creating and manipulating rhythms, which are structured as lists of durations or timepoints. Because of their stochastic nature, several OMTP functions resemble or overlap with resources in the OpenMusic kernel (including `om-random`, `perturbation`, `nth-random`, and `permut-random`) as well as other user libraries (notably OMAlea and OMLZ). However, the tools in OMTP are specialized in ways that set them apart from other OpenMusic resources and give this library a "flavor" all its own.

Version 2.5 is an advance on the previous public release of OMTP, version 1.0, in two senses. First, the main generative function, `m-rhythm` (named "marktime" in version 1.0) has gained additional features, notably target-based weighting, and interpolation between probability tables is simpler and more flexible. And second, several new functions have been added that operate on the same basic stochastic principles as `m-rhythm`: `m-empty-bars`, `m-vari-pulse`, and `m-mapped-rhythm`. While output from the original `m-rhythm` function tends to require quantization before it can be represented in conventional music notation, these new functions allow the creation of complex rhythms that remain within the limits of idiomatic metrical notation. (For a constrasting approach to this problem, see the OMRC library, which influenced the design of these new functions in OMTP.)

Users of version 1.0 may notice that certain functions (`analyze-durs`, `homogenize`) have been removed from version 2.5. It was felt that these functions were at odds with the central idea of the library, which is that complex rhythmic results can be produced by means of small event pools and simple statistical controls. (The `analyze-durs` function tended to produce large event pools and probability tables, and `homogenize` did not provide a very useful means of simplifying these parameters.)

The "m" functions

The re-naming of old functions and the addition of new ones yields a family of functions whose names begin with the prefix "m": `m-rhythm`, `m-partition`, `m-empty-bars`, `m-vari-pulse`, and `m-mapped-rhythm`. The "m" stands for Markov, since each of these functions involves a low-order Markov process, in which random selections from a pool (explicit or implicit) are controlled by a one-, two-, or three-dimensional probability table. (A larger number of dimensions can be obtained by resetting the global constant `*max-ptable-rank*`, but this may degrade the performance of various OMTP functions.) Users for whom this process is not immediately intuitive are advised to begin by exploring the `m-rhythm` function; the main *Reference* entry on this function discusses a number of simple examples. Study of the full set of tutorial patches provided with OMTP is also recommended.

Guidelines for probability table construction

A great variety of probability-table constructs are possible, almost all of which have musical potential. But the user should be alert to two "problematic" situations: loops and rows of zeros. The following table (Figure 1) includes two loops. The simpler loop results from the highlighted cell in the second row: the only event with a nonzero chance of succeeding event2 is event2 again; thus if event2 is selected once, it will be selected over and over again for the remainder of the process. A similar loop, involving event3 and event4, is caused by the highlighted cells in the third and fourth rows. In some situations the effect of a loop may be desirable (and particularly interesting results can result from interpolation between two probability tables, one of which includes loops: as the effective table evolves, the resulting rhythm will gradually become trapped in — or escape from — a loop). But the user should take care to avoid unintentional loops in a probability-table design.

	event1	event2	event3	event4
event1	0.25	0.25	0.25	0.25
event2	0	1	0	0
event3	0	0	0	1
event4	0	0	0.5	0.5

■ ***Figure 1***

A second "problematic" situation arises when a probability table contains one or more rows consisting entirely of zeros. In the example given in Figure 2, the 3×3×3 table on the left contains one such row, corresponding to the context `(event1 event3)`. Theoretically, this context should never arise: as the shaded cells indicate, each of the contexts `(event1 event1)`, `(event2 event1)`, or `(event3 event1)` has a zero chance of progressing directly to `event3`. However, the problematic context could occur in the form of `init-conds` supplied by the user or inherited from the `fin-conds` of another "m" function, in which case calculations with the given `ptable` would fail, because each possible continuation has a zero chance of being selected. To guard against this possibility, the normalization procedure performed within each "m" function replaces any row of zeros with a row of equal nonzero values in a `ptable` parameter. A typical result is illustrated on the right-hand side of Figure 2.

event1	event2	event3
4	2	0
1	1	2
0	*0*	*0*
1	0	0
0	0	2
2	1	2
0	2	0
0	1	1
0	1	0

normalization →

event1	event2	event3
2/3	1/3	0
1/4	1/4	1/2
1/3	*1/3*	*1/3*
1	0	0
0	0	1
2/5	1/5	2/5
0	1	0
0	1/2	1/2
0	1	0

■ ***Figure 2***

Generators

m-rhythm

inputs

`pool`	pool of durations/duration-patterns for random selection	list of *M* items, each item a number (representing a duration) or a list of numbers (representing a duration-pattern) or list of the form (*bpf start pool-min pool-max*)
`ptable`	statistical control on random selection (Markov process)	*n*-dimensional table structured as a depth-*n* nested list: *n*=1: list of *M* numbers *n*=2: list of *M* items, each item a list of *M* numbers *n*=3: list of *M* items, each item a list of *M* subitems, each subitem a list of *M* numbers *etc.* or list of the form (*bpf start ptable-min ptable-max*)
`tot-time`	desired duration of output rhythm	positive number
`init-conds`	context for first few random selections	list of at least $n - 1$ integers, each in the range from 0 through $M - 1$ (will tolerate a single number in place of a list containing a single number)

optional inputs

`pattern-boundary-mode`	determines format of `pattern-boundaries` output	menu selection: `note` or `note+rests`
`target-points`	specifies set of timepoints with which coincidences in ldur can be favored or disfavored	list in which each sublist is either a list of timepoints or a special list of the form (`:PERIODIC` *period begin end*)
`target-hit-weight`	multiplier for probability of pool elements that would cause coincidences with `target-points`	nonnegative number (if greater than 1, favors coincidences; if less than 1, disfavors them)
`target-distance`	maximum distance from target that counts as coincidence with target	nonnegative number (defaults to 0)

outputs

`ldur`	list of durations representing a rhythm	list of numbers
`pattern-boundaries`	list of durations representing a rhythm	list of numbers
`fin-conds`	record of context for last random selections when the process finishes	list of numbers

The `m-rhythm` function generates a rhythm in the form of a list of durations. It operates by repeatedly selecting a duration or duration-pattern at random from `pool` and appending it to the end of `ldur` until the result has a total duration of at least `tot-time`. Random selections are made according to a Markov process of low order specified in `ptable`. If `ptable` is 1-dimensional, the probabilistic weight of each event is constant (apart from the possibility of evolving probabilities described in *BPF-driven interpolation*, below). If `p-table` is *n*-dimensional (with the maximum possible value of *n* determined by the global constant `*max-ptable-rank*`), then the probabilistic weights in effect for each selection depend on the most recent (*n* − 1) selections or on `init-conds` at the beginning of the process.

