UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Automatic Improvisation: A Study in Human/Machine Collaboration

Permalink
https://escholarship.org/uc/item/1kt0315h

Author
Wilson, Adam James

Publication Date
2009

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Automatic Improvisation: A Study in Human/Machine Collaboration

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Music

by

Adam James Wilson

Committee in charge:

Professor Philippe Manoury, Chair
Professor Diana Deutsch
Professor Shlomo Dubnov
Professor Miller Puckette
Professor Shahrokh Yadegari

2009

Copyright
Adam James Wilson, 2009
All rights reserved.

The dissertation of Adam James Wilson is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

Chair

University of California, San Diego

2009

TABLE OF CONTENTS

Signature Page . . . iii

Table of Contents . . . iv

List of Figures . . . v

Acknowledgements . . . vi

Vita and Publications . . . vii

Abstract of the Dissertation . . . viii

1 Motivations . . . 1

2 The System . . . 3
- 2.1 Real-Time Input . . . 3
- 2.2 Generative Processes . . . 4
  - 2.2.1 Factor Oracle . . . 4
  - 2.2.2 Rhythmic Flocking . . . 6
- 2.3 Pre-Composed Elements . . . 7
  - 2.3.1 L-Systems . . . 7
  - 2.3.2 Polyrhythms . . . 9
- 2.4 Guitar Processing . . . 9
  - 2.4.1 Guitar-Gong Hybrid . . . 9
  - 2.4.2 Guitar Effects . . . 10
- 2.5 Preparation . . . 10

A Max/MSP Externals . . . 12
- A.1 factorOracle.c . . . 12
- A.2 quantizeAlphabet.c . . . 32
- A.3 phraseDetect.c . . . 39

B Max/MSP Patches . . . 74

C CL-SYS . . . 121

D Duoquadragintapus DVD-R . . . 141
- D.1 Duoquadragintapus.mp4 . . . 141
- D.2 Duoquadragintapus.wav . . . 141
- D.3 Duoquadragintapus.wav.L, Duoquadragintapus.wav.R . . . 141

E Performance Program . . . 142

Bibliography . . . 143

LIST OF FIGURES

Figure B.1: Main control window. 75
Figure B.2: Main patch (p MAIN) 76
Figure B.3: Axon input filter (p axon_input) 77
Figure B.4: Absolute time to time interval conversion (p onsets_to_durations) 78
Figure B.5: MIDI filters (p filters) 79
Figure B.6: Input settings for MIDI filters (p MIDI_input_settings) 80
Figure B.7: Automated event scheduling (p score_playback) 81
Figure B.8: Factor oracle probabilities for bass accompaniment (p bass-probability) . . . 82
Figure B.9: Factor oracle probabilities for harmonic accompaniment (p lizards-probability) 83
Figure B.10: Dissonance values for chord voicings (p lizards-dissonance) 84
Figure B.11: X-alphabet factor oracle probabilities (p sound-file-player-oracle-probability) 85
Figure B.12: Weights for percussion flock timbral group selection (p weights) 86
Figure B.13: Timbral group selection for percussion flocks (p selectFlock) 87
Figure B.14: Phrase detection (p phrase-detection-for-robots) 88
Figure B.15: Phrase selection (p selectPhrase) 89
Figure B.16: Flocking information sent via OSC over UDP to Supercollider (p toUDP) . 90
Figure B.17: OSC message formatting for flock initialization (p assembleFlock) 91
Figure B.18: Ludbot MIDI trigger assignments (p ludbox_map) 92
Figure B.19: OSC for flock suspension, resumption, and destruction (p sus_res_kill_index) 93
Figure B.20: X-alphabet pitch-duration pairs mapped to sound file "buckets" (p sfplayer) 94
Figure B.21: Sound file players (p sfplay_objects) 95
Figure B.22: Sound file bucket selection (p choose_soundfile) 96
Figure B.23: Sound file selection (poly~ poly_buckets) 97
Figure B.24: Sound file transposition (p get_pitch_scalar) 98
Figure B.25: Sound file → sound file list assignments (p fill_sflists) 99
Figure B.26: 5.1 surround mix for pre-rendered sound files (p sf_mixer) 100
Figure B.27: Player for pre-composed ludbot sequences (p MIDI_bot_player) 101
Figure B.28: Guitar → ludbot mappings for ludbot "lockstep" mode (p ludbot-lockstep) . 102
Figure B.29: Separation of guitar signal phase and amplitude (p hilbert) 103
Figure B.30: Oscillator bank for gong timbre simulation (p oscilbank) 104
Figure B.31: Parametric control of individual oscillators (poly~ poly_oscil) 105
Figure B.32: Guitar velocity mapped to gong inharmonicity (p velocityToInharmonicity) 106
Figure B.33: Factor oracle generation of harmonic accompaniment (p harmony-oracle) . . 107
Figure B.34: Voicing selection for harmonic accompaniment (p voicing) 108
Figure B.35: MIDI information formatted for VST synthesizer plug-ins (p makeVSTnote) 109
Figure B.36: Pitch values formatted for AAS synthesizers (p convertMIDI-1) 110
Figure B.37: Factor oracle generation of bass accompaniment (p bass-oracle) 111
Figure B.38: MIDI information formatted for VST sampler plug-in (p makeBassNote) . . 112
Figure B.39: Pitch values formatted for Trilogy sampler (p float-to-MIDI-key-and-bend) . 113
Figure B.40: Pitch-shift, chorus, and reverb effects (p guitarEffects) 114
Figure B.41: Pitch-shift settings (p delay) 115
Figure B.42: Pitch shifter (p ps_by_delay), adapted from Puckette's G09.pitchshift.pd . . 116
Figure B.43: Chorus settings (p chorus) 117
Figure B.44: Grouped pitch shifters (poly~ poly_ps-by-delay) 118
Figure B.45: MIDI footswitch controller mappings (p footswitch) 119
Figure B.46: Mixer (p mixer) 120

ACKNOWLEDGEMENTS

Thanks to William Brent for constructing and troubleshooting "ludbots," his implementation of MIDI-controlled percussion-playing robots, and for the use of his rhythmic flocking software. Thanks to Dustin Rafael for countless hours of equipment setup and hardware troubleshooting. Thanks to Rand Steiger and Steve Schick for seeing the value in providing funding for materials to build additional ludbots, which have, since the completion of this project, become available to other students, faculty, and researchers in the UC San Diego Music Department. Thanks to the staff at the UCSD Center for Research in Computing and the Arts for providing, in conjunction with the California Institute for Telecommunications Technology, the equipment, research space, and performance space without which an initial realization of this project would not have been possible. Thanks to Carol Krumhansl for advice concerning the perception of musical phrases. Finally, thanks to my committee members for support, guidance, understanding, and criticism.

Parts of chapter two have been previously published as:

Wilson, Adam James. "Flocking in the Time-Dissonance Plane," *Proceedings of the International Computer Music Conference*, Montreal, Canada, 2009, pp. 387-390.

Additional parts of chapter two have been previously published as:

Wilson, Adam James. "A Symbolic Sonification of L-Systems," *Proceedings of the International Computer Music Conference*, Montreal, Canada, 2009, pp. 203-206.

VITA

1997	B.A., Oberlin College
1999	M.Mus., Composition, University of Illinois at Urbana-Champaign
2005	Permanent Secondary Teaching Certificate, University of the State of New York
2009	Ph.D., Music Composition, University of California, San Diego

PUBLICATIONS

Wilson, Adam James. "Flocking in the Time-Dissonance Plane," *Proceedings of the International Computer Music Conference*, Montreal, Canada, 2009, pp. 387-390.

Wilson, Adam James. "A Symbolic Sonification of L-Systems," *Proceedings of the International Computer Music Conference*, Montreal, Canada, 2009, pp. 203-206.

ABSTRACT OF THE DISSERTATION

Automatic Improvisation: A Study in Human/Machine Collaboration

by

Adam James Wilson

Doctor of Philosophy in Music

University of California San Diego, 2009

Professor Philippe Manoury, Chair

Described herein is a practical experiment in the spontaneous composition of music through real-time interaction between a human composer-performer and a semi-autonomous network of computer processes. A software system called *The Duoquadragintapus* was devised for this purpose, and used in performance to create and record musical variants of the same name, bounded by the resources and behaviors of generative processes allocated to the system by its author, who also occupies the role of composer-performer when the system is active. *The Duoquadragintapus* includes pre-composed music that can be incorporated at the whim of its human collaborator, as well as a number of independent processes designed to transform material derived from live instrumental performance and navigate relationships between live input and pre-rendered output. The human performer has two responsibilities: first, as an improvising instrumentalist, providing raw material for real-time software processes, the functioning of which remain independent of his control during performance; secondly, as a software engineer and composer, determining the boundaries of the system prior to performance and selecting when and how many of the computer system's various process classes are allowed manifest instances during performance. In this version of the *The Duoquadragintapus*, the performer uses a fretless electric guitar as an input instrument, and music is output through a battery of 32 percussion-playing robots and a 5.1 surround sound speaker array (though the system can easily be adapted to accommodate other types of input and output). High-level system control originates in MIDI information from the guitar and from a footswitch controller. The major components of the system include signal processing of audio from the guitar, factor oracle generation of accompaniment harmonies, factor oracle selection of pre-composed audio snippets generated via sonification of L-systems, phrase detection and rhythmic interpolation of phrases, a "lockstep" mode in which percussion robots mirror the rhythms played by the guitarist, and pre-scored control of performance variables such as changes in probabilities governing factor oracle traversal, choices for accompaniment voicing, and changes in ownership of parameter control from human to computer.

1 Motivations

Most of the research I've done in the context of music has been motivated by creative needs. In all cases, an imagined composition, compositional idea, or improvisational structure precedes any software or hardware tools I eventually design to realize it. Although it is often more practical to work creatively within the confines of existing and well-tested tools, I prefer to advocate for the primacy of the imagination – we should build tools to serve the needs of our creative impulses, rather than cripple our ideas in order to satisfy the constraints of existing and familiar creative frameworks.

For several years I've imagined building a computer system that generates musical structure in real-time in response to and abstracted from the musical structures, timbres, and natural gestures produced by human performers. I have built and tested two such systems, including the system describe herein. Spontaneous creation of musical structure is useful for building coherence into freely improvised music (whether solo performer or group), facilitating modular compositions, and for rapid prototyping and testing of compositional ideas. One does not have this freedom with a score that is completely frozen and meant to be performed exclusively by human musicians; changes must be rewritten and relearned, which is often impossible, or at least extremely impractical. Similarly, improvising musicians working solely with other humans, who may respond quickly and easily to musical features we labor to teach computers to recognize, are often unable to spontaneously develop complex or large-scale formal structures; such structures have to be agreed upon preconditionally, and modifying them also requires periods of relearning and rehearsal. In both contexts, the computer can be effectively used to encapsulate and execute formal ideas; generative algorithms may be coupled with MIR systems that identify a wide range of types of musical information, and the types of inputs used as triggers or material for development through the use of generative algorithms can be tailored to suit the characteristics of a composer's score, a solo improviser's style, or the aggregate tendencies of an improvising collective.

I am ultimately interested in designing modular computer systems that have the ability to mediate between sets of unknown musical catalysts, originating in a score or improviser, and sets of abstracted compositional ideas. The compositional abstractions with which music information retrieval strategies are paired present a more subjective/personal problem, and may

need to be developed or modified with some reference to the intended input. Whatever form they take, it is important to realize that they should not be passive or "reactionary," statically responding to isolated input events of a certain class or depending upon the existence of a large-scale structure in the input to appear coherent. This generative algorithmic layer should serve to build bridges between disparate input materials, highlight subtle repetitions or similarities across input materials, or form an independent structure using material from the input as a scaffolding. In its most extreme incarnation, this set of tools should be designed to create structure out of musical fodder that may not exhibit, in and of itself, any perceivable formal structure.

A further advantage of developing software tools for spontaneous generation of musical structure is the richness of output possibilities available through the computer. Although group improvisers benefit from the ability of the computer to aid in consolidating structure, and composers from the ability of the computer to introduce structural flexibility, an extremely talented solo improviser may be capable of a performance that combines both spontaneity and structural coherence. In this case, there is still an advantage to be had from working with a real-time computer system, particularly if the improviser in question is able to program some of the abstractions in the proposed system and learn to how to trigger and shape them, thereby gaining access – through his instrument – to musical materials and generative possibilities (small- or large-scale) that are not available to him on his instrument alone. In this environment, the traditional instrument can become a "hyperinstrument," not only in the timbral domain, but in the compositional domain. Output possibilities – including signal processing of the input, triggering of samplers and synthesizers, and even real-time production of music notation and video or audio cue signals to be fed back to the performer – can be one-to-one results of an action on a musical instrument, or more interestingly, the results of processes set in motion by an action on the instrument.

Perhaps the most useful feature arising from human interaction with a digital collaborator is the potential for unpredictable feedback, occurring directly through the aforementioned possibility of literal cues to performers, indirectly through the influence of computer-generated musical output on an improviser, or conceptually, through the potentially surprising results of a composer's attempt to encapsulate a familiar musical abstraction as an algorithmic entity. Music is never fully imagined and then flawlessly created to the specifications of that imagination, though it may be possible to gauge its banality or attractiveness to the extent that it is; music can only be intentional if it more or less conforms to musical surfaces with which its creator is already familiar. Building a system out of abstract blocks of process, rather than concrete musical structures, allows for complex interactions that have the potential to yield serendipitous accidents. Such accidents are keys to the production of novel sound and structure.