The `pool` parameter may consist of a mixture of single durations and duration-patterns. Each single duration may represent a note or (when coded as a negative value) a rest. Duration-patterns are strings of durations selected and appended to `ldur` in a single step of the generative process. Suppose that four consecutive random selections during the generative process are the duration `50`, the duration-pattern `(-10 10 10)`, the duration `-25`, and the duration-pattern `(20 20)`. Then the corresponding portion of the output ldur will be `(… 50 -10 10 10 -25 20 20)`. The `pattern-boundaries` output keeps track of the individual durations and duration-patterns that constitute `ldur`. Returning to the previous example, with `pattern-boundary-mode` set to `note`, the corresponding portion of the output pattern-boundaries would be `(… 50 30 25 40)`. With `pattern-boundary-mode` set to `note+rest`, the same portion of the output would be `(… 50 10 -10 -10 25 20 20 …)`; this format, which marks the first element of each pattern with a note and the remaining elements with rests, is intended to provide a representation of pattern boundaries that will remain synchronized with `ldur` when both are quantized identically.

Parameter restrictions

1. The size of each `ptable` dimension must match the number of elements in `pool` (so a 6-element `pool` requires `ptable` to be a 6-element list, or a 6×6 table, or similar).
2. Each value in `init-conds` must be a valid index into `pool`.
3. The minimum length of the `init-conds` list is one less than the number of `ptable` dimensions; if the `init-conds` list is longer than this minimum, extra values (at the head of the list) will be ignored.

Probability table examples

(a) 1-dim table `(3/5 1/5 1/5)`

3/5	1/5	1/5

Requires `pool` with three elements. There is a 3/5 chance of selecting the first element, a 1/5 chance of selecting the second, and a 1/5 chance of selecting the third.

(b) 1-dim table `(12 4 4)`

12	4	4

Equivalent to (a): `ptable` rows need not be normalized.

(c) 2-dim table `((0.8 0.2) (0.5 0.5))`

0.8	0.2
0.5	0.5

Requires `pool` with 2 elements. If the most recent selection was the first element (or if no selections have been made yet and the last element of `init-conds` is 0), then there is an 80% chance of selecting the first

element and a 20% chance of selecting the second. If the most recent selection was the second event (or if no selections have been made yet and the last element of `init-conds` is 1), then the chances of selecting the first and second events are equal.

(d) 3-dim table `(((8 2) (5 5) ((1 0) (1 0)))`

step: $j-2$	$j-1$		
e_0	e_0	8	2
e_0	e_1	5	5
e_1	e_0	1	0
e_1	e_1	1	0

Requires `pool` with two elements, e_0 and e_1. Probabilities governing the selection to be made at step j depend on the selections already made at steps j–2 and j–1. For instance, if e_0 was selected at step j–2 and e_1 was selected at step j–1, then at step j the probabilities are 8/10 for e_0 and 2/10 for e_1.

BPF-driven interpolation

An OpenMusic **bpf** or **break-point function** consists of line segments concatenated to make a continuous curve. The `pool` or `ptable` parameters (or both) can be formatted as special four-element lists, where the first element is a bpf that is used to control interpolation between two pool or ptable parameters held in the third and fourth positions of the list. For instance, when supplied with the parameter `(`*bpf start ptable-min ptable-max*`)`, `m-rhythm` will read *bpf* at x-values corresponding to the elapsed time at each step of the generative process (taking the value *start* as the time at the beginning of the process). And it will use the associated y-values to drive interpolation between the two *ptable* parameters in the list, using values from *ptable-min* when *bpf* is at its minimum and from *ptable-max* when *bpf* is at its maximum, and interpolating between values from *ptable-min* and *ptable-max* as *bpf* ranges between these extremes. Similarly, the parameter `(`*bpf start pool-min pool-max*`)` will cause `m-rhythm` to interpolate between values drawn from *pool-min* and *pool-max*. In this case, corresponding elements in the two *pool* parameters must have the same size: if the *j*th element of *pool-min* is `(15 30)`, then the *j*th element of *pool-max* can be `(20 20)`, say, but not `(10 10 10)` or `(25)`.

Target-based weighting

The optional `target-points` and `target-hit-weight` parameters provide some control over coincidences between the resulting rhythm and a predefined set of timepoints. The `target-points` parameter specifies these timepoints; it is a list in which each sublist is either a list of timepoints or a special list of the form `(:PERIODIC` *period begin end*`)`. The latter type of sublist represents the timepoints *begin*, *begin* + (1 × *period*), *begin* + (2 × *period*), and so on.

At each step in the construction of `ldur`, `m-rhythm` tests every duration and duration-pattern in `pool`; each time it finds an element whose endpoint would coincide with a timepoint in `target-points`, it multiplies the probability of that element by `target-hit-weight`. Thus a `target-hit-weight` value greater than 1 favors coincidences with `target-points`, while a value less than 1 disfavors them. The beginning of the output `ldur` always aligns with timepoint 0.

The optional `target-distance` parameter determines the maximum distance that counts as a coincidence with a timepoint in `target-points`. At its default value of 0, this requires points to align precisely; larger values allow less precise alignment. Finally, when the `pool` includes duration-patterns (consisting of more than one duration each), target-based weighting regulates coincidences involving the *start* of these patterns, but not coincidences involving points in their *interiors*.

Like any system where different rules compete and sometimes conflict, the interaction here between the main statistical controls and target-based weighting can be hard to calibrate. If results from this interaction are unsatisfactory, it may help to adjusting the weightings closer to 1 (to favor the main statistical controls) or further from 1 (to favor the target-based weighting effect).

m-mapped-rhythm

inputs

`pool`	pool of generic durations/duration-patterns for random selection	list of *M* items, each item an integer (representing a generic duration) or a list of numbers summing to an integer (representing a generic duration-pattern)
`ptable`	statistical control on random selection (Markov process)	*n*-dimensional table structured as a depth-*n* nested list: *n*=1: list of *M* numbers *n*=2: list of *M* items, each item a list of *M* numbers *n*=3: list of *M* items, each item a list of *M* subitems, each subitem a list of *M* numbers *etc.* or list of the form (*bpf start ptable-min ptable-max*)
`vari-pulse`	list of specific durations representing variable pulse onto which generic durations will be mapped	list of positive numbers
`tot-time`	desired duration of output rhythm	positive number
`init-conds`	context for first few random selections	list of at least $n - 1$ integers, each in the range 0 … $(M - 1)$

optional inputs

`offset`	point beyond start of `vari-pulse` at which the generated rhythm will begin	nonnegative number (defaults to zero)
`pattern-boundary-mode`	determines format of `pattern-boundaries` output	menu selection: `note` or `note+rests`

outputs

`ldur`	list of durations representing a rhythm	list of numbers
`pattern-boundaries`	list of durations representing a rhythm	list of numbers
`fin-conds`	record of context for last random selections when the process finishes	list of numbers

Generates a rhythm by mapping randomly-selected generic durations from `pool` onto a variable pulse `vari-pulse`. The mapping works as follows: if the selected generic duration has integer value *n*, then the next *n* elements of `vari-pulse` will be grouped together to form an element of the resulting rhythm `ldur`. The restriction of generic durations to integer values is meant to ensure that the resulting rhythm is compatible with the same metrical frameworks as the variable pulse onto which it was mapped. Note that fractional values may be incorporated in a duration pattern, as long as the total duration of the pattern is an integer value.

Selections from the `pool` are handled in the same manner as for `m-rhythm`, and bpf-driven interpolation is possible on the `ptable` (but not the `pool`) parameter.

The `vari-pulse` parameter can be any list of durations, but the usual assumption is that consecutive elements should be nearly equal: given the duration *j* and its successor *k*, *k* should resemble *j* more nearly than it resembles some integer multiple or division of *j*. In fact, the expected structure for `vari-pulse` is the type created by the function `m-vari-pulse`. If a nonzero `offset` is specified, then `ldur` will be padded at the beginning with a rest spanning a suitable number of `vari-pulse` elements to approximate `offset`.

m-empty-bars

inputs

`pool`	pool of time signatures representing empty bars	list of *M* items, each item a pair of positive integers
`ptable`	statistical control on random selection (Markov process)	*n*-dimensional table structured as a depth-*n* nested list: *n*=1: list of *M* numbers *n*=2: list of *M* items, each item a list of *M* numbers *n*=3: list of *M* items, each item a list of *M* subitems, each subitem a list of *M* numbers *etc.* or list of the form (*bpf start ptable-min ptable-max*)
`tot-time`	desired duration of output `empty-bars`	positive number
`init-conds`	context for first few random selections	list of at least $n - 1$ integers, each in the range $0 \ldots (M - 1)$

outputs

`empty-bars`	list of time signatures representing empty bars	list of pairs of positive integers
`fin-conds`	record of context for last random selections when the process finishes	list of numbers

Generates a list of empty bars. The length of each bar is determined by a time signature, which takes the form of a pair (*numer denom*). The minimal requirement is for *numer* and *denom* to be positive integers; normally, *denom* will be a small power of 2, as this is the usual constraint on the denominator of a time signature. The resulting list of empty bars is intended primarily to serve as an input to `m-vari-pulse`.

m-vari-pulse

inputs

`pulse-map`	map that associates one or more pulses with each time signature that may be seen in `empty-bars`	list of sublists in which the head is a pair of positive integers (*numer denom*) representing a time signature, and the tail contains *M* pulse lists, each consisting of numbers whose absolute values sum to the ratio *numer*/*denom*
`ptable`	statistical control on random selection (Markov process)	*n*-dimensional table structured as a depth-*n* nested list: *n*=1: list of *M* positive numbers *n*=2: list of *M* items, each item a list of *M* positive numbers *n*=3: list of *M* items, each item a list of *M* subitems, each subitem a list of *M* positive numbers *etc.* or list of the form (*bpf start ptable-min ptable-max*)
`empty-bars`	list of time-signatures, each determining the length of an empty bar	list of pairs of positive integers
`init-conds`	context for first few random selections	list of at least $n - 1$ integers, each in the range $0 \ldots (M - 1)$

outputs

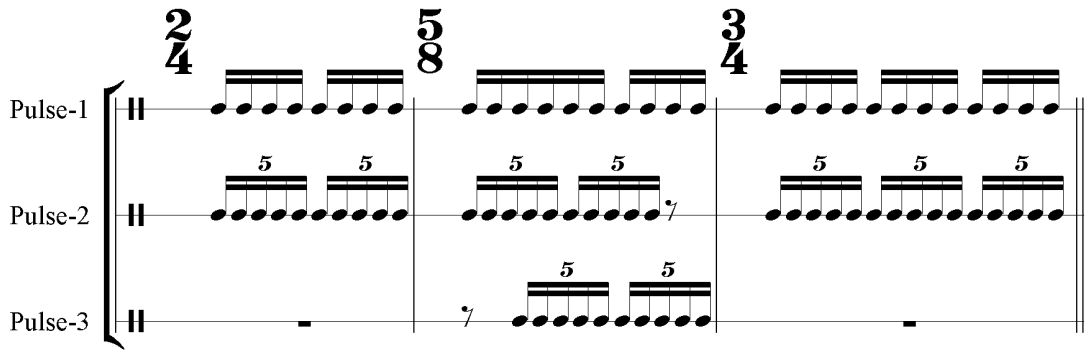
`ldur`	list of durations representing a variable pulse	list of positive numbers
`fin-conds`	record of context for last random selections when the process finishes	list of numbers

Generates a randomly varying pulse by selectively tracking the pulses available in `pulse-map`, which is structured as as shown in Figure 3. There are two required conditions and one expected (but not required) condition for each of the pulses in `pulse-map`. The first requirement is that if a rest or group of rests occurs, the beginning of the rest or group of them must coincide with the onset of a note in another pulse. The second requirement is that, if a pulse contains no rests, than at least one of its note onsets must coincide with a note onset in another pulse. It is expected, furthermore, that each pulse will consist mainly of a succession of equal durations, consistent with the usual concept of pulse; rests typically occur in cases where the measure length cannot be evenly divided by a tuplet-based pulse.

```
((time-sig-1 pulse-1 pulse-2 … pulse-M) (time-sig-2 pulse-1 pulse-2 … pulse-M) …)
```

■ ***Figure 3***

Figure 4 gives a simple example of the structure, depicted in conventional music notation and in the format required for the `pulse-map` parameter. (*Tip:* it can be helpful to begin by sketching a pulse map in conventional music notation and translating it afterwards into list form for use as an input to `m-vari-pulse`.)

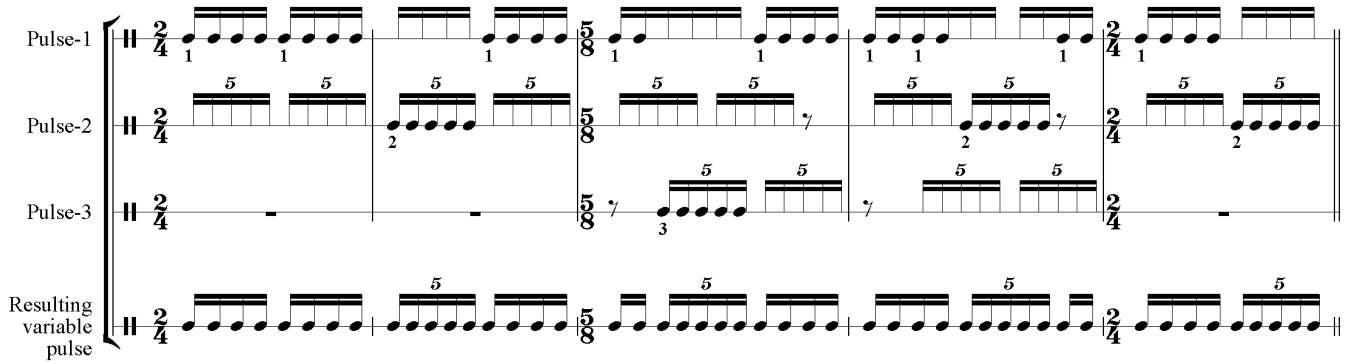



```
(((2 4)
  (1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16)
  (1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20)
  (-1/2))
 ((5 8)
  (1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16)
  (1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 -1/8)
  (-1/8 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20))
 ((3 4)
  (1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16 1/16)
  (1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20 1/20)
  (-1/2)))
```