2 The System

2.1 Real-Time Input

The Duoquadragintapus takes real-time input from three sources: (1) a MIDI representation of each note played on the input instrument, (2) the audio signal from the input instrument, and (3) a footswitch MIDI controller. The guitar being used for input in this version of the piece has both magnetic pickups and separate piezo-electric pickups for each string. The audio signal is routed directly to Max/MSP – the environment in which the core of the real-time system is implemented – and the pitch of each signal from the piezos is detected and converted to MIDI using a Terratec Axon AX-50 guitar-to-MIDI converter. The MIDI signals from the strings are sent to the Max patch via USB, and only the strongest of the six sources is allowed through at any given time – constraining the input to monophony. Since the guitar being used is fretless (due to the composer's predilection for microtones), and the majority of the playing is necessarily single-line, the gating provides a kind of noise reduction (see figure B.3). The absolute times of note-ons originating at the piezo elements are converted to durations, and the dominant piezo signal is filtered to compensate for vibrato and microtonal portamento, unintentional low-velocity vibrations, and short-duration miss-strikes; durations are also clipped to a maximum length, which prevents the storage of a duration that lasts inordinately long as a result of the player choosing to rest for a substantial period of time (see figures B.4, B.5, and B.6).

The audio signal from the guitar is routed first through a noise reduction unit, then split and sent to the Max patch and to a class-A tube amplifier. The tube amplifier is used as a preamp to color the sound with high-gain distortion. The output of the amplifier is routed through a power brake to a second noise reduction unit, followed by a set of speaker simulators, then on to the Max patch where it is passed through a reverberator and out to a central mixing board for location in 5.1 space. Within the Max patch, the clean signal is subjected to a number of audio processes, described in section 2.4. The original signal and the outputs of processes acting in parallel on the signal are sent separately to the the mixer for spatialization.

The footswitch is used to turn on and off groups of guitar signal processes, to launch pre-composed snippets of music, and to control the density of otherwise autonomous algorithmic processes, affecting the surface shape and structure of the composition (see figure B.45). This

is the only top-level compositional control available to the composer-performer once the piece is underway; other variables that affect structure are fixed in the patch prior to performance.

2.2 Generative Processes

A number of semi-autonomous parallel algorithmic processes are set in motion once the guitarist begins to play. While the ways in which these processes work to respond to input or trigger events are not definable in performance, the composer-performer is explicitly given the ability, in some cases, to control the density of a process class by silencing all instances of that class. In other cases, he is allowed to set a limit on the number of duplicate processes of a particular class operating in parallel on different sets of data. The surface of events can also be indirectly influenced by the composer-performer, relying on an intimate knowledge of the ways in which the algorithms respond to input to tailor his playing. While this kind of influence can produce generally predictable results when the player concentrates his effect on isolated process instances, the overwhelming complexity of the output is such that the performer often has to switch from stimulus to response mode, forced to react to music regurgitated by the computer as he would to an unexpected musical transformation introduced by a human collaborator.

2.2.1 Factor Oracle

Of all the algorithmic processes employed in *The Duoquadragintapus*, factor oracle analysis is the one that allows the computer to behave most like a seasoned improvising partner. The factor oracle is an efficient automaton capable of producing at least all of the substrings of a particular input string (it may also find substrings that don't occur in the input). Allauzen, Crochemore, and Raffinot describe an "online" version of the factor oracle construction algorithm, in which the automaton can be built incrementally as elements are added to the input string (Allauzen et al. 5-8). Elements of the input string become nodes in the automaton. Each node is connected to the following node by a forward link. Some nodes are connected to non-adjacent nodes by forward links that appear when the origin node for the link ends a substring with a suffix identical to the suffix of a substring ended by the node to which the link points. Backwards links, called suffix links, trace the path of a recursive supply function used to determine whether more forward links will be added when a new element appears in the input string. The greater the ratio of suffix links to forward links chosen when traversing a factor oracle, the less the output string will correlate with the input string (Assayag and Dubnov 4). This ratio can be construed as the probability of congruence between input and output.

A generalized factor oracle program for *The Duoquadragintapus* ("factorOracle") was implemented in C and compiled as an external object for Max/MSP 5.0 (see Appendix A, section 1). The first inlet of the factorOracle object accepts any of the data types available in the

default Max/MSP environment, including integers, floats, symbols, and lists. The input string is formed by concatenating data arriving at the first inlet until a memory limit established upon initialization of the object is reached. The second inlet sets the length of the output string to be generated each time a "bang" is received in the first inlet, and the third inlet sets the probability of congruence between input and output strings. In *The Duoquadragintapus*, factorOracle objects are used for two different tasks: real-time construction of a harmonic progression and bassline, and navigation of a sequence of keys made from a normalized cross-alphabet of pitch and duration information. Each key in the second case triggers the playback of a unique pre-recorded snippet of audio, barring interference from the composer-performer or from built-in limits on the number of audio files that can be played back simultaneously.

Velocity, pitch, and duration information are dealt with separately in the construction of the harmonic material and bassline. Incoming notes are separated into three input strings corresponding to the three parameters. These strings are then passed to three factorOracle objects used for generating harmonies and three factorOracle objects used for generating the bassline (see figures B.33 and B.37). Harmony oracles are constrained to output strings of 5, 6, or 7 elements. All possible voicings, or orderings of pitch-classes, for 5, 6, and 7 note chords – not including augmentation of each unique arrangement of pitch-classes by extra octaves – are calculated and stored prior to performance. Voicings are ordered by dissonance, from the "closest" harmony, having the lowest sum of intervallic distances between every pair of notes, to the most "open" harmony, having the largest sum of intervallic distances between every pair of notes (see figure B.34). For each chord produced by the harmony oracles, we select an index from the list of voicings corresponding to the number of notes in the chord; the lower the index, the greater the dissonance of the output harmony. Notes from bass oracles are transposed by powers of two to bring the pitch-class of each note as close as possible to an absolute pitch in the lower end of the audible range. Changes in probabilities for harmony oracles and bass oracles and changes in dissonance values for the chords output by the harmony oracles are preset in the Max patch (see figures B.7, B.8, B.9, and B.10).

A single factorOracle object, coupled with another Max/MSP external object called "quantizeAlphabet" (see appendix A, section 2), determines (for part of the piece) which pre-recorded audio snippets will be sent to the speakers. The quantizeAlphabet object maps a cross-alphabet of guitar pitch-duration pairs to fifteen sets of unique pre-rendered sound files, each distinguished by a particular combination of timbre, register, and duration. Initially, the player can navigate the matrix of cross-alphabet mappings, intentionally eliciting certain types of sounds by playing high and short notes, long and low notes, etc. After a period of time, the score control patch (figure B.7) gates off the information coming directly from the player to the map and promotes a factorOracle object in his place. Since the entirety of the performer's cross-alphabet string of rhythms and registral choices is stored in the factorOracle object, we can use the factorOracle to trigger sequences of pre-recorded sounds that more or less mirror

or distort the past choices of the performer by changing the value of the oracle's probability argument. Changes in probability values for this factorOracle object are preset in the Max patch (figure B.11), along with the time at which control over triggering sound files is switched from the human performer to the oracle (figure B.7).

2.2.2 Rhythmic Flocking

A second category of generative process used in *The Duoquadragintapus* involves recognition and manipulation of phrases produced by the instrumentalist. According to Krumhansl and Jusczyk, the most significant perceptual markers of phrase ends occur when one encounters in a melody a sharp drop in average pitch duration and/or a sharp drop in average pitch height (Krumhansl and Jusczyk 72). Another Max/MSP external, "phraseDetect," was coded to use these metrics to listen for phrases in a stream of notes (see Appendix A, section 3). Each instantiation of the phraseDetect object must have an analysis window size, equal to the number of notes to be parsed for phrases at one time. Other initialization arguments include the minimum number of contiguous notes allowed to be identified as a phrase, the maximum number of contiguous notes allowed to be identified as a phrase, the maximum duration allowed in a phrase, and a threshold value for determining phrase ends. One threshold value is used for both duration and register, and it is defined as a fraction of the difference between the median duration (or register) and the maximum duration (or register) in a given analysis window. If there are leftover notes at the end of an analysis, they are moved to the beginning of the next analysis window.

The phraseDetect object outputs an index number and lists of pitches, velocities, and durations for each of two categories of phrases: phrases that end with with a sharp drop in duration but no significant drop in pitch height, and phrases that end more convincingly with a sharp drop in duration and a simultaneous drop in pitch height. These phrases are dumped into two separate "buckets." The bucket of stronger phrases is the primary source of fodder for manipulation by the computer. The bucket of weaker phrases is only tapped if the other bucket is empty or all of the phrases in the other bucket have been used.

Pairs of phrases are selected and sent to a version of the flocking algorithm, implemented by William Brent in Supercollider.

> In general, the flocking algorithm is used to describe the optimal movements of several independent agents as they attempt to maintain predetermined positions relative to one another and with respect to the position of a freely moving leader. Normally, the movements of the agents are confined to a plane. Each of the followers (1) observes the positions of the agents, including itself, (2) calculates the centroid of the agents positions, (3) rotates and translates the predetermined pattern of desired positions around and along the line formed by the leader and the centroid, (4) observes the position of each follower relative to the available positions in the rotated and translated pattern, (5) selects the position in the projected pattern that is closer to its current position than to the positions of any of the other followers, and (6)

> moves incrementally to the selected position. Followers repeat these calculations and movements each time the leader changes position (Wilson 387).

In Brent's implementation, flocking agents are notes in a phrase and their positions are described by the rhythmic intervals between each pair of neighboring notes in the phrase. For each instance of a flock, the starting positions of notes/agents are taken from the durations list of a phrase discovered by the phraseDetect object; destination positions are taken from the durations list of a second phrase. With each iteration of the phrase, or flock, over a specified period of time, beginning with notes in their starting positions in time, the rhythmic intervals between notes undergo a degree of interpolation as the agents attempt to find their new positions. By the last iteration, the rhythmic disposition of the notes is identical to the disposition of the notes in the destination phrase.

The output of this process is realized by the percussion-playing ludbots. An attempt is made to differentiate simultaneous rhythmic flocks by routing each flock to a different timbral group of percussion instruments (see figures B.13, B.16, B.17, and B.18). Although the maximum number of simultaneous flocks is limited to two, there are so many other processes happening concurrently that the composer-performer may find it desirable to remove flocking from the output entirely. Footswitch control enables the composer to kill all currently playing flocks, to turn off the process completely while not disturbing currently playing flocks (by setting the density to zero), and to set flock density to the maximum. Since a flock is launched every time the guitarist plays a note – unless there are no phrases available or the maximum number of flocks is currently being executed – the performer may play a single flock by setting the flock density to the maximum, playing one note on the guitar, and then setting the density back to zero.

2.3 Pre-Composed Elements

2.3.1 L-Systems

Over 600 sound files were pre-rendered for use in *The Duoquadragintapus*. When sound file playback is enabled, a new sound file is triggered each time the guitarist plays a note. Which file is played depends on information coming directly from the guitarist or from information selected from a string of stored data in a factor oracle, as detailed in section 2.2.1. Simultaneous playback of files is allowed, up to a maximum of 10. The footswitch controller allows the composer-performer to adjust the density of the events in performance by setting the density threshold to either 0 or 10. Setting the density to 0 does not interrupt sound events already being played, so densities between 0 and 10 can be achieved by switching the density to the maximum, playing a few notes, and then switching the density back to zero.

Many of the sound files were rendered using a sonification method that creates an aural

representation of the branching structures exhibited by plants. Such representations are known as Lindenmayer systems (L-systems) after Aristid Lindenmayer, who defined them. L-systems are essentially recursive rewriting systems: a rewriting rule is applied to an inital "generator" object, then the same rule is applied to the rewritten object, etc. In L-systems, the pattern of branches attached directly to the stalk or trunk constitutes the generator object. The branching structure of the generator is replicated on a smaller scale around each of its branches, then around the newly drawn branches, etc. The structural quality of an analogous auditory object made in this fashion is less dependent on the process itself than on the specific visual-to-auditory mapping employed.

> In this case, time is oriented to the direction of growth: the trunk represents a pitch to be played at time = 0. The node, or root, at which each branch attaches to its parent represents a point in time corresponding to the distance between the base of the trunk and the intersection of the trunk with a perpendicular line drawn between the trunk and the node. The location of this point is actually determined by multiplying the cosine of the angle of the branch's parent relative to the trunk by the distance between the root of the parent branch and the root of the branch in question, and then adding the result to the point in time given by the root of the parent branch. While branch nodes become time-points, their associated angles become pitches occurring at these time-points. All angles are expressed relative to the trunk (Wilson 204-205).

The code for the construction algorithm above, "CL-SYS," is given in Appendix C. CL-SYS – a conflation of "Common Lisp" and "L-SYStems" – has one dependency, Heinrich Taube's Common Music 2.6. Common Music allows CL-SYS to write MIDI files. Output files are rendered in Applied Acoustics Systems physical modelers using customized gong and string instrument models. Buckets of sound file categories are created, and normalized cross-alphabet pairs of durations and pitches available on the guitar are associated with each of the buckets (see section 2.2.1). The files in each bucket are unique from each other and from the files in other buckets (they are all constructed using different generators and rewriting variables), but within each bucket files share a timbre, length, and registral bandwidth (there are 3 timbres, 3 registers, and 5 categories of duration – from a maximum of 1 second to a maximum 2 minutes in length).

The center, or "trunk" frequency of each L-system is generally perceived as its tonal center. In order to introduce coherence between the guitar and the sounds (beyond that provided by the simultaneous attack of the guitar and an associated sound file), the real-time component of the program attempts to find an L-system in the chosen bucket that can be transposed by resampling without perceivable timbral distortion into a ratio of a perfect unison, octave, fourth, or fifth with the fundamental frequency of the guitar note triggering the sound file (see figure B.24). The center frequency for each L-system is stored prior to performance, so no real-time analysis is required. Other pre-composed sounds were constructed using granular synthesis, out of wavlets culled from hundreds of recorded gong samples. The results of the granulations were analyzed by ear and using Michael Klingbeil's *Sinusoidal Partial Editing and Resynthesis* (SPEAR) program to determine the strongest perceived tone to be used as the basis for transposition.