■ ***Figure 4***

In Figure 4, there are three pulses, and `m-vari-pulse` will track them selectively to construct a variable pulse. Each of the measures found in `empty-bars` is expected to have time signature 2/4, 5/8, or 3/4. Suppose the first measure is 2/4. The `ptable` parameter determines a weighting for Pulses 1, 2, and 3 at the beginning of this bar, but Pulse 3 is void at this point, so tracking will begin with Pulse 1 or 2. The same pulse continues to be tracked until one of two conditions arises: (1) the current pulse becomes void; (2) an element of the current pulse coincides with elements of one or more of the other pulses. Each time either of these conditions is met, `pulse-map` is scanned to determine which pulses contain an onset at the current time, and one of these pulses is selected at random to be tracked next. An illustration of this process is shown in Figure 5, with `pulse-map` as in Figure 4, and with `empty-bars` = `((2 4) (2 4) (5 8) (5 8) (2 4))`. For simplicity, `ptable` is not specified.

Unlike the parameter of the same name in other TimePack functions, the `ptable` parameter in `m-vari-pulse` is restricted to *nonzero* probabilistic weights. This ensures that each time the conditions for random selection are met, at least one pulse will be available for selection. (Zero-values entered by the user will be converted internally to very small nonzero values.)



■ ***Figure 5***

Transformations

conform

inputs

`in-points`	list of timepoints representing a rhythm	list of numbers in ascending order
`to-points`	list of timepoints representing a rhythm	list of numbers in ascending order

optional inputs

`lim`	limit on deforming effects	number

outputs

`out-points`	list of timepoints representing a rhythm	list of numbers in ascending order

Returns a list `out-points` in which each point in `in-points` is replaced by its best approximation from `to-points`. The result is typically a deformation of `in-points` as its contents are made to conform to some subset of the contents of `to-points`.

The purpose of the optional parameter `lim` is to limit the amount of deformation. The absolute value of `lim` establishes a maximum distance that a point *p* can be displaced by `conform`. When this distance would be exceeded, `conform` instead sends *p* to `out-points` unchanged (if `lim` is nonnegative) or witholds *p* from `out-points` altogether (if `lim` is negative).

Examples

```
(conform '(0 5 15 45) '(1 14 18 70)) ==> (1 1 14 70)
(conform '(0 5 15 45) '(1 14 18 70) 10) ==> (1 1 14 45)
(conform '(0 5 15 45) '(1 14 18 70) -2) ==> (1 14)
```

m-partition

inputs

`timepoints`	list of timepoints representing a rhythm	list of numbers in ascending order
`ptable`	statistical control on random selection (Markov process)	*n*-dimensional table structured as a depth-*n* nested list: *n*=1: list of *M* numbers *n*=2: list of *M* items, each item a list of *M* numbers *n*=3: list of *M* items, each item a list of *M* subitems, each subitem a list of *M* numbers *etc.* or list of the form (*bpf ptable-min ptable-max*)
`init-conds`	context for first few random selections	list of at least *n* – 1 integers, each in the range from 0 through *M* – 1 (will tolerate a single number in place of a list containing a single number)

outputs

`subrhythms`	*M* lists of timepoints representing rhythms	list of *M* lists of numbers

The `m-partition` function partitions a rhythm into two or more subrhythms by sending each timepoint to a randomly selected destination. Random selections are made according to a Markov process of low order specified in `ptable`. If `ptable` is 1-dimensional, the probabilistic weight of each destination is constant (apart from the possibility of evolving probabilities, which operate as in the `m-rhythm` function). If `ptable` is *n*-dimensional, the probabilistic weights in effect for each selection depend on the most recent $(n - 1)$ selections, or on `init-conds` at the beginning of the process. The number of destinations (sublists in the output `subrhythms`) is determined by the shape and size of `ptable`: an $M \times M \times \ldots \times M$ table implies *M* destinations.

Interpolation between two probability tables is accomplished as in the `m-rhythm` function. To partition a list of durations rather than timepoints, begin by processing the list with OpenMusic's `x->dx` function.

Utilities

timepoints->density

inputs

`timepoints`	list of timepoints representing a rhythm	list of numbers in ascending order
`framesize`	density is measure in timepoints per frame	positive number
`tmin`	timepoints at which density measurement begins	number
`tmax`	timepoint at which density measurement ends	number

outputs

`x-points`	starting timepoints of successive frames	list of numbers
`y-points`	each element counts timepoints that occur within frame beginning at corresponding element of `x-points`	list of nonnegative integers

Counts the number of timepoints in successive frames of size `framesize`, starting at `tmin` and continuing until `tmax` is reached or exceeded. Returns two lists: `x-points` lists the startpoints of successive frames, and `y-points` counts the timepoints in each frame (on or after the startpoint, and before the next frame's startpoint). These lists can be connect to the `bpf-factory` inlets of the same names to create a bpf that plots the fluctuating density of `timepoints` per frame.

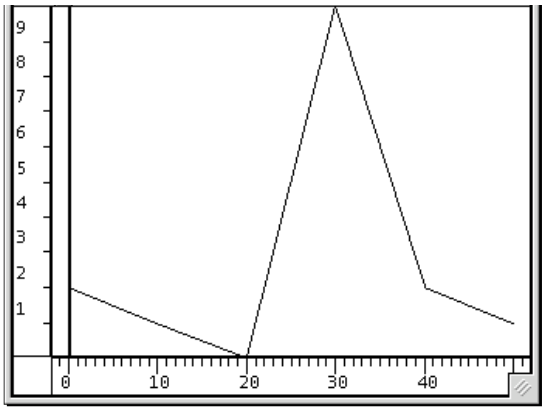
One suggested use of this function is to generate a new rhythm whose most active regions occur during periods of inactivity in an existing rhythm. To achieve this result:

1. if the existing rhythm is a list of durations, convert it to a list of timepoints using `dx->x`
2. process the timepoint list with `timepoints->density`
3. wire the resulting `x-points` and `y-points` to a `bpf-factory` to produce a bpf representation of the original rhythm's density
4. set up a rhythm-generating patch using (for example) `m-rhythm`, and use the bpf from step (4) to control interpolation between a ptable describing more active behavior (for *ptable-min*) and one describing less active behavior (for *ptable-max*)

Example

```
(timepoints->density    '(0 5 10 30 31 32 33 34 35 36 37 38 39 40 45 50 60 70)
                        10
                        0
                        50)

==>  (0 10 20 30 40 50)
     (2 1 0 10 2 1)
```



resize-ptable

inputs

`in-table`	probability table	list of *M* numbers; or list of *M* items, each item a list of *M* numbers
`pieces`	description of how to resize `in-table`	list of pairs; each pair a list of two positive integers

outputs

`out-table`	probability table	list of *N* numbers; or list of *N* items, each item a list of *N* numbers

Allows a probability table to be used with an event space of a different size and/or structure than the one for which it was originally designed.

Tries to express the shape of `in-table` in a new probability table related to the original in a way expressed by `pieces`. The first pair (*in*$_1$ *out*$_1$) in `pieces` means "express the first *in*$_1$ rows and columns of `in-table` in the first *out*$_1$ rows and columns of `out-table`", and successive pairs control the processing of successive rows and columns. For example, the `pieces` list `((7 8))` takes a 7×7 `in-table` and produces an 8×8 `out-table`; the `pieces` list `((4 4) (3 4))` does the same thing but concentrates the change in the last three columns of `in-table`.

In the `pieces` list, the sum of the first elements of all the pairs must equal the size *M* of each `in-table` dimension, and the sum of the second elements of all the pairs determines the size *N* of each `out-table` dimension.

thru

inputs

`in`	input data	any value

outputs

`out`	input data unchanged	same value

Passes data through unchanged. Intended to facilitate layout of a complex patch onscreen.

gm-pgmout

inputs

pgm	General MIDI patch name	menu selection (standard 128 patch names)
chan	MIDI channel	1–16

Sends MIDI program change to patch pgm on channel chan.

Constants

max-pool-size

The pool of durations/duration-patterns from which random selections are made is limited to this size at maximum (default value: 64). Changing this value is not recommended.

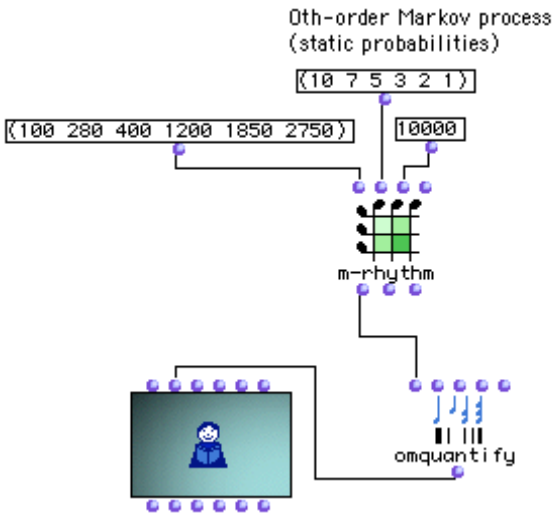
max-ptable-rank

The largest-order Markov process available is one less than this limit (default value: 3). Changing this value is not recommended.

Tutorial patches

m-rhythm basics

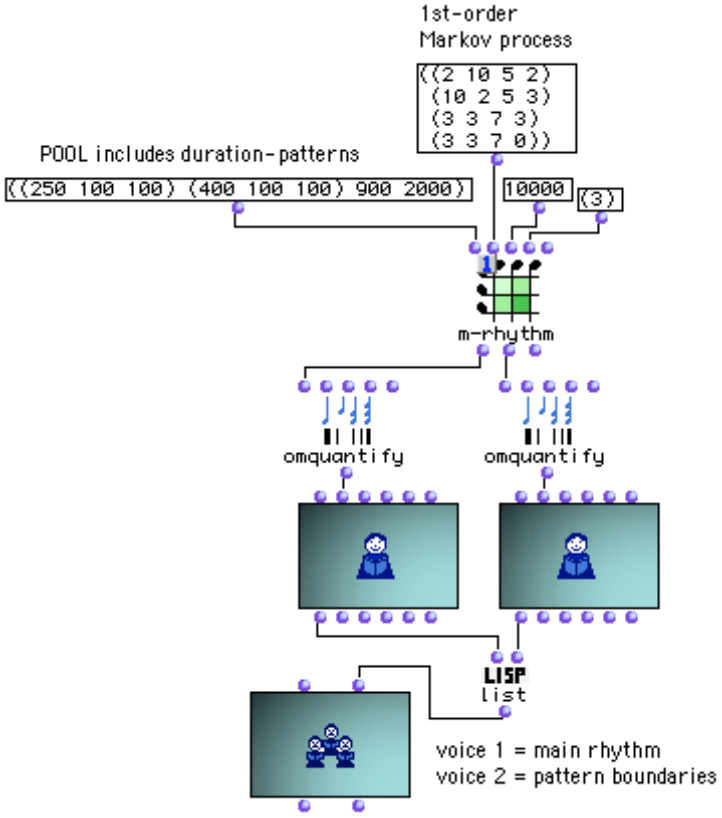
This patch contains two demonstrations of basic uses of the `m-rhythm` function. On the left-hand side (Figure 6), `m-rhythm` is used on a pool of six durations ranging from `100` to `2750`. (The units depend on context, but it's reasonable to assume these are milliseconds.) Each duration is assigned a fixed probabilistic weight: the duration `100` has a weight of $\frac{10}{10 + 7 + 5 + 3 + 2 + 1} = 0.357$, and so on. The `m-rhythm` object will make random selections from the `pool` repeatedly, appending them to the list returned at the first outlet, until the accumulated duration is at least `10000` milliseconds. The resulting list of durations is then converted by `omquantify` into a rhythm tree and sent to a `voice` factory for display in conventional music notation. Because `m-rhythm` is based on a random process, each time you evaluate the `voice` factory you will see a different (but statistically similar) rhythmic result.



■ ***Figure 6***

The right-hand side of the same patch shows `m-rhythm` used with more complicated parameters (Figure 7). In this case, the pool consists of four elements, two of which are duration-patterns. The `ptable` is structured as a 4×4 table, so each random selection influences the probabilities that will be used at the next step. For instance, if the last element, `2000`, is chosen at step *n*, then at step *n*+1 the four elements will have probabilities of 0.231, 0.231, 0.538, and 0, as determined by the last row of `ptable`. Since duration-patterns are used, it is interesting to trace individual durations to the patterns that include them, so we wire the second outlet (which reports pattern boundaries) to an additional `omquantify` object and `voice` factory and then combine the two voices in a `poly` object. Because the `m-rhythm` object feeds two subsequent objects in this example, it is put into "1" mode.[1] Otherwise the two objects would receive different randomly generated results, and the pattern boundaries captured in voice 2 would not match up to the content of voice 1.