In this iteration of *The Duoquadragintapus*, the guitarist plays with a battery of live percussion instruments, including metal gongs and pipes. Gong models and gong samples were chosen for sounds that the performer could elicit in order to build a connection between the two sound worlds – string and metal. String sounds for pre-rendered files were also chosen based on their potential to bridge the two timbral worlds; these are based on a hammered-string model, which provides a type of attack somewhere between the pluck of the guitar and the sharp strike of hard mallets on metal surfaces.

2.3.2 Polyrhythms

One potential pitfall in the realization of *The Duoquadragintapus* is the tendency of the work to fall into rhythmic amorphousness; the guitarist can play rhythmically, and this will be reflected to some extent in the rhythmic phrases produced by the flocking algorithm (see section 2.2.2); the source and destination phrases are repeated, several times if desired (this behavior is settable in Brent's flocking program), but long periods of interpolation and the absence of a connection to the guitar by pitch quickly obscure these repetitions. Factor oracles generating the harmonic accompaniment and bassline have greater potential to duplicate large chunks of rhythmic material instantiated by the composer-performer, but these rhythms can often be overwhelmed by a high density of active ludbots or a high density of ametrical L-systems and gong granulations.

The problem was mitigated by pre-composing a number of rhythmic sequences, each a few minutes long – steady, perceivable polyrhythmic pulses with no more than two contrasting rhythms at any one time, to be articulated by the ludbots. There are 16 of these rhythmic segments (see figure B.27), which could, if presented without any of the other elements of the piece, produce a fairly substantial length of music in the style of Conlon Nancarrow. The composer has the ability to launch one of the sequences from the footswitch controller when he feels the need to introduce some rhythmic regularity. The impact of the sequences can be enhanced by setting the sound file density to zero and gating off flocks at the same moment in which a sequence is triggered.

2.4 Guitar Processing

2.4.1 Guitar-Gong Hybrid

A novel species of guitar processing, inspired by Miller Puckette's "Patch for Guitar," is employed in another attempt to blend the sound worlds of the electric stringed instrument with live percussion, percussion models, and percussion samples (Puckette 3). A single tone from the gong model used to render L-systems was analyzed in SPEAR, and the average frequencies of partials and their relative amplitudes saved in an oscillator bank (see figures B.30 and B.31). Each

time a note-on from the guitar is detected, the sinusoids in the oscillator bank are transposed so that the perceived fundamental frequency of the simplified gong model matches the fundamental frequency of the guitar note. Simultaneously, the amplitude information from the guitar is separated from its phase information and multiplied by the output of the oscillator bank, to produce a gong timbre with the amplitude envelope of a plucked string (see figure B.29). Changes in guitar attack velocity also affect the timbre of the gong; the louder the guitar attack, the greater the inharmonicity of the gong (see figure B.32).

2.4.2 Guitar Effects

The guitarist is capable of producing a myriad of timbres in the *The Duoquadragintapus*; there is a high-gain signal from the class-A tube amplifier, a signal resulting in a sort of guitar-gong hybrid, and a clean signal. Aside from reverberation, which is applied in various amounts to all three signals, the clean signal can be pitch-shifted, and the guitar-gong hybrid signal is chorused by a bank of pitch shifters that can duplicate the signal in a harmonic series of ratios or in a set of very close ratios, producing a cluster. Chorused sounds can also be pitch-shifted. Many combinations of timbre are possible here, but they are limited to three choices (the high-gain sound is omnipresent, unless the volume of the magnetic pickups on the guitar is cut): (1) no pitch shift on the clean signal, and a substantial upward pitch-shift in the cluster-chorused hybrid guitar-gong; (2) a substantial upward pitch shift on the clean guitar signal, and a substantial upward pitch shift in the harmonic-series-chorused hybrid guitar-gong; (3) a substantial downward pitch shift on the clean guitar signal, and a substantial downward pitch shift in the cluster-chorused guitar-gong. (See figures B.40, B.41, B.42, B.43, and B.44.) All of these timbres are routed separately to an outboard mixer and located in 5.1 space.

The guitarist also has the opportunity to perform in “lockstep” mode, selectable at the footswitch, in which each note on the guitar within a specified range is mapped to a different percussion-playing robot. When a note in the specified range is struck, the ludbot associated with that particular tone responds immediately. The guitarist can choose, for example, to lower the volume on the instrument’s magnetic pickups, cutting off all audio output from the guitar except for the piezos, which are used exclusively for pitch tracking, and play a percussion solo using the guitar as a controller.

2.5 Preparation

With such a complex system, involving both pre-composed elements and the ability to generate music and control dozens of parameters in real-time, sculpting a coherent piece takes a lot of practice. The first concert of the *The Duoquadragintapus* took place on May 19, 2009, in Atkinson Hall at the University of California, San Diego. Due to the extraordinary amount

of hardware and software to set up and troubleshoot, we were only able to test components of the system prior to that day. Before the 8:00 PM concert on May 19, I was able to perform the entire 15-minute piece, using the complete system, a total of 8 times. By virtue of having conceived the system, composed the pre-rendered musical sequences, and having programmed the majority of the system processes, I had already developed some ideas about how I would use it to build a piece of music in real-time. But virtuosic physical control over such an instrument – and this is really a large instrument, of which the guitar is only a small part – requires much more than theoretical understanding; there is the superficial kinetic aspect of manipulating pedals and guitar strings, but, more importantly, there is the exploration of connections between actions and system responses, which are quite clear when dealing with each component process of the system separately, but quite abstruse when dealing with them altogether. I attempted in practice to divide my time, as judiciously as possible, between (1) stimulating or limiting the agents of the system, and (2) producing music on my instrument to complement the output of the system. I also began to find the limits of perception for simultaneous instantiations of various processes, given the constraints of the timbral and physical space occupied by the piece; this sort of experimentation provided the basis for setting the maximum flock and sound file densities (2 and 10, respectively).

I don't look at the first performance of the piece as the representation of a finished work. *The Duoquadragintapus* is not really a single piece, but a world of related possible compositions, bounded by the rules of the system, the capabilities of the input and output devices attached to it, and the people with which it interacts. A second performance, held on July 11, 2009, also in Atkinson Hall, benefitted from new percussion instruments (chosen for their potential to blend more effectively with the electronic timbres) and a better sense on the part of the composer – owing to close scrutiny of the recording of the May 19 concert – of the formal possibilities latent in the system. The difference between the recording of the premiere and the July 11 concert already shows an appreciable evolution in the coherence of the sonic result. The goal, with subsequent performances, recordings, and modifications of the system, is to refine the sound world of *The Duoquadragintapus* without inhibiting its flexibility to produce a variety of distinct pieces.

Parts of chapter two have been previously published as:

Wilson, Adam James. "Flocking in the Time-Dissonance Plane," *Proceedings of the International Computer Music Conference*, Montreal, Canada, 2009, pp. 387-390.

Additional parts of chapter two have been previously published as:

Wilson, Adam James. "A Symbolic Sonification of L-Systems," *Proceedings of the International Computer Music Conference*, Montreal, Canada, 2009, pp. 203-206.

A Max/MSP Externals

A.1 factorOracle.c

```
////////////////////////////////////////////////////////////////////////////////
//
// factorOracle, a MAX external
// Adam James Wilson
// ajwilson@ucsd.edu.com
//
////////////////////////////////////////////////////////////////////////////////


////////////////////////////////// License.

/*
This software is copyrighted by Adam James Wilson and others. The following
terms (the "Standard Improved BSD License") apply. Redistribution and use in
source and binary forms, with or without modification, are permitted provided
that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright  notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived
   from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
*/
```

```
//////////////////////////////// Potential improvements.

/*
1. The original external was compiled using XCode 2.5 for Max/MSP v. 5.0 on a
   Macintosh PPC running Mac OS 10.4. After recompiling for a Macintosh Intel
   machine, using XCode 3.1.2, float boxes connected to inlets on the
   factoOracle object would not send proper values unless the functions they
   called were set to accept doubles instead of floats. Investigate this
   issue.
2. Make an fO object specifically for MIDI, and add support for cross-alphabet
   combinations of intervals or absolute pitches, durations, and amplitudes.
3. Make the buffer continuous, or "forgetful," so that at any given time there
   are two oracles running -- one that starts halfway into the first and
   finishes at 1.5 times the length of the first, at which point the first
   would be freed and a third begun. This removes the need for an upper limit
   on the number of input atoms.
4. Allow the possibility of loading a stored list of atoms that you can continue
   with live input, and a way to save live input for import into future
   factorOracle instances (a sort of "training memory" for a particular
   improvisor's style).
5. Find a way to deal effectively with harmony.
6. Come up with a better probability model for traversing the factor oracle to
   generate new strings.
7. Use lengthCommonSuffix to find longer/short common suffixes in the suffix
   chain originating at a particular state (longer common suffixes produce
   smoother intra-string transitions, while shorter ones produce choppier
   transitions).
*/


//////////////////////////////// Dependencies.

#include "ext.h"           // Always required for MAX.
#include "ext_obex.h"      // Required for new style MAX objects.
#include "ext_systhread.h" // MAX thread-locking functions.


//////////////////////////////// Type definitions.

typedef struct _state
{

    long suffixLink;                 // The index of an earlier state in the
                                     // array of states in the containing
                                     // t_factorOracle object.
    long lengthCommonSuffix;         // Not used; to be implemented at a later
                                     // date.
    long numberOfTransitionElements; // Size of both transitionElements and
                                     // transitionEndStates.
```

```
    long *transitionElements;        // A list of indices referring to the
                                     // alphabet array in the containing
                                     // t_factorOracle object.
    long *transitionEndStates;       // A list of indices referring to the
                                     // array of states in the containing
                                     // t_factorOracle object.

} t_state;

////////////////////////////////

typedef struct _factorOracle
{

    t_object ob;          // The object itself (this must always come first
                          // inside the main object struct).
    void *m_outlet1;      // Output list of values generated by the oracle when
                          // a bang is received.
    long input_size;      // The maximum length of both alphabet and states.
    t_atom *alphabet;     // Unique elements from input. Alphabet keys are the
                          // indexes of elements in the alphabet, stored in
                          // t_state.transitionElements.
    long input_index;     // The next index x to be used to store an input atom
                          // in states[x].transitionElements.
    long alphabet_index;  // The next index to be used to store a unique input
                          // atom in alphabet.
    t_state *states;      // An array of t_state; oracle information is stored
                          // here.
    long output_length;   // The length of the output string.
    double probability;   // Probability of congruence between input and output
                          // strings (closer to 1.0 is more congruent).
    t_atom *output;       // The output string as an array of atoms.

} t_factorOracle;


//////////////////////////////// Required function prototypes.

void *factorOracle_new(t_symbol *s, long argc, t_atom *argv);
void factorOracle_free(t_factorOracle *x);
void factorOracle_assist(t_factorOracle *x, void *b, long m, long a, char *s);


//////////////////////////////// Functions that respond to MAX GUI input.

void factorOracle_in1(t_factorOracle *x, double value);
void factorOracle_in2(t_factorOracle *x, long value);
void factorOracle_anything
     (t_factorOracle *x, t_symbol *s, long argc, t_atom *argv);
void factorOracle_list(t_factorOracle *x, t_symbol *s, long argc, t_atom *argv);
void factorOracle_float(t_factorOracle *x, double value);
```

```
void factorOracle_int(t_factorOracle *x, long value);
void factorOracle_bang(t_factorOracle *x);
void factorOracle_clear(t_factorOracle *x);


//////////////////////////////// Internal functions.

long longMemberOfAlphabet(long a, t_factorOracle *x);
long floatMemberOfAlphabet(double a, t_factorOracle *x);
long symbolMemberOfAlphabet(t_symbol *a, t_factorOracle *x);
long longMemberOfTransitionElements(long transition, long k, t_factorOracle *x);
void buildOracle(long transition, t_factorOracle *x);
void listAlphabet(t_factorOracle *x);
void listOracleData(t_factorOracle *x);
void listOutput(t_factorOracle *x);


//////////////////////////////// Global class pointer variable.

void *factorOracle_class;


//////////////////////////////// Function definitions.

int main(void)
{
    t_class *c;

    c = class_new("factorOracle", (method)factorOracle_new,
        (method)factorOracle_free, (long)sizeof(t_factorOracle), 0L, A_GIMME, 0
        );
    class_addmethod(c, (method)factorOracle_assist, "assist", A_CANT, 0);

    // Probability of congruence between input and output strings.
    class_addmethod(c, (method)factorOracle_in1, "in1", A_FLOAT, 0);

    class_addmethod(c, (method)factorOracle_in2, "in2", A_LONG, 0);

    // The length of the output string.
    class_addmethod(c, (method)factorOracle_anything, "anything", A_GIMME, 0);

    class_addmethod(c, (method)factorOracle_list, "list", A_GIMME, 0);
    class_addmethod(c, (method)factorOracle_float, "float", A_FLOAT, 0);
    class_addmethod(c, (method)factorOracle_int, "int", A_LONG, 0);

    // Defer to the back of the queue.
    class_addmethod(c, (method)factorOracle_bang, "bang", A_DEFER, 0);
    class_addmethod(c, (method)factorOracle_clear, "clear", A_NOTHING, 0);

    class_register(CLASS_BOX, c);
    factorOracle_class = c;
```

```
    return 0;

}

////////////////////////////////

void *factorOracle_new(t_symbol *s, long argc, t_atom *argv)
{
    t_factorOracle *x = NULL;

    if (x = (t_factorOracle *)object_alloc(factorOracle_class)) {
        object_post((t_object *)x, "A new %s object was instantiated: 0x%X.",
        s->s_name, x);

        intin(x, 1);
        intin(x, 2);
        x->m_outlet1 = listout((t_object *)x);
        x->input_index = 0;
        x->alphabet_index = 0;

        // Argument 1
        if ((argc >= 1) && ((argv)->a_type == A_LONG)) {
            long nAtoms = atom_getlong(argv);
            if (nAtoms >= 1) {
                object_post((t_object *)x,
                    "Number of atoms allocated for input string: %ld.", nAtoms);
                // Note: sysmem_newptrclear returns a pointer of type char --
                // cast it to t_atom.
                x->alphabet =
                    (t_atom*)sysmem_newptrclear(nAtoms * sizeof(t_atom));
                x->states =
                    (t_state*)sysmem_newptrclear(nAtoms * sizeof(t_state));
                x->input_size = nAtoms;
            } else {
                object_post((t_object *)x, "Argument 1 must be an integer, at
                    least 1, specifying the number of atoms allocated for input
                    string. Allocating default: %ld.", 1000000);
                x->alphabet =
                    (t_atom*)sysmem_newptrclear(1000000 * sizeof(t_atom));
                x->states =
                    (t_state*)sysmem_newptrclear(1000000 * sizeof(t_state));
                x->input_size = 1000000;
            }
        } else {
            object_post((t_object *)x, "Argument 1 must be an integer, at least
                1, specifying the number of atoms allocated for input string.
                Allocating default: %ld.", 1000000);
            x->alphabet = (t_atom*)sysmem_newptrclear(1000000 * sizeof(t_atom));
            x->states = (t_state*)sysmem_newptrclear(1000000 * sizeof(t_state));
            x->input_size = 1000000;
```

```
    }

    // Argument 2
    if ((argc >= 2) && ((argv+1)->a_type == A_LONG)) {
        long oLength = atom_getlong(argv+1);
        object_post((t_object *)x,
            "Length of output string: %ld.", oLength);
        x->output_length = oLength;
    } else {
        object_post((t_object *)x, "Argument 2 must be an integer specifying
            the length of the output string. Setting to default: %ld.", 5);
        x->output_length = 5; // Default output length.
    }
    // Allocate memory for output array.
    x->output =
        (t_atom*)sysmem_newptrclear(x->output_length * sizeof(t_atom));

    // Argument 3
    if ((argc >= 3) && ((argv+2)->a_type == A_FLOAT)) {
        double probS = atom_getfloat(argv+2);
        if ((0 <= probS) && (probS <= 1)) {
            object_post((t_object *)x, "Probability of congruence between
                input and output: %f.", atom_getfloat(argv+2));
            x->probability = atom_getfloat(argv+2);
        } else {
            object_post((t_object *)x, "Argument 3 must be a float between 0
                and 1 (inclusive) specifying the probability of congruence
                between input and output. Setting to default: %f.", 1.0);
            // Default probability -- all forward links, most similar to
            // input:
            x->probability = 1.0;
        }
    } else {
        object_post((t_object *)x, "Argument 3 must be a float between 0 and
            1 (inclusive) specifying the probability of congruence between
            input and output. Setting to default: %f.", 1.0);
        // Default probability -- all forward links, most similar to
        // input:
        x->probability = 1.0;
    }

    // Warn about extra arguments.
    if (argc >= 4) {
        // inlet_new(x, "bang"); // Use this function to sprout new inlets.
        object_post((t_object *)x, "Ignoring extra arguments (up to three
            optional arguments may be specified).");
    }

}
// Rough maximum possible size for default factorOracle object (fO class +
// 1000000 state structs + 1000000 atoms (maximum alphabet size) + 2x1000000
```

```
    // long arrays (maximum sizes for transitionsElements and
    // transitionEndStates)) is approximately 34MB.
    /*
    post("Memory ceiling for default factorOracle = %ld MB",
        ((sizeof(t_factorOracle)+(1000000 * sizeof(t_state))+
        (1000000 * sizeof(t_atom))+(2000000 * sizeof(long))) / (1024 * 1024)));
    */

    // Seed for rand() function; to get the most variance, seed once and use
    // rand() as many times as possible -- a new call for each random number
    // desired.
    srand((unsigned)time(NULL));

    return (x);

}

////////////////////////////////

void factorOracle_free(t_factorOracle *x)
{

    // Put all free() calls here, for when the factorOracle object is destroyed.

    long i;

    sysmem_freeptr(x->output);
    sysmem_freeptr(x->alphabet);

    // You must clear all the arrays pointed to by each struct state in states
    // before you clear states, or the memory associated with the struct arrays
    // will persist.  Unless you specifically allocate array sizes on the struct
    // (i.e. "long array[20]), the struct is just POINTING to arrays -- these
    // arrays have to be explicitly freed before freeing the struct holding them.
    for (i = 0; i < x->input_index; i ++) {
        sysmem_freeptr(x->states[i].transitionElements);
        sysmem_freeptr(x->states[i].transitionEndStates);
    }
    sysmem_freeptr(x->states);

}

////////////////////////////////

void factorOracle_assist(t_factorOracle *x, void *b, long m, long a, char *s)
{

    if (m == ASSIST_INLET) { // inlets
        switch (a) {
            case 0: sprintf(s, "Anything; \"clear\" message resets the oracle;
                bang causes output."); break;
```

```
            case 1: sprintf(s, "Integer [ i > 0 ]; specifies the number of
                elements in the output string."); break;
            case 2: sprintf(s, "Float [ 0.0 - 1.0 ]; specifies the probability
                 of congruence between input and output strings."); break;
        }
    }
    else { // outlets
        sprintf(s, "Output string as a list of elements.");
    }

}

////////////////////////////////

void factorOracle_in1(t_factorOracle *x, double value)
{

    post("probability before: %f", value);
    if (value > 1) {
        x->probability = 1.0;
    } else if (value < 0) {
        x->probability = 0;
    } else {
        x->probability = value;
    }
    post("probability after = %f", x->probability);
    /*
    //post("value = %f", value);
    //post("float value = %f", (float)value);
    //post("double value = %f", (double)value);
    //post("long value = %ld", (long)value);
    //post("value minus 1.0 = %f", (value - 1.0));
    if ((long)value == 1) {
        x->probability = value;
        post("got here");
    } else {
        x->probability = (fmod(fabs(value), 1.0));
    }
    */

}

////////////////////////////////

void factorOracle_in2(t_factorOracle *x, long value)
{

    if (value == 0) {
        x->output_length = 1;
    } else {
        x->output_length = abs(value);
```

```
    }

    critical_enter(0);

    x->output = (t_atom*)sysmem_resizeptrclear(x->output,
        (x->output_length * sizeof(t_atom)));

    critical_exit(0);

    //post("output_length = %ld", x->output_length);

}

////////////////////////////////

// index 0 - x is true, -1 is false (not a member)

long longMemberOfAlphabet(long a, t_factorOracle *x)
{

    long test = -1;
    long i;
    t_atom *ap;
    for (i = 0, ap = x->alphabet; i < x->alphabet_index; i++, ap++) {
        if ((ap)->a_type == A_LONG) {
            if (a == atom_getlong(ap)) {
                test = i;
                break;
            }
        }
    }
    return test;

}

////////////////////////////////

// index 0 - x is true, -1 is false (not a member)

long floatMemberOfAlphabet(double a, t_factorOracle *x)
{

    long test = -1;
    long i;
    t_atom *ap;
    for (i = 0, ap = x->alphabet; i < x->alphabet_index; i++, ap++) {
        if ((ap)->a_type == A_FLOAT) {
            if (a == atom_getfloat(ap)) {
                test = i;
                break;
            }
```

```
        }
    }
    return test;

}

////////////////////////////////

// index 0 - x is true, -1 is false (not a member)

long symbolMemberOfAlphabet(t_symbol *a, t_factorOracle *x)
{

    long test = -1;
    long i;
    char *asn;
    asn = a->s_name;
    long alength = strlen(asn);
    t_atom *ap;
    for (i = 0, ap = x->alphabet; i < x->alphabet_index; i++, ap++) {
        // If both items being compared are symbols and both are the same
        // length, check if they have identical sequences of characters.
        if ((ap)->a_type == A_SYM) {
            t_symbol *s;
            s = atom_getsym(ap);
            char *sn;
            sn = s->s_name;
            if (alength == (strlen(sn)))  {
                long j;
                for  (j = 0; j < alength; j++) {
                    if (asn[j] != sn[j]) {
                        break;
                    } else if (j == (alength - 1)) {
                        test = i;
                        break;
                    }
                }
            }
        }
        if (test == i) {
            break;
        }
    }
    return test;

}

////////////////////////////////

// index 0 - x is true, -1 is false (not a member)
```

```
long longMemberOfTransitionElements(long transition, long k, t_factorOracle *x)
{

    long test = -1;
    long i;
    for (i = 0; i < x->states[k].numberOfTransitionElements; i++) {
        if (transition == x->states[k].transitionElements[i]) {
            test = i;
            break;
        }
    }