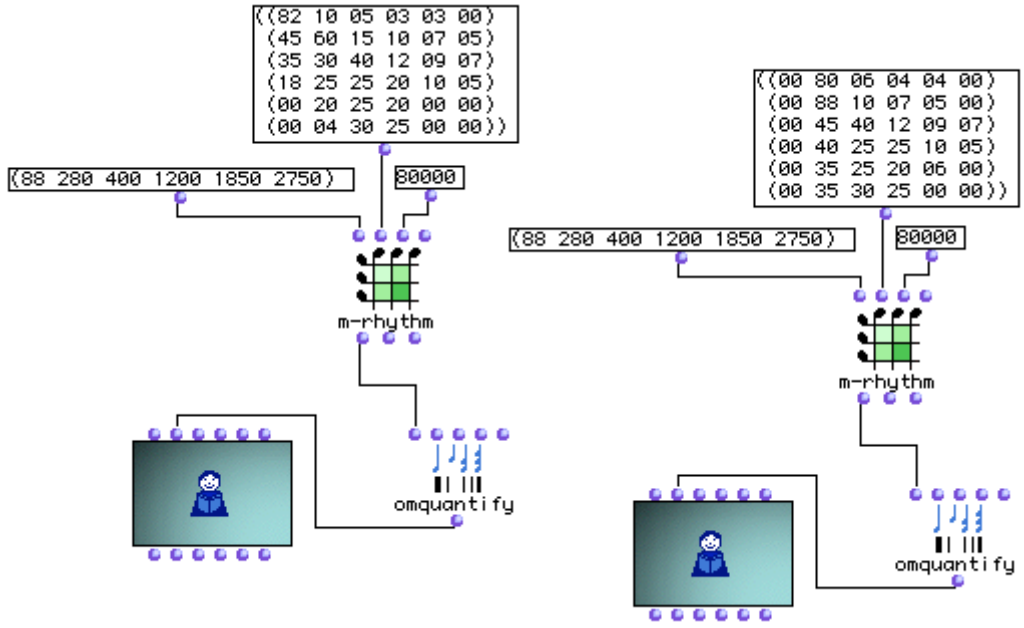
[1] To access this mode: select the `m-rhythm` object, press "b" to get a small box marked "X," and click that box once to switch it to "1."



■ *Figure 7*

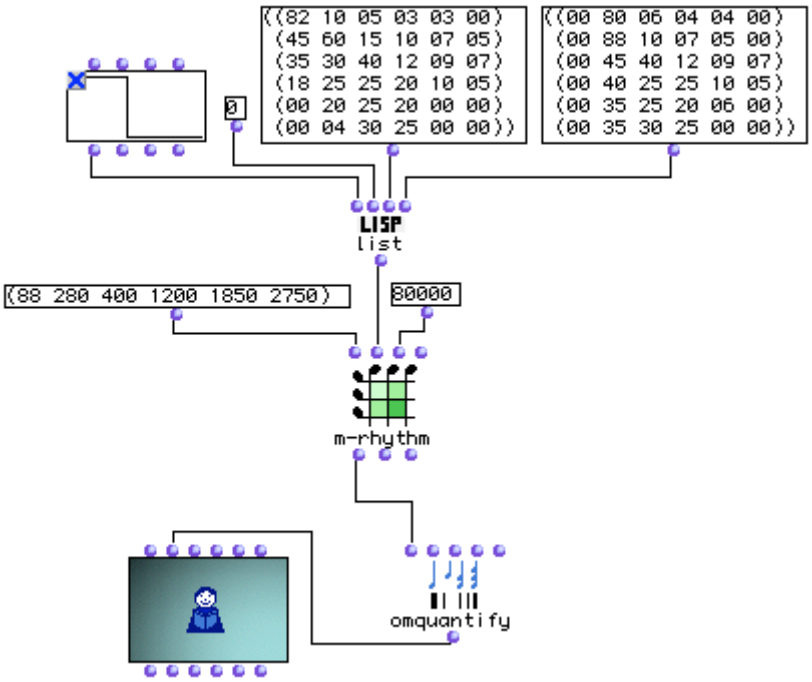
m-rhythm interpolation

There are three parts to this patch. The first two (both given in Figure 8) show `m-rhythm` used with the same `pool` but two different `ptable` parameters.



■ *Figure 8*

In the third (rightmost) part of this patch (shown in Figure 9), the same `pool` is used once more. The `ptable` this time is a more complex structure, combining both of the preceding `ptable` parameters, with interpolation between the two driven by a break-point function (bpf). The bpf begins at its maximum value and drops almost instantaneously to its minimum value. Thus random selections will be controlled by the probabilities in the second table initially, and by the first table further on. If you study typical outputs from parts one and two of this patch, you will be able to recognize the shift from one characteristic to the other partway through the output from this third part of the patch.

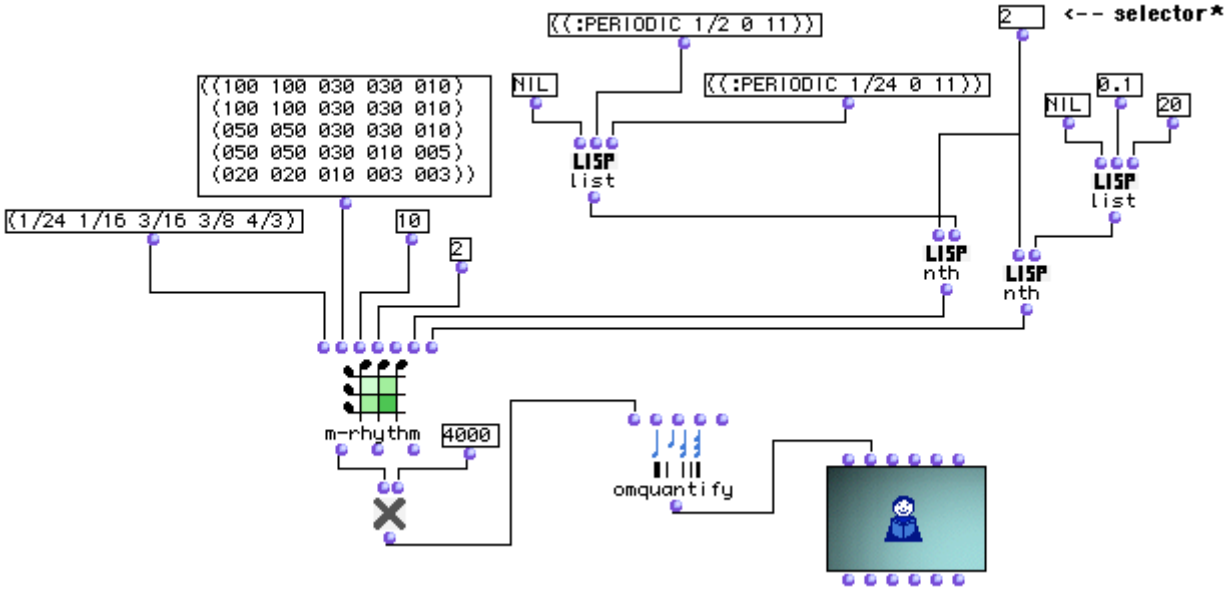


■ ***Figure 9***

m-rhythm target

This patch (Figure 10) demonstrates some of the effects possible with the target-based weighting feature of `m-rhythm`. In addition to the usual connections to the first four inlets of `m-rhythm`, connections are made here to the `target-points` and `target-hit-weight` (sixth and seventh) inlets. When the "selector" box at the upper right is set to `0`, the target-based-weighting feature is inactive. Changing the selector to `1` sets `target-points` to the sequence `(0 1/2 1 3/2 … 11)` and `target-hit-weight` to `0.1`. With these settings, a choice from the `pool` will have its probabilistic weight multiplied by 0.1 whenever it would produce a note coinciding with one of the `target-points`. When the time signature is 2/4, the `target-points` are successive downbeats, and these settings result in rhythms that tend to avoid the downbeat. Finally, changing the selector to `2` sets `target-points` to the sequence `(0 1/24 1/12 3/24 … 11)` and `target-hit-weight` to `20`. With these settings, a choice from the `pool` will have its probabilistic weight multiplied by 20 whenever it would produce a note coinciding with one of the `target-points`. This results in rhythms that tend to adhere to a grid of triplet sixteenth notes.