    return test;

}

////////////////////////////////

void buildOracle(long transition, t_factorOracle *x)
{

    x->states[x->input_index].transitionElements =
        (long*)sysmem_newptrclear(sizeof(long));
    x->states[x->input_index].transitionEndStates =
        (long*)sysmem_newptrclear(sizeof(long));
    x->states[x->input_index].transitionElements[0] = transition;
    x->states[x->input_index].transitionEndStates[0] = (x->input_index + 1);
    x->states[x->input_index].numberOfTransitionElements = 1;

    long k;
    if (x->input_index == 0) {
        x->states[x->input_index].suffixLink = -1;
        k = -1;
    } else {
        k = x->states[x->input_index].suffixLink;
    }

    while ((k != -1) && ((longMemberOfTransitionElements(transition, k, x)) ==
        -1)) {

        x->states[k].transitionEndStates =
            (long*)sysmem_resizeptr(x->states[k].transitionEndStates,
            (x->states[k].numberOfTransitionElements+1) * sizeof(long));
        x->states[k].transitionElements =
            (long*)sysmem_resizeptr(x->states[k].transitionElements,
            (x->states[k].numberOfTransitionElements+1) * sizeof(long));
        // Add a forward link from state k to current state + 1 by transition.
        x->states[k].transitionEndStates[
            x->states[k].numberOfTransitionElements]
                = (x->input_index + 1);
        x->states[k].transitionElements[x->states[k].numberOfTransitionElements]
            = transition;
```

```
        x->states[k].numberOfTransitionElements =
            (x->states[k].numberOfTransitionElements + 1);
        // Follow the suffix link back from state k.
        k = x->states[k].suffixLink;

    }

    if (k == -1) {

        x->states[x->input_index+1].suffixLink = 0;

    } else {
        // When we find the current transition symbol is already coming from k,
        // set the suffix link from the current state + 1 to the state that
        // the transition symbol points to from k.
        x->states[(x->input_index + 1)].suffixLink =
            x->states[k].transitionEndStates[longMemberOfTransitionElements(
                transition,k,x)];

    }

}

////////////////////////////////

void factorOracle_list(t_factorOracle *x, t_symbol *s, long argc, t_atom *argv)
{

    //post("List method called.");
    //post("Message selector is %s.", s->s_name);
    //post("There are %ld arguments.", argc);

    critical_enter(0);

    // ONLY ALLOW IN NEW ATOMS IF WE ARE UNDER THE INPUT BUFFER SIZE.
    long loopBelow;
    if ((x->input_index + argc) < x->input_size) {
        loopBelow = argc;
    } else {
        loopBelow = (x->input_size - x->input_index);
    }

    long i;
    long transition;
    t_atom *ap;
    for (i = 0, ap = argv; i < loopBelow; i++, ap++) {
        if ((ap)->a_type == A_LONG) {
            long l = atom_getlong(ap);
            transition = longMemberOfAlphabet(l, x);
            // If l is not a member of the alphabet, add it to the alphabet, add
            // the current alphabet_index to the oracle, and increment
```

```
            // alphabet_index.
            if (transition == -1) {
                atom_setlong((x->alphabet+(x->alphabet_index)), l);
                buildOracle(x->alphabet_index, x);
                x->alphabet_index = (x->alphabet_index + 1);
            } else {
                // Otherwise, add to the oracle the alphabet_index at which l is
                // already stored.
                buildOracle(transition, x);
            }
        } else if ((ap)->a_type == A_FLOAT) {
            double f = atom_getfloat(ap);
            transition = floatMemberOfAlphabet(f, x);
            // If f is not a member of the alphabet, add it to the alphabet, add
            // the current alphabet_index to the oracle, and increment
            // alphabet_index.
            if (transition == -1) {
                atom_setfloat((x->alphabet+(x->alphabet_index)), f);
                buildOracle(x->alphabet_index, x);
                x->alphabet_index = (x->alphabet_index + 1);
            } else {
                // Otherwise, add to the oracle the alphabet_index at which f is
                // already stored.
                buildOracle(transition, x);
            }
        } else if ((ap)->a_type == A_SYM) {
            t_symbol *s = atom_getsym(ap);
            transition = symbolMemberOfAlphabet(s, x);
            // If s is not a member of the alphabet, add it to the alphabet, add
            // the current alphabet_index to the oracle, and increment
            // alphabet_index.
            if (transition == -1) {
                atom_setsym((x->alphabet+(x->alphabet_index)), atom_getsym(ap));
                buildOracle(x->alphabet_index, x);
                x->alphabet_index = (x->alphabet_index + 1);
            } else {
                // Otherwise, add to the oracle the alphabet_index at which s is
                // already stored.
                buildOracle(transition, x);
            }
        } else {
            object_error((t_object *)x, "Forbidden argument.");
        }
        // Increment input index each time.
        x->input_index = (x->input_index + 1);
    }

    critical_exit(0);

}
```

```
////////////////////////////////

void factorOracle_anything(t_factorOracle *x, t_symbol *s, long argc,
    t_atom *argv)
{

    //post("Anything method called.");
    //post("Message selector is %s.", s->s_name);
    //post("There are %ld arguments.", argc);

    critical_enter(0);

    // ONLY ALLOW IN NEW ATOMS IF WE ARE UNDER THE INPUT BUFFER SIZE.
    if (x->input_index < x->input_size) {
        // Add the "message selector" symbol to the alphabet, if
        // necessary, and the oracle.
        long transition = symbolMemberOfAlphabet(s, x);
        // If s is not a member of the alphabet, add it to the alphabet, add the
        // current alphabet_index to the oracle, and increment alphabet_index.
        if (transition == -1) {
            atom_setsym((x->alphabet+(x->alphabet_index)), s);
            buildOracle(x->alphabet_index, x);
            x->alphabet_index = (x->alphabet_index + 1);
        } else {
            // Otherwise, add to the oracle the alphabet_index at which s is
            // already stored.
            buildOracle(transition, x);
        }
        // Increment input index.
        x->input_index = (x->input_index + 1);

    }

    critical_exit(0);

    // Call _list on remaining arguments.
    factorOracle_list(x, gensym("list"), argc, argv);

}

////////////////////////////////

void factorOracle_float(t_factorOracle *x, double value)
{

    //post("Float method called.");

    critical_enter(0);

    // ONLY ALLOW IN NEW ATOMS IF WE ARE UNDER THE INPUT BUFFER SIZE.
    if (x->input_index < x->input_size) {
```

```
        long transition;
        transition = floatMemberOfAlphabet(value, x);
        // If transition is not a member of the alphabet, add it to the
        // alphabet, add the current alphabet_index to the oracle, and increment
        // alphabet_index.
        if (transition == -1) {
            atom_setfloat((x->alphabet+(x->alphabet_index)), value);
            buildOracle(x->alphabet_index, x);
            x->alphabet_index = (x->alphabet_index + 1);
        } else {
            // Otherwise, add to the oracle the alphabet_index at which
            // transition is already stored.
            buildOracle(transition, x);
        }
        // Increment input index.
        x->input_index = (x->input_index + 1);

    }

    critical_exit(0);

}

////////////////////////////////

void factorOracle_int(t_factorOracle *x, long value)
{

    //post("Int method called.");

    critical_enter(0);

    // ONLY ALLOW IN NEW ATOMS IF WE ARE UNDER THE INPUT BUFFER SIZE.
    if (x->input_index < x->input_size) {

        long transition;
        transition = longMemberOfAlphabet(value, x);
        // If transition is not a member of the alphabet, add it to the
        // alphabet, add the current alphabet_index to the oracle, and increment
        // alphabet_index.
        if (transition == -1) {
            atom_setlong((x->alphabet+(x->alphabet_index)), value);
            buildOracle(x->alphabet_index, x);
            x->alphabet_index = (x->alphabet_index + 1);
        } else {
            // Otherwise, add to the oracle the alphabet_index at which
            // transition is already stored.
            buildOracle(transition, x);
        }
        // Increment input index.
```

```
        x->input_index = (x->input_index + 1);

    }

    critical_exit(0);

}

////////////////////////////////

void listAlphabet(t_factorOracle *x)
{

    long i;
    for (i = 0; i < x->alphabet_index; i++) {
        if ((x->alphabet+i)->a_type == A_LONG) {
            post("alphabet element %ld: long (%ld)", i,
                atom_getlong(x->alphabet+i));
        } else if ((x->alphabet+i)->a_type == A_FLOAT) {
            post("alphabet element %ld: float (%f)", i,
                atom_getfloat(x->alphabet+i));
        } else if ((x->alphabet+i)->a_type == A_SYM) {
            post("alphabet element %ld: symbol (%s)", i,
                atom_getsym(x->alphabet+i)->s_name);
        } else {
            post("alphabet element %ld: unknown type (%s)", i,
                ((x->alphabet+i)->a_type));
        }
    }
    post("alphabet_index = %ld", x->alphabet_index);

}

////////////////////////////////

void listOracleData(t_factorOracle *x)
{
    long i;
    long j;
    for (i = 0; i < x->input_index; i++) {
        long nte = x->states[i].numberOfTransitionElements;
        long suf = x->states[i].suffixLink;
        post("State %ld numberOfTransitionElements = %ld.", i, nte);
        post("State %ld suffixLink points to state %ld.", i, suf);
        for (j = 0; j < nte; j++) {
            t_atom *trn;
            trn = x->alphabet+(x->states[i].transitionElements[j]);
            long end = x->states[i].transitionEndStates[j];
            if (trn->a_type == A_LONG) {
                long ltrn = atom_getlong(trn);
                post("    State %ld transition element %ld points to state
```

```
                    %ld.", i, ltrn, end);
            } else if (trn->a_type == A_FLOAT) {
                double ftrn = atom_getfloat(trn);
                post("    State %ld transition element %f points to state %ld.",
                    i, ftrn, end);
            } else if (trn->a_type == A_SYM) {
                char *strn;
                strn = atom_getsym(trn)->s_name;
                post("    State %ld transition element %s points to state %ld.",
                    i, strn, end);
            } else {
                post("    A bad argument (other than float, int, or symbol) of
                    type %s got added to state %ld transitionElements.",
                    trn->a_type, i);
            }
        }
    }
    long fnte = x->states[x->input_index].numberOfTransitionElements;
    long fsuf = x->states[x->input_index].suffixLink;
    post("Final state (%ld) numberOfTransitionElements = %ld.", i, fnte);
    post("Final state (%ld) suffixLink points to state %ld.", i, fsuf);
    post("input_index = %ld", x->input_index);

}

////////////////////////////////

void listOutput(t_factorOracle *x)
{

    long i;
    for (i = 0; i < x->output_length; i++) {
        if ((x->output+i)->a_type == A_LONG) {
            post("output element %ld: long (%ld)", i,
                atom_getlong(x->output+i));
        } else if ((x->output+i)->a_type == A_FLOAT) {
            post("output element %ld: float (%f)", i,
                atom_getfloat(x->output+i));
        } else if ((x->output+i)->a_type == A_SYM) {
            post("output element %ld: symbol (%s)", i,
                atom_getsym(x->output+i)->s_name);
        } else {
            post("output element %ld: unknown type (%s)", i,
                ((x->output+i)->a_type));
        }
    }

}

////////////////////////////////
```

```
void factorOracle_bang(t_factorOracle *x)
{

    //critical_enter(0);

    if (x->input_index > 0) {

        critical_enter(0);

        // PATTERN MATCHING TESTS
        /*
        long li;
        long l = 123;
        long fi;
        double f = 0.123;
        long si;
        t_atom ts[1];
        atom_setsym(ts, gensym("abc"));
        t_symbol *s = atom_getsym(ts);

        // test long matcher
        li = longMemberOfAlphabet(l, x);
        post("123 is a member of the alphabet? == %ld", li);

        // test float matcher
        fi = floatMemberOfAlphabet(f, x);
        post("0.123 is a member of the alphabet? == %ld", fi);

        // test symbol matcher
        si = symbolMemberOfAlphabet(s, x);
        post("abc is a member of the alphabet? == %ld", si);
        */

        //post("input_index = %ld", x->input_index);

        long accum = 0;
        long teIndex;

        // Start from the state pointed to by the last element entered.
        long state_index = x->input_index;
        while (accum < x->output_length) {
            long r1 = rand();
            long r2 = rand();
            // If we're at the last state in the input string, we must take its
            // suffix link.
            if (state_index == x->input_index) {
                //post("last state");
                state_index = x->states[state_index].suffixLink;
            // If we're at the first state in the input string, we can't take a
            // suffix link and must choose a random forward link.
            } else if (state_index == 0) {
```

```
        //post("first state");
        teIndex =
         (r2%(x->states[state_index].numberOfTransitionElements));
        x->output[accum] =
            x->alphabet[x->states[state_index].transitionElements[
                teIndex]];
        state_index = x->states[state_index].transitionEndStates[
            teIndex];
        accum = (accum + 1);
    // Otherwise . . .
    // If a random value [ x : 0 <= x < maximum_input_probability ] is
    // less than the input probability, then there is a greater chance
    // that the input probability is closer to 1.0, which means that we
    // want to be closer to the original string and must choose a random
    // forward link.
    } else if ((r1%1000000) < ((long)(1000000.0 * x->probability))) {
        //post("forward link");
        teIndex =
         (r2%(x->states[state_index].numberOfTransitionElements));
        x->output[accum] =
            x->alphabet[x->states[state_index].transitionElements[
                teIndex]];
        state_index = x->states[state_index].transitionEndStates[
            teIndex];
        accum = (accum + 1);
    // If a random value [ x : 0 <= x < maximum_input_probability ] is
    // greater than or equal to the input probability, there is a
    // greater chance that the input probability is closer to 0.0, which
    // means we want to be farther away from the original string and
    // must choose the suffix link.
    } else {
        //post("suffix link");
        state_index = x->states[state_index].suffixLink;
    }
}

// IF X->PROBABILITY = 0, YOU WILL JUST OSCILLATE BETWEEN STATE ZERO'S
// IMMEDIATE FORWARD LINKS, BECAUSE, WHEREVER YOU'RE AT, THE ORACLE WILL
// JUST TAKE THE ENTIRE SUFFIX CHAIN BACK TO STATE ZERO, GO FORWARD BY
// ONE TRANSITION ELEMENT, THEN REPEAT.

// Checks:

// This console spew eventually eats up all of your memory.
//listAlphabet(x);
//listOracleData(x);
//listOutput(x);

// Kick the output out an inlet as a list; you don't want to fire out
// individual elements -- the only reason to do this is if you have MIDI
// elements and can fire notes off according to duration; eventually you
```

```
        // can make MIDI-specific version of f0 that does this, when you have
        // implemented cross-alphabet support.

        // CAN'T HAVE OUTLET CALLS INSIDE CRITICAL AREAS
        critical_exit(0);

        outlet_list(x->m_outlet1,NULL,x->output_length,x->output);

    }

    //critical_exit(0);

}

////////////////////////////////

void factorOracle_clear(t_factorOracle *x)
{

    //object_post((t_object *)x, "Clear method called.");

    long i;

    critical_enter(0);

    sysmem_freeptr(x->output);
    sysmem_freeptr(x->alphabet);
    x->output = (t_atom*)sysmem_newptrclear(x->output_length * sizeof(t_atom));
    x->alphabet = (t_atom*)sysmem_newptrclear(x->input_size * sizeof(t_atom));

    // You must clear all the arrays pointed to by each struct _state in states
    // before you clear states, or the memory associated with the struct arrays
    // will persist.  Unless you specifically allocate array sizes on the struct
    // (i.e. "long array[20]), the struct is just POINTING to arrays -- these
    // arrays have to be explicitly freed before freeing the struct holding
    // their pointers.
    for (i = 0; i < x->input_index; i ++) {
        sysmem_freeptr(x->states[i].transitionElements);
        sysmem_freeptr(x->states[i].transitionEndStates);
    }
    sysmem_freeptr(x->states);
    x->states = (t_state*)sysmem_newptrclear(x->input_size * sizeof(t_state));

    x->alphabet_index = 0;
    x->input_index = 0;

    critical_exit(0);

}

//////////////////////////////// eof
```

A.2 quantizeAlphabet.c

```
////////////////////////////////////////////////////////////////////////////////
//
//  quantizeAlphabet, a MAX external
//  Adam James Wilson
//  ajwilson@ucsd.edu
//
////////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////// License.

/*
This software is copyrighted by Adam James Wilson and others. The following
terms (the "Standard Improved BSD License") apply. Redistribution and use in
source and binary forms, with or without modification, are permitted provided
that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright  notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived
   from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
*/


//////////////////////////////// Description.

/*
Maps a cross-alphabet of element pairs to a specified range.
*/


//////////////////////////////// Dependencies.

#include "ext.h"            // This include is always required for MAX.
#include "ext_obex.h"       // This include is required for newstyle MAX objects.
```

```
#include "ext_systhread.h" // MAX thread-locking functions.


//////////////////////////////// Type definitions.

typedef struct _quantizeAlphabet
{

    t_object ob;
    long l_in;
    void *l_proxy;
    long m_in;
    void *m_proxy;
    long n_in;
    void *n_proxy;
    void *m_outlet1;
    long size;
    long min_dur; // milliseconds
    long max_dur; // milliseconds
    double min_key; // 40 for guitar
    double max_key; // 88 for guitar
    double min_cross_alphabet_number;
    double max_cross_alphabet_number;
    double range;
    long shift;
    long dur;
    double key;

} t_quantizeAlphabet;


//////////////////////////////// Required function prototypes.

void *quantizeAlphabet_new(t_symbol *s, long argc, t_atom *argv);
void quantizeAlphabet_free(t_quantizeAlphabet *x);
void quantizeAlphabet_assist(t_quantizeAlphabet *x, void *b, long m, long a,
    char *s);


//////////////////////////////// Functions that respond to MAX GUI input.

void quantizeAlphabet_float(t_quantizeAlphabet *x, double value);
void quantizeAlphabet_int(t_quantizeAlphabet *x, long value);


//////////////////////////////// Internal functions.

void quantizeAlphabet_map(t_quantizeAlphabet *x);


//////////////////////////////// Global class pointer variable.
```

```
void *quantizeAlphabet_class;


//////////////////////////////// Function definitions.

int main(void)
{

    t_class *c;
    c = class_new("quantizeAlphabet", (method)quantizeAlphabet_new,
        (method)quantizeAlphabet_free, (long)sizeof(t_quantizeAlphabet), 0L,
        A_GIMME, 0);

    class_addmethod(c, (method)quantizeAlphabet_float, "float", A_FLOAT, 0);
    class_addmethod(c, (method)quantizeAlphabet_int, "int", A_LONG, 0);

    class_register(CLASS_BOX, c); /* CLASS_NOBOX */
    quantizeAlphabet_class = c;

    return 0;

}

////////////////////////////////

void *quantizeAlphabet_new(t_symbol *s, long argc, t_atom *argv)
{

    t_quantizeAlphabet *x = NULL;

    if (x = (t_quantizeAlphabet *)object_alloc(quantizeAlphabet_class)) {
        object_post((t_object *)x, "A new %s object was instantiated: 0x%X.",
        s->s_name, x);

        // rightmost inlet (size argument)
        x->l_proxy = proxy_new((t_quantizeAlphabet *)x, 1, &x->l_in);

        // second-from-right inlet (shift amount)
        x->m_proxy = proxy_new((t_quantizeAlphabet *)x, 2, &x->m_in);

        // third-from-right inlet (duration argument)
        x->n_proxy = proxy_new((t_quantizeAlphabet *)x, 3, &x->n_in);

        // leftmost inlet (frequency argument, causes output)
        x->m_outlet1 = intout((t_object *)x);

        // Make these variables settable in future versions.
        x->min_dur = 83;
        x->max_dur = 4000;
        x->min_key = 40.0;
```

```
        x->max_key = 88.0;
        x->min_cross_alphabet_number = ((double)x->min_dur * x->min_key);
        x->max_cross_alphabet_number = ((double)x->max_dur * x->max_key);
        x->range =
            (x->max_cross_alphabet_number - x->min_cross_alphabet_number);

        // Argument 1
        if ((argc >= 1) && ((argv)->a_type == A_LONG)) {
            x->size = atom_getlong(argv);
        } else {
            object_post((t_object *)x, "Argument 1 must be an integer,
            specifying the size of the quantization list. Setting size to
            default: 10.");
            x->size = 10;
        }

        // Argument 2
        if ((argc >= 2) && ((argv+1)->a_type == A_LONG)) {
            x->shift = atom_getlong(argv+1);
        } else {
            object_post((t_object *)x, "Argument 2 must be an integer,
            specifying the a shift amount for cross-alphabet to file mappings.
            Setting size to default: 0 (no shift).");
            x->shift = 0;
        }

    }

    return (x);

}

////////////////////////////////

void quantizeAlphabet_assist(t_quantizeAlphabet *x, void *b, long m, long a,
    char *s)
{
    switch (a) {
        case 0: sprintf(s, "MIDI keynum.cents to be scaled between 40.00 and
            88.00."); break;
        case 1: sprintf(s, "Duration in milliseconds to be scaled between 83
            and 4000."); break;
        case 2: sprintf(s, "Shift amount (small number integer) used to change
            input/map relations."); break;
        case 3: sprintf(s, "Size of the output cross-alphabet."); break;
    }

}

////////////////////////////////
```

```
void quantizeAlphabet_free(t_quantizeAlphabet *x)
{

    // Proxies and clocks have to be freed using freeobject.
    freeobject((t_object *)x->l_proxy);
    freeobject((t_object *)x->m_proxy);
    freeobject((t_object *)x->n_proxy);

}


////////////////////////////////

void quantizeAlphabet_float(t_quantizeAlphabet *x, double value)
{

    switch (proxy_getinlet((t_object *)x)) {
        case 0:
            // MIDI keynum.cents.
            //post("MIDI keynum.cents value received in leftmost inlet.");
            x->key = value;
            quantizeAlphabet_map(x);
            break;
        case 1:
            // Size.
            //post("Size value received in the rightmost inlet.");
            x->size = (long)value;
            break;
        case 2:
            // Shift.
            //post("Shift value received in the inlet second from the right.");
            x->shift = (long)value;
            break;
        case 3:
            // Duration.
            //post("Duration value received in the inlet second from the
            //left.");
            //post("value = %f", value);
            //post("long value = %ld", (long)value);
            //post("changed x dur to %ld", x->dur);
            x->dur = (long)value;
            break;
    }

}

////////////////////////////////

void quantizeAlphabet_int(t_quantizeAlphabet *x, long value)
{
```

```
    switch (proxy_getinlet((t_object *)x)) {
        case 0:
            // MIDI keynum.cents.
            //post("MIDI keynum.cents value received in leftmost inlet.");
            x->key = (double)value;
            quantizeAlphabet_map(x);
            break;
        case 1:
            // Size.
            //post("Size value received in the rightmost inlet.");
            x->size = value;
            break;
        case 2:
            // Shift.
            //post("Shift value received in the inlet second from the right.");
            x->shift = value;
            break;
        case 3:
            // Duration.
            //post("Duration value received in the inlet second from the
            //left.");
            x->dur = value;
            //post("changed x dur to %ld", x->dur);
            break;
    }

}

////////////////////////////////

void quantizeAlphabet_map(t_quantizeAlphabet *x)
{

    //post("x->durb = %ld", x->dur);
    //post("x->keyb = %f", x->key);

    if (x->dur > x->max_dur) {
        x->dur = x->max_dur;
    } else if (x->dur < x->min_dur) {
        x->dur = x->min_dur;
    }

    if (x->key > x->max_key) {
        x->key = x->max_key;
    } else if (x->key < x->min_key) {
        x->key = x->min_key;
    }

    double xAlpha;

    if (x->shift == 0) {
```

```
        xAlpha = ((double)x->dur * x->key);
    } else {
        xAlpha = (fmod((((double)x->dur * x->key) + (x->range / x->shift)),
            x->range) + x->min_cross_alphabet_number);
    }

    //post("x->dur = %ld", x->dur);
    //post("x->key = %f", x->key);
    //post("xAlpha = %f", xAlpha);

    long mapVal = ((long)(((xAlpha - x->min_cross_alphabet_number) / x->range) *
        (((double)x->size) - 1.0)) + 1);

    outlet_int(x->m_outlet1, mapVal);

    /*
     Shorter durations and lower notes produce lower cross alphabet numbers,
     higher numbers and higher durs produce higher x-alpha numbers; make this
     relationship shift as the piece progresses -- add an inlet for causing
     a shift.

     Map: lower numbers represent longer, lower sounds; higher numbers represent
     higher, faster sounds; timbre is cyclical. Make sure directory and file
     numbering in Max collections follows this pattern.

     001 gong, longest, lowest
     002 string, longest, lowest
     003 timbre3, longest, lowest
     004 flock, longest, lowest
     005 granulation, longest, lowest
     006 gong, second-longest, lowest
     007 string, second-longest, lowest
     008 timbre3, second-longest, lowest
     009 flock, second-longest, lowest
     010 granulation, second-longest, lowest
     ...
     096 gong, second-shortest, highest
     097 string, second-shortest, highest
     098 timbre3, second-shortest, highest
     099 flock, second-shortest, highest
     100 granulation, second-shortest, highest
     101 gong, shortest, highest
     102 string, shortest, highest
     103 timbre3, shortest, highest
     104 flock, shortest, highest
     105 granulation, shortest, highest
    */

}

//////////////////////////////// eof
```