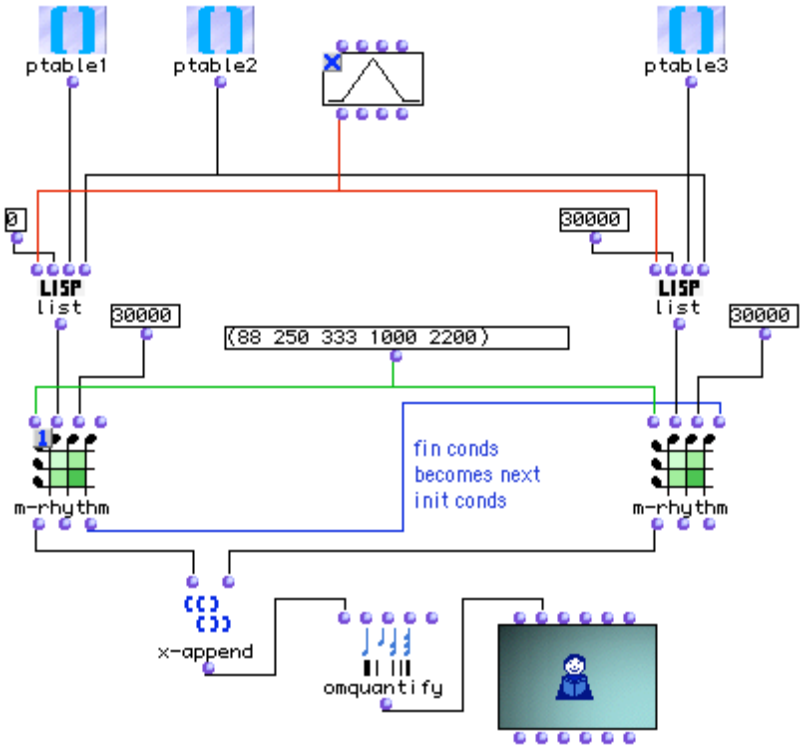
Target-based weighting can also be used to generate a counter-rhythm that seeks out or avoids coincidences with a rhythm you have already generated. In this application, the already-generated rhythm would serve as the `target-points` parameter, and `target-hit-weight` would be set to favor or disfavor coincidences with it.



■ *Figure 10*

m-rhythm cascade

This patch shows two `m-rhythm` objects arranged in a cascade (Figure 11), with the `fin-conds` output of the first object feeding the `init-conds` input of the second. While interpolation between two probability tables (or event pools) can be handled by a bpf driving a single `m-rhythm` object, cascade structures like the one illustrated here allow a larger number of tables (or pools) to be involved. In this case, the first `m-rhythm` object goes from `ptable1` to `ptable2`, while the second goes from `ptable2` to `ptable3`.

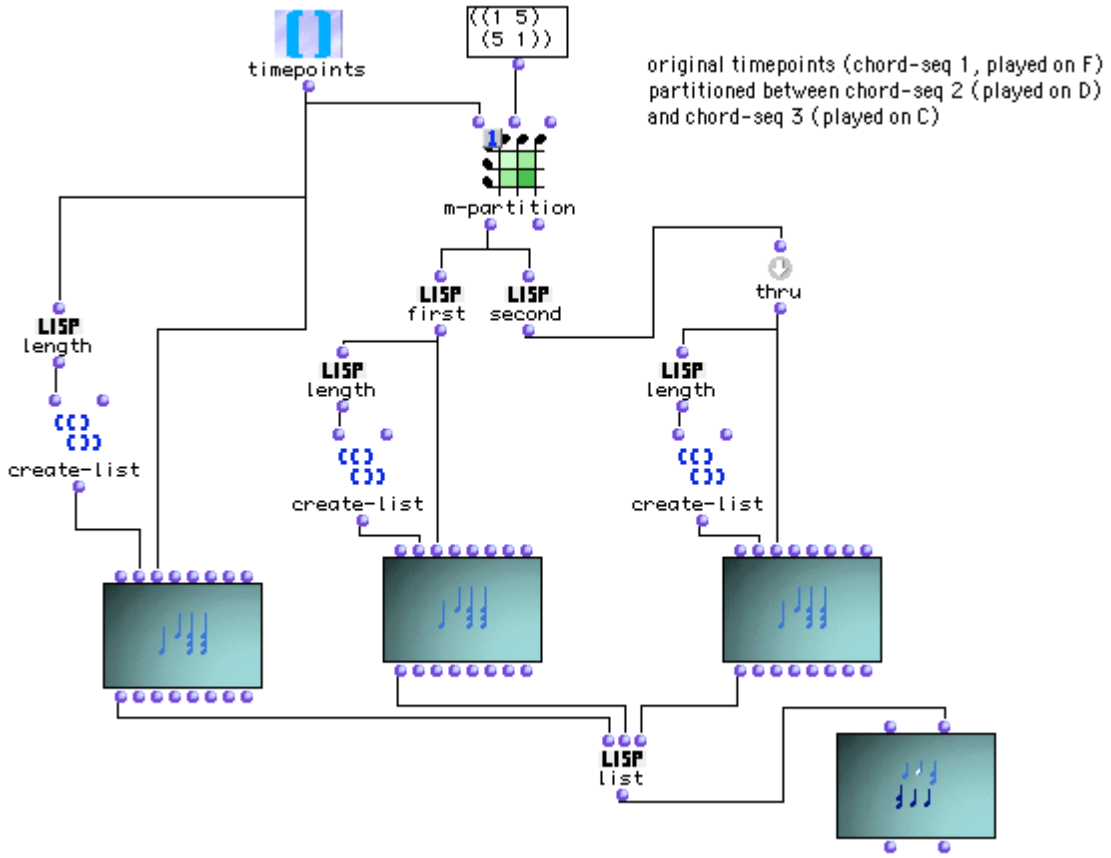


■ *Figure 11*

m-partition

This patch uses `m-partition` to distribute a given list of `timepoints` between two destination lists (Figure 12); the original list and its distributed parts are collected in `chord-seq` objects and combined in a `multi-seq` object. Select the `timepoints` object and press "v" to see its contents displayed in the Listener window. The number of partitions is determined by the shape and size of the `ptable` parameter; in this case, a 2×2 table controls partitioning between two destinations. These destinations are associated with contrasting pitches so the effect of partitioning can be heard easily in the resulting `multi-seq`.

The use of `chord-seq` (rather than `voice`) objects in this patch eliminates the need for quantization. The unquantized data could be used to drive a synthesis engine, or it could be saved as a MIDI file and opened in a sequencer for additional processing.