A.3 phraseDetect.c

```
///////////////////////////////////////////////////////////////////////////////
//
//  phraseDetect, a MAX external
//  Adam James Wilson
//  ajwilson@ucsd.edu
//
///////////////////////////////////////////////////////////////////////////////


///////////////////////////////// License.

/*
This software is copyrighted by Adam James Wilson and others. The following
terms (the "Standard Improved BSD License") apply. Redistribution and use in
source and binary forms, with or without modification, are permitted provided
that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright  notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived
   from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
*/


///////////////////////////////// Description.

/*
    * Finds the median change and the maximum change in duration between each
      pair of consecutive notes in a fairly long musical sample.
    * User sets a threshold between the median and the maximum changes.
    * Every point exceeding the threshold is considered a potential phrase end.
    * Every pair of phrase beginnings and ends are examined -- if there are
      fewer than three notes between them (including the edges), they are
      discarded.
    * The same process is applied for register, and if the registral boundaries
```

```
      match up with the durational boundaries, we rank that phrase a stronger
      than phrases with no corresponding registral drops.
*/


//////////////////////////////// Notes and potential improvements:

/*
    1. Find repetitions by checking each new phrase with the set of previously
       output phrases and looking for similar contours in rhythm and pitch
       height (POTENTIALLY OUTSIDE OF PHRASEDETECT).
    2. NOTE: ADDING OBJECT ERRORS AND POSTS PRINTED TO THE CONSOLE CAN CAUSE
       CRASHES! HAVE ERRORS OCCUR SILENTLY (EXCEPT ON INITIALIZATION), BUT
       RETAIN OBJECT ERRORS THAT YOU CAN UNCOMMENT FOR DEBUGGING. ALSO, NOTE
       THAT YOU CAN'T WRAP CALLS TO OUTLET_ IN CRITICAL AREAS -- THEY ARE
       THREAD-SAFE BY DESIGN.
*/


//////////////////////////////// Dependencies.

#include "ext.h"           // This include is always required for MAX.
#include "ext_obex.h"      // This include is required for newstyle MAX objects.
#include "ext_systhread.h" // MAX thread-locking functions.


//////////////////////////////// Type definitions.

typedef struct _phrase
{

    double *frqs; // floats or longs cast to floats, [ 0.00 - 127.00,
                  // MIDI_keynum.cents]
    double *amps; // floats, [ 0.0 - 1.0, amplitude scalars ]
    long  *durs;  // longs, [ 0 - n, milliseconds ]
    long length;

} t_phrase;

////////////////////////////////

typedef struct _phraseDetect
{

    t_object ob; // The object itself (this must always come first inside the
                 // main object struct).
    long m_in;
    void *m_proxy;
    long n_in;
    void *n_proxy;
    double *frqs;
```

```
    double *amps;
    long  *durs;
    long frqs_length;
    long amps_length;
    long durs_length;
    long analysis_window;
    long minPhraseLength;
    long maxPhraseLength;
    long maxPhrasesLength;
    long maxDurLength;
    void *m_outlet1;
    void *m_outlet2;
    void *m_outlet3;
    void *m_outlet4;
    void *m_outlet5;
    void *m_outlet6;
    void *m_outlet7;
    void *m_outlet8;
    double fine_boundary;         // This is a coeff 0 - 1, a percentage within
                                  // the threshold range -- higher is closer to
                                  // max duration diff.
    double *register_differences;
    t_phrase *phrases_d;          // An array of d_phrases whose growth is
                                  // limited by analysis_window size and
                                  // minPhraseLength.
    t_phrase *phrases_drd;        // An array of drd_phrases whose growth is
                                  // limited by analysis_window size and
                                  // minPhraseLength.
    long drd_phrase_count;        // This counts drd_phrases on a
                                  // per/analysis_window basis.
    long d_phrase_count;          // This counts d_phrases on a
                                  // per/analysis_window basis.
    long drd_phrase_index;        // This keeps a running total of ALL
                                  // drd_phrases discovered during the object's
                                  // existence.
    long d_phrase_index;          // This keeps a running total of ALL d_phrases
                                  // discovered during the object's existence.

} t_phraseDetect;


//////////////////////////////// Required function prototypes.

void *phraseDetect_new(t_symbol *s, long argc, t_atom *argv);
void phraseDetect_free(t_phraseDetect *x);
void phraseDetect_assist(t_phraseDetect *x, void *b, long m, long a, char *s);


//////////////////////////////// Functions that respond to MAX GUI input.

void phraseDetect_list(t_phraseDetect *x, t_symbol *s, long argc, t_atom *argv);
```

```
void phraseDetect_float(t_phraseDetect *x, double value);
void phraseDetect_int(t_phraseDetect *x, long value);
void phraseDetect_ft1(t_phraseDetect *x, double value);
void phraseDetect_clear(t_phraseDetect *x);


//////////////////////////////// Internal functions.

long aMemberOfB(long l, long arr[], long arrsize);
void bubbleSort(long a[], long array_size);
void bubbleSortf(double a[], long array_size);
double phraseDetect_getDursThreshold(t_phraseDetect *x);
double phraseDetect_getFrqsThreshold(t_phraseDetect *x);
long phraseDetect_findMaxDur(t_phraseDetect *x, long startIndex, long endIndex);
void phraseDetect_crunch(t_phraseDetect *x);
void phraseDetect_output(t_phraseDetect *x);
void phraseDetect_clearNonEmptyInputBuffers(t_phraseDetect *x);
long phraseDetect_badFrqsNumber(t_phraseDetect *x, double f);
long phraseDetect_badAmpsNumber(t_phraseDetect *x, double f);
long phraseDetect_badDursNumber(t_phraseDetect *x, long l);
long phraseDetect_badFrqsArgList(t_phraseDetect *x, long argc, t_atom *argv);
long phraseDetect_badAmpsArgList(t_phraseDetect *x, long argc, t_atom *argv);
long phraseDetect_badDursArgList(t_phraseDetect *x, long argc, t_atom *argv);
void phraseDetect_frqsNumberIn(t_phraseDetect *x, double f);
void phraseDetect_ampsNumberIn(t_phraseDetect *x, double f);
void phraseDetect_dursNumberIn(t_phraseDetect *x, long l);
void phraseDetect_frqsListIn(t_phraseDetect *x, long argc, t_atom *argv);
void phraseDetect_ampsListIn(t_phraseDetect *x, long argc, t_atom *argv);
void phraseDetect_dursListIn(t_phraseDetect *x, long argc, t_atom *argv);


//////////////////////////////// Global class pointer variable.

void *phraseDetect_class;


//////////////////////////////// Function definitions.

int main(void)
{

    t_class *c;
    c = class_new("phraseDetect", (method)phraseDetect_new,
        (method)phraseDetect_free, (long)sizeof(t_phraseDetect), 0L, A_GIMME, 0
        );
    class_addmethod(c, (method)phraseDetect_assist, "assist", A_CANT, 0);
    class_addmethod(c, (method)phraseDetect_list, "list", A_GIMME, 0);
    class_addmethod(c, (method)phraseDetect_float, "float", A_FLOAT, 0);
    class_addmethod(c, (method)phraseDetect_int, "int", A_LONG, 0);
    class_addmethod(c, (method)phraseDetect_ft1, "ft1", A_FLOAT, 0);
    class_addmethod(c, (method)phraseDetect_clear, "clear", A_GIMME, 0);
```

```
    class_register(CLASS_BOX, c);
    phraseDetect_class = c;

    return 0;

}

////////////////////////////////

void *phraseDetect_new(t_symbol *s, long argc, t_atom *argv)
{

    t_phraseDetect *x = NULL;

    if (x = (t_phraseDetect *)object_alloc(phraseDetect_class)) {

        object_post((t_object *)x, "A new %s object was instantiated: 0x%X.",
            s->s_name, x);

        floatin(x, 1); // fine control (second from right inlet)

        // durs (third from right inlet)
        x->m_proxy = proxy_new((t_phraseDetect *)x, 2, &x->m_in);

        // amps (fourth from right inlet)
        x->n_proxy = proxy_new((t_phraseDetect *)x, 3, &x->n_in);

        // frqs (default inlet (0), leftmost inlet).

        x->frqs_length = 0;
        x->amps_length = 0;
        x->durs_length = 0;

        x->m_outlet1 = listout((t_object *)x);
        x->m_outlet2 = listout((t_object *)x);
        x->m_outlet3 = listout((t_object *)x);
        x->m_outlet4 = intout((t_object *)x);
        x->m_outlet5 = listout((t_object *)x);
        x->m_outlet6 = listout((t_object *)x);
        x->m_outlet7 = listout((t_object *)x);
        x->m_outlet8 = intout((t_object *)x);

        x->drd_phrase_count = 0; // initial value for drd_phrase_count
        x->d_phrase_count = 0;   // initial value for d_phrase_count
        x->drd_phrase_index = 0; // initial value for drd_phrase_index
        x->d_phrase_index = 0;   // initial value for d_phrase_index

     // Initialization args left to right: analysis window size, minimum
     // phrase length, maximimum phrase length, maximum allowable duration,
     // threshold.
```

```
// Argument 1
if ((argc >= 1) && ((argv)->a_type == A_LONG)) {
    long window_size = atom_getlong(argv);
    if (window_size >= 3) { // minimum allowable phrase length: 3 notes
            object_post((t_object *)x, "Setting analysis window size to
                %ld.", window_size);
            x->analysis_window = window_size;
    } else {
            object_post((t_object *)x, "Argument 1 must be an integer,
                at least 3, specifying analysis window size. Setting to
                default: %ld.", 32);
            x->analysis_window = 32; // default window size is 32 notes
    }
} else {
    object_post((t_object *)x, "Argument 1 must be an integer, at least
        3, specifying analysis window size. Setting to default: %ld.",
        32);
    x->analysis_window = 32; // default window size is 32 notes
}

// Argument 2
if ((argc >= 2) && ((argv+1)->a_type == A_LONG)) {
    long min_length = atom_getlong(argv+1);
    if ((min_length >= 3) && (min_length <= x->analysis_window)) {
        object_post((t_object *)x, "Setting minimum phrase length to
            %ld.", min_length);
        x->minPhraseLength = min_length;
    } else {
        object_post((t_object *)x, "Argument 2 must be an integer, at
            least 3 and less than or equal to analysis window size,
            specifying minimum notes per phrase. Setting to default:
            %ld.", 3);
        x->minPhraseLength = 3; // default minimum phrase length
                                // (phrases less than 3 notes long will
                                // be ignored)
    }
} else {
    object_post((t_object *)x, "Argument 2 must be an integer, at least
        3 and less than or equal to analysis window size, specifying
        minimum notes per phrase. Setting to default: %ld.", 3);
    x->minPhraseLength = 3; // default minimum phrase length (phrases
                            // less than 3 notes long will be ignored)
}
// The input list cannot be greater than analysis_window (there's no
// point), so the biggest list you can have before processing occurs is
// (2 * (analysis_window - 1)): 62 if analysis_window = 32. Therefore,
// calloc a number of t_phrase equal to ceiling[ (2 * (window_size - 1))
// / minPhraseLength ] and set phraseDetect_dursListIn so that it can't
// accept a list whose size is > analysis_window.
x->maxPhrasesLength = (long)(((double)(2 * (x->analysis_window - 1)) /
```

```
    (double)x->minPhraseLength) + 0.5);
x->phrases_d = (t_phrase*)sysmem_newptrclear(
    x->maxPhrasesLength * sizeof(t_phrase));
x->phrases_drd = (t_phrase*)sysmem_newptrclear(
    x->maxPhrasesLength * sizeof(t_phrase));
//object_post((t_object *)x, "maxPhrasesLength =  %ld.",
    x->maxPhrasesLength);

// Argument 3 -- max phrase length
if ((argc >= 3) && ((argv+2)->a_type == A_LONG)) {
    long max_length = atom_getlong(argv+2);
    if ((max_length >= x->minPhraseLength) &&
        (max_length <= x->analysis_window)) {
        object_post((t_object *)x, "Setting maximum phrase length to
            %ld.", max_length);
        x->maxPhraseLength = max_length;
    } else {
        object_post((t_object *)x, "Argument 3 must be an integer
            greater than or equal to minimum phrase length and less than
            or equal to analysis window size, specifying maximim phrase
            length. Setting to default (window size): %ld.",
            x->analysis_window);

        // default maximum phrase length (phrases longer than
        // x->analysis_window will be ignored)
        x->maxPhraseLength = x->analysis_window;
    }
} else {
    object_post((t_object *)x, "Argument 3 must be an integer greater
        than or equal to minimum phrase length and less than or equal to
        analysis window size, specifying maximim phrase length. Setting
        to default (window size): %ld.", x->analysis_window);

    // default maximum phrase length (phrases longer than
    // x->analysis_window will be ignored)
    x->maxPhraseLength = x->analysis_window;
}

// Argument 4 -- max duration length
if ((argc >= 4) && ((argv+3)->a_type == A_LONG)) {
    long max_dlength = atom_getlong(argv+3);
    if (max_dlength > 0) {
        object_post((t_object *)x, "Setting maximum duration to %ld.",
            max_dlength);
        x->maxDurLength = max_dlength;
    } else {
        object_post((t_object *)x, "Argument 4 must be an integer
            greater than 0 specifying the maximum duration allowed in a
            phrase. Setting to default: %ld.", 600000);
        x->maxDurLength = 600000; // default maximum duration is 10
                                  // minutes (phrases with longer
```

```
                                        // durations will be ignored)
            }
        } else {
            object_post((t_object *)x, "Argument 4 must be an integer greater
                than 0 specifying the maximum duration allowed in a phrase.
                Setting to default: %ld.", 600000);
            x->maxDurLength = 600000; // default maximum duration is 10 minutes
                                      // (phrases with longer durations will be
                                      // ignored)
        }

        // Argument 5 -- initital threshold setting between median and maximum
        // durations, register differences.
        if ((argc >= 5) && ((argv+4)->a_type == A_FLOAT)) {
            double threshold = atom_getfloat(argv+4);
            if ((threshold >= 0.0) && (threshold <= 1.0)) {
                object_post((t_object *)x, "Setting initial threshold to %f.",
                threshold);
                x->fine_boundary = threshold;
            } else {
                object_post((t_object *)x, "Argument 5 must be a float between
                0.0 and 1.0, inclusive, specifying the threshold for phrase end
                detection. Setting to default: %f.", 0.75);
                x->fine_boundary = 0.75;
            }
        } else {
            object_post((t_object *)x, "Argument 5 must be a float between 0.0
                and 1.0, inclusive, specifying the threshold for phrase end
                detection. Setting to default: %f.", 0.75);
            x->fine_boundary = 0.75;
        }

        // To do: add a warning for more than 5 arguments.

    }

    return (x);

}

////////////////////////////////

void phraseDetect_assist(t_phraseDetect *x, void *b, long m, long a, char *s)
{

    if (m == ASSIST_INLET) { // inlet
        sprintf(s, "Inlet %ld, durations list.", a);
    }
    else {  // outlet
        sprintf(s, "I am outlet %ld", a);
    }
```

```
}

////////////////////////////////

void phraseDetect_free(t_phraseDetect *x)
{

    //object_free(x->m_clock);
    phraseDetect_clearNonEmptyInputBuffers(x);
    long i;
    for (i = 0; i < x->d_phrase_count; i++) {
        /*
        free(x->phrases_d[i].frqs);
        free(x->phrases_d[i].amps);
        free(x->phrases_d[i].durs);
        */
        sysmem_freeptr(x->phrases_d[i].frqs);
        sysmem_freeptr(x->phrases_d[i].amps);
        sysmem_freeptr(x->phrases_d[i].durs);
    }
    for (i = 0; i < x->drd_phrase_count; i++) {
        /*
        free(x->phrases_drd[i].frqs);
        free(x->phrases_drd[i].amps);
        free(x->phrases_drd[i].durs);
        */
        sysmem_freeptr(x->phrases_drd[i].frqs);
        sysmem_freeptr(x->phrases_drd[i].amps);
        sysmem_freeptr(x->phrases_drd[i].durs);
    }
    /*
    free(x->phrases_d);
    free(x->phrases_drd);
    */
    sysmem_freeptr(x->phrases_d);