■ ***Figure 12***

empty-bars vari-pulse

The left-hand side of this patch (Figure 13) uses `m-empty-bars` to construct a list of empty bars spanning 25 whole-note units, where each empty bar is determined by a time signature represented as a (*numerator denominator*) pair. The statistical controls on this process involve bpf-driven interpolation between two probability tables: the left-hand table at the beginning of the process and the right-hand table at the end, with a gradual transition in between. A particular instance of the output from this object has been created (by shift-option clicking on the first outlet) and named "empty-bars." It will be put to use twice: as an input to `m-vari-pulse` on the right-hand side of this patch, and in the tutorial patch `mapped-rhythm`.

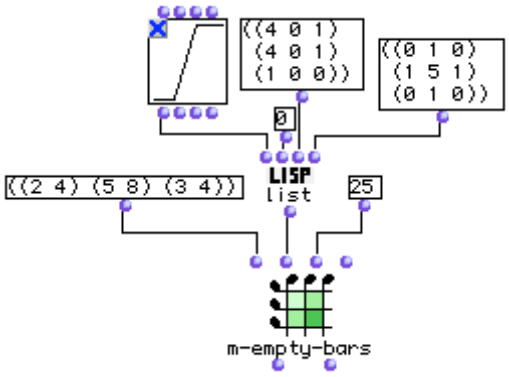


Figure 13

The right-hand side of the same patch (Figure 14) uses `m-vari-pulse` to fill "empty-bars" with a pulse that varies, according to statistical controls, while maintaining an idiomatic fit to the metrical constraints of "empty-bars." Double-click on the `pulse-map` subpatch to see how this parameter is formatted, and compare Figure 15, which shows the same data in conventional music notation. Because `m-vari-pulse` operates by tracing portions of the pulses in `pulse-map`, it can be converted to a rhythm tree (via `mktree`) to produce a metrically coherent result with no need for quantization. Note that the "empty-bars" list, used as a parameter of `m-vari-pulse`, is also supplied to `mktree`, so that the resulting variable pulse can be viewed in the appropriate metrical setting.

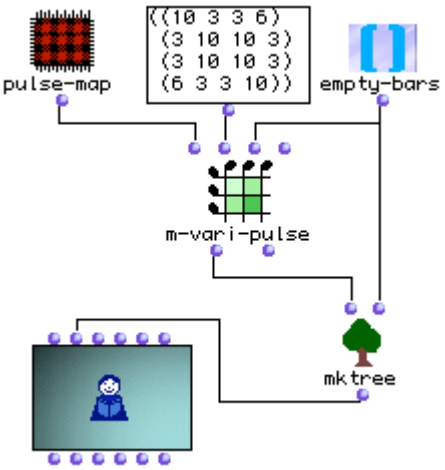


Figure 14

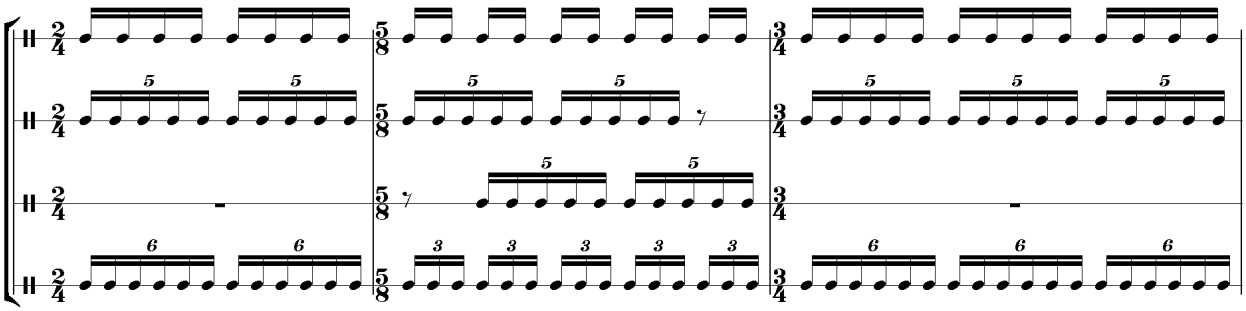
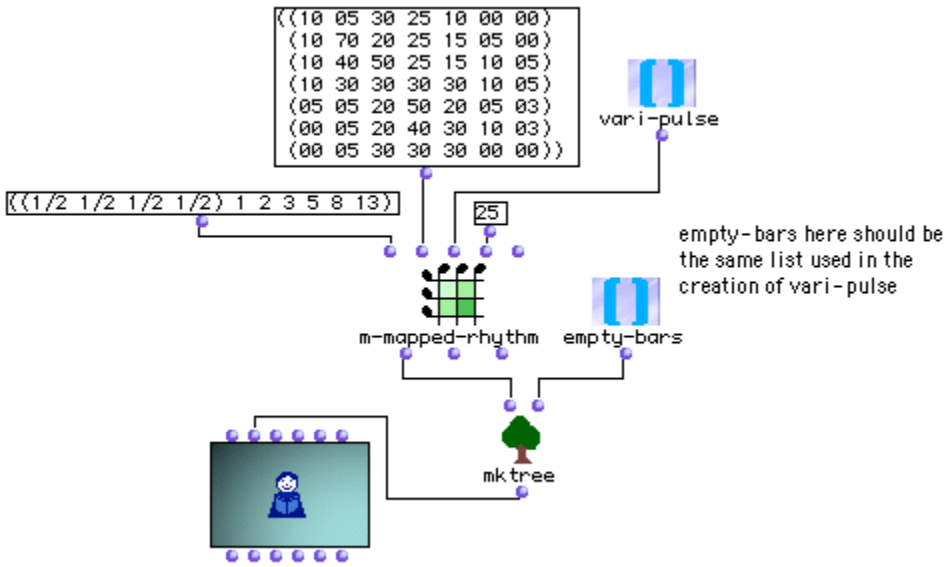


Figure 15

mapped-rhythm

In this patch demonstrating `m-mapped-rhythm` (Figure 16), the "empty-bars" and "vari-pulse" elements both derive from the tutorial patch `empty-bars vari-pulse`; "empty-bars" is not required by `m-mapped-rhythm` but is used further on by `mktree` in order to construct an appropriate metrical setting for the resulting mapped rhythm.

The `pool` from which `m-mapped-rhythm` makes random selections consists of a duration-pattern `(1/2 1/2 1/2 1/2)` and six simple durations. Each of these elements consists of one or more *generic durations* that will assume specific values once they are mapped onto a portion of "vari-pulse." The generic duration `5`, for instance, means "combine the next five elements of 'vari-pulse' and append the resulting duration to the output rhythm." Note that the duration-pattern in the `pool` includes fractional values, but that its contents still sum to an integer value as required. This allows access to sub-pulse details while keeping the output rhythm aligned to the variable pulse and its associated metrical context. The random selections are controlled, as usual, by the data supplied to the `ptable` inlet.



■ ***Figure 16***