    sysmem_freeptr(x->phrases_drd);

    // Proxies and clocks have to be freed using freeobject.
    freeobject((t_object *)x->m_proxy);
    freeobject((t_object *)x->n_proxy);

}

////////////////////////////////

/*
void floatToLongArray(float *fArray, long *lArray, long array_size)
{

    long i;
```

```
    for (i = 0; i < array_size; i ++) {
        lArray[i] = (long)fArray[i];
    }

}
*/

////////////////////////////////

void phraseDetect_output(t_phraseDetect *x)
{

    if (x->drd_phrase_count > 0) {
        // Spit out the first drd t_phrase.
        //
        critical_enter(0);
        //
        x->drd_phrase_index = (x->drd_phrase_index + 1);
        long l = x->phrases_drd[0].length;
        //post("l drd =================== %ld", l);
        t_atom dargv[l];
        atom_setlong_array(l,dargv,l,x->phrases_drd[0].durs);
        t_atom aargv[l];
        atom_setdouble_array(l,aargv,l,x->phrases_drd[0].amps);
        t_atom fargv[l];
        atom_setdouble_array(l,fargv,l,x->phrases_drd[0].frqs);
        //
        critical_exit(0);
        //
        outlet_list(x->m_outlet1,NULL,l,dargv);
        outlet_list(x->m_outlet2,NULL,l,aargv);
        outlet_list(x->m_outlet3,NULL,l,fargv);
        outlet_int(x->m_outlet4, x->drd_phrase_index);

        // Shift all of the remaining instances of t_phrase up one slot in the
        // array of phrases.
        long i;
        long nl;
        //
        critical_enter(0);
        //
        for (i = 1; i < x->drd_phrase_count; i++) {
            nl = x->phrases_drd[i].length;
            //post("nl drd =================== %ld", nl);
            /*
            free(x->phrases_drd[(i - 1)].frqs);
            free(x->phrases_drd[(i - 1)].amps);
            free(x->phrases_drd[(i - 1)].durs);
            */
            sysmem_freeptr(x->phrases_drd[(i - 1)].frqs);
            sysmem_freeptr(x->phrases_drd[(i - 1)].amps);
```

```
            sysmem_freeptr(x->phrases_drd[(i - 1)].durs);
            /*
            x->phrases_drd[(i - 1)].frqs = calloc(nl,sizeof(float));
            x->phrases_drd[(i - 1)].amps = calloc(nl,sizeof(float));
            x->phrases_drd[(i - 1)].durs = calloc(nl,sizeof(long));
            */
            x->phrases_drd[(i - 1)].frqs =
                (double*)sysmem_newptrclear(nl * sizeof(double));
            x->phrases_drd[(i - 1)].amps =
                (double*)sysmem_newptrclear(nl * sizeof(double));
            x->phrases_drd[(i - 1)].durs =
                (long*)sysmem_newptrclear(nl * sizeof(long));
            long e;
            for (e = 0; e < nl; e++) {
                x->phrases_drd[(i - 1)].frqs[e] = x->phrases_drd[i].frqs[e];
                x->phrases_drd[(i - 1)].amps[e] = x->phrases_drd[i].amps[e];
                x->phrases_drd[(i - 1)].durs[e] = x->phrases_drd[i].durs[e];
                x->phrases_drd[(i - 1)].length = nl;
            }
        }
        // Free the last t_phrase.
        /*
        free(x->phrases_drd[(x->drd_phrase_count - 1)].frqs);
        free(x->phrases_drd[(x->drd_phrase_count - 1)].amps);
        free(x->phrases_drd[(x->drd_phrase_count - 1)].durs);
        */
        sysmem_freeptr(x->phrases_drd[(x->drd_phrase_count - 1)].frqs);
        sysmem_freeptr(x->phrases_drd[(x->drd_phrase_count - 1)].amps);
        sysmem_freeptr(x->phrases_drd[(x->drd_phrase_count - 1)].durs);
        x->drd_phrase_count = (x->drd_phrase_count - 1);
        //
        critical_exit(0);
        //
    }
    if (x->d_phrase_count > 0) {
        // Spit out the first d t_phrase.
        //
        critical_enter(0);
        //
        x->d_phrase_index = (x->d_phrase_index + 1);
        long l = x->phrases_d[0].length;
        //post("l d =================== %ld", l);
        t_atom dargv[l];
        atom_setlong_array(l,dargv,l,x->phrases_d[0].durs);
        t_atom aargv[l];
        atom_setdouble_array(l,aargv,l,x->phrases_d[0].amps);
        t_atom fargv[l];
        atom_setdouble_array(l,fargv,l,x->phrases_d[0].frqs);
        //
        critical_exit(0);
        //
```

```
outlet_list(x->m_outlet5,NULL,l,dargv);
outlet_list(x->m_outlet6,NULL,l,aargv);
outlet_list(x->m_outlet7,NULL,l,fargv);
outlet_int(x->m_outlet8, x->d_phrase_index);

// Shift all of the remaining instances of t_phrase up one slot in the
// array of phrases.
long i;
long nl;
//
critical_enter(0);
//
for (i = 1; i < x->d_phrase_count; i++) {
    nl = x->phrases_d[i].length;
    //post("nl d =================== %ld", nl);
    /*
    free(x->phrases_d[(i - 1)].frqs);
    free(x->phrases_d[(i - 1)].amps);
    free(x->phrases_d[(i - 1)].durs);
    */
    sysmem_freeptr(x->phrases_d[(i - 1)].frqs);
    sysmem_freeptr(x->phrases_d[(i - 1)].amps);
    sysmem_freeptr(x->phrases_d[(i - 1)].durs);
    /*
    x->phrases_d[(i - 1)].frqs = calloc(nl,sizeof(float));
    x->phrases_d[(i - 1)].amps = calloc(nl,sizeof(float));
    x->phrases_d[(i - 1)].durs = calloc(nl,sizeof(long));
    */
    x->phrases_d[(i - 1)].frqs =
        (double*)sysmem_newptrclear(nl * sizeof(double));
    x->phrases_d[(i - 1)].amps =
        (double*)sysmem_newptrclear(nl * sizeof(double));
    x->phrases_d[(i - 1)].durs =
        (long*)sysmem_newptrclear(nl * sizeof(long));
    long e;
    for (e = 0; e < nl; e++) {
        x->phrases_d[(i - 1)].frqs[e] = x->phrases_d[i].frqs[e];
        x->phrases_d[(i - 1)].amps[e] = x->phrases_d[i].amps[e];
        x->phrases_d[(i - 1)].durs[e] = x->phrases_d[i].durs[e];
        x->phrases_d[(i - 1)].length = nl;
    }
}
// Free the last t_phrase.
/*
free(x->phrases_d[(x->d_phrase_count - 1)].frqs);
free(x->phrases_d[(x->d_phrase_count - 1)].amps);
free(x->phrases_d[(x->d_phrase_count - 1)].durs);
*/
sysmem_freeptr(x->phrases_d[(x->d_phrase_count - 1)].frqs);
sysmem_freeptr(x->phrases_d[(x->d_phrase_count - 1)].amps);
sysmem_freeptr(x->phrases_d[(x->d_phrase_count - 1)].durs);
```

```
        x->d_phrase_count = (x->d_phrase_count - 1);
        //
        critical_exit(0);
        //
    }
    /*
    if ((x->drd_phrase_count == 0) && (x->d_phrase_count == 0)) {
        clock_unset(x->m_clock);
    } else {
        //double time;
        //clock_getftime(&time);
        //post("instance %lx is executing at time %.2f", x, time);
        clock_fdelay(x->m_clock, 0.1);
    }
    */
    if ((x->drd_phrase_count > 0) || (x->d_phrase_count > 0)) {
        phraseDetect_output(x);
    }

}

////////////////////////////////

void phraseDetect_ft1(t_phraseDetect *x, double value)
{
    post("threshold before: %f", value);
    if (value > 1) {
        x->fine_boundary = 1.0;
    } else if (value < 0) {
        x->fine_boundary = 0;
    } else {
        x->fine_boundary = value;
    }
    post("threshold after: %f", x->fine_boundary);
    /*
    post("News threshold before: %f", value);
    if ((long)value == 1) {
        x->fine_boundary = value;
    } else {
        x->fine_boundary = (fmod(fabs(value), 1));
    }
    post("News threshold after: %f", x->fine_boundary);
    //outlet_float(x->m_outlet2, x->fine_boundary);
     */

}

////////////////////////////////

long aMemberOfB(long l, long arr[], long arrsize)
{
```

```
    long i;
    long r = 0;
    for (i = 0; i < arrsize; i++) {
        if (l == arr[i]) {
            r = 1;
            break;
        }
    }
    return r;

}

////////////////////////////////

/*
long indexOfMax(long arr[], long arrsize)
{

    long value = 0;
    long index = 0;
    long i;
    for (i = 0; i < arrsize; i++) {
        if (arr[i] > value) {
            value = arr[i];
            index = i;
        }
    }
    return index;

}
*/

////////////////////////////////

void bubbleSort(long a[], long array_size)
{

     long i, j, temp;
     for (i = 0; i < (array_size - 1); ++i)
     {
          for (j = 0; j < array_size - 1 - i; ++j )
          {
               if (a[j] > a[j+1])
               {
                    temp = a[j+1];
                    a[j+1] = a[j];
                    a[j] = temp;
               }
          }
     }
```

```
}

////////////////////////////////

void bubbleSortf(double a[], long array_size)
{

     long i, j;
     double temp;
     for (i = 0; i < (array_size - 1); ++i)
     {
          for (j = 0; j < array_size - 1 - i; ++j )
          {
               if (a[j] > a[j+1])
               {
                    temp = a[j+1];
                    a[j+1] = a[j];
                    a[j] = temp;
               }
          }
     }

}

////////////////////////////////

double phraseDetect_getDursThreshold(t_phraseDetect *x)
{

    long i;
    long sortedDurations[x->durs_length];
    for (i = 0; i < x->durs_length; i++) {
        sortedDurations[i] = x->durs[i];
    }
    bubbleSort(sortedDurations, x->durs_length);

    // Check bubbleSort:
    /*
    for (i = 0; i < x->durs_length; i++) {
        post("========= sortedDurations[%ld] = %ld", i, sortedDurations[i]);
    }
    */

    double lowerBound;

    // even example: size 4 / 2 = 2,     2 - 0.5 = middle "index" 1.5:
    // | | ^ | |
    //  odd example: size 5 / 2 = 2.5, 2.5 - 0.5 = middle  index  2:
    // | | >|< | |
    // You'll never have a size of 0 because window defaults to at least the
```

```
    // smallest settable length of a phrase: 3.

    // This is the index of the MEDIAN value.
    double indexOfLowerBound = (((double)x->durs_length * 0.5) - 0.5);
    //post("========= indexOfLowerBound = %f", indexOfLowerBound);
    double c;
    double f;
    long lc;
    long lf;
    if ((fmod(indexOfLowerBound, 1.0)) == 0.0) {
        // if indexOfLowerBound is a whole number
        //post("========= condition 1 (whole number index);
        long l;
        l = (long)indexOfLowerBound;
        lowerBound = (double)sortedDurations[l];
    } else {
        // if indexOfLowerBound has a fractional part
        //post("========= condition 2 (fractional index)");
        c = indexOfLowerBound+1.0;
        f = indexOfLowerBound;
        lc = (long)c;
        lf = (long)f;
        lowerBound = ((double)(sortedDurations[lf]+sortedDurations[lc]) / 2.0);
    }
    // For the maximum duration, pick the smallest of the largest duration in
    // the window or the maximum allowable duration.
    double upperBound;
    if (sortedDurations[(x->durs_length - 1)] < x->maxDurLength) {
        upperBound = (double)sortedDurations[(x->durs_length - 1)];
    } else {
        upperBound = (double)x->maxDurLength;
    }
    double threshold =
    (((upperBound - lowerBound) * x->fine_boundary) + lowerBound);
    //post("Durations lowerBound = %f", lowerBound);
    //post("Durations upperBound = %f", upperBound);
    //post("Durations threshold == %f", threshold);

    return threshold;

}

////////////////////////////////

double phraseDetect_getFrqsThreshold(t_phraseDetect *x)
{

    long numberOfDifferences = (x->frqs_length - 1);
    double differences[numberOfDifferences];
    long i;
    long a;
```

```
for (i = 1, a = 0; i < x->frqs_length; i++, a++) {
    differences[a] = (x->frqs[i] - x->frqs[i - 1]);
}

double sortedDifferences[numberOfDifferences];
for (i = 0; i < numberOfDifferences; i++) {
    sortedDifferences[i] = fabs(differences[i]);
}
bubbleSortf(sortedDifferences, numberOfDifferences);

double lowerBound;

// even example: size 4 / 2 = 2,     2 - 0.5 = middle "index" 1.5: | | ^ | |
//  odd example: size 5 / 2 = 2.5, 2.5 - 0.5 = middle  index  2:
// | | >|< | |
//  You'll never have a size of 0 because window defaults to at least the
// smallest settable length of a phrase: 3.

// This is the index of the MEDIAN value.
double indexOfLowerBound = (((double)numberOfDifferences * 0.5) - 0.5);
double c;
double f;
long lc;
long lf;
if ((fmod(indexOfLowerBound, 1.0)) == 0) {
    // if indexOfLowerBound is a whole number
    long l;
    l = (long)indexOfLowerBound;
    lowerBound = sortedDifferences[l];
} else {
    // if indexOfLowerBound has a fractional part
    c = indexOfLowerBound+1.0;
    f = indexOfLowerBound;
    lc = (long)c;
    lf = (long)f;
    lowerBound = ((sortedDifferences[lf] + sortedDifferences[lc])/2.0);
}

double upperBound = sortedDifferences[(numberOfDifferences - 1)];
double threshold =
    (((upperBound - lowerBound) * x->fine_boundary) + lowerBound);
//post("Pitch height difference lowerBound = %.2f",lowerBound);
//post("Pitch height difference upperBound = %.2f",upperBound);
//post("Pitch height diffeernce threshold ==  %.2f",threshold);

// Set x->register_differences = differences.
x->register_differences =
    (double*)sysmem_newptrclear(numberOfDifferences * sizeof(double));
for (i = 0; i < numberOfDifferences; i++) {
    x->register_differences[i] = differences[i];
}
```

```
    return threshold;

}

////////////////////////////////

void phraseDetect_clearNonEmptyInputBuffers(t_phraseDetect *x)
{

    critical_enter(0);

    // You could free a pointer twice unless you make sure the size is > 0 for
    // each list before attempting to clear.
    if (x->frqs_length > 0) {
        //free(x->frqs);
        sysmem_freeptr(x->frqs);
        x->frqs_length = 0;
    }
    if (x->amps_length > 0) {
        //free(x->amps);
        sysmem_freeptr(x->amps);
        x->amps_length = 0;
    }
    if (x->durs > 0) {
        //free(x->durs);
        sysmem_freeptr(x->durs);
        x->durs_length = 0;
    }

    critical_exit(0);

}

////////////////////////////////

long phraseDetect_badFrqsNumber(t_phraseDetect *x, double f)
{

    long badArg = 0;
    if ((f < 0.0) || (f > 127.0)) {
        badArg = 1;
        /*
        object_error((t_object *)x, "Bad argument in input to the leftomost
            inlet (MIDI key numbers). Input to this inlet can only be a
            float or an int or a list of floats and/or ints between 0.0 and
            127.0, inclusive.");
        */
    }
    return badArg;
```

```
}

////////////////////////////////

long phraseDetect_badAmpsNumber(t_phraseDetect *x, double f)
{

    long badArg = 0;
    if ((f < 0.0) || (f > 1.0)) {
        badArg = 1;
        /*
        object_error((t_object *)x, "Bad argument in input to the second inlet
            from left (amplitude scalars). Input to this inlet can only be a
            float or a list of floats between 0.0 and 1.0, inclusive.");
        */
    }
    return badArg;

}

////////////////////////////////

long phraseDetect_badDursNumber(t_phraseDetect *x, long l)
{

    long badArg = 0;
    if (l < 0) {
        badArg = 1;
        /*
        object_error((t_object *)x, "Bad argument in input to the third inlet
        from left (durations). Input to this inlet can only be a positive
        integer or a list of integers representing durations in milliseconds.");
        */
    }
    return badArg;

}

////////////////////////////////

long phraseDetect_badFrqsArgList(t_phraseDetect *x, long argc, t_atom *argv)
{

    long badArgList = 0;
    long i;
    for (i = 0; i < argc; i++) {
        if (((argv + i)->a_type != A_LONG) && ((argv + i)->a_type != A_FLOAT)) {
            /*
            object_error((t_object *)x, "Bad argument in input to the leftomost
                inlet (MIDI key numbers). Input to this inlet can only be a
                float or an int or a list of floats and/or ints between 0.0 and
```

```
                127.0, inclusive.");
            */
            badArgList = 1;
            break;
        } else if ((argv + i)->a_type == A_FLOAT) {
            double f = atom_getfloat(argv + i);
            if ((f < 0.0) || (f > 127.0)) {
                badArgList = 1;
                /*
                object_error((t_object *)x, "Bad argument in input to the
                    leftomost inlet (MIDI key numbers). Input to this inlet can
                    only be a float or an int or a list of floats and/or ints
                    between 0.0 and 127.0, inclusive.");
                */
                break;
            }
        } else if ((argv + i)->a_type == A_LONG) {
            long l = atom_getlong(argv + i);
            if ((l < 0) || (l > 127)) {
                badArgList = 1;
                /*
                object_error((t_object *)x, "Bad argument in input to the
                    leftomost inlet (MIDI key numbers). Input to this inlet can
                    only be a float or an int or a list of floats and/or ints
                    between 0.0 and 127.0, inclusive.");
                */
                break;
            }
        }
    }
    return badArgList;

}

////////////////////////////////

long phraseDetect_badAmpsArgList(t_phraseDetect *x, long argc, t_atom *argv)
{

    long badArgList = 0;
    long i;
    for (i = 0; i < argc; i++) {
        if ((argv + i)->a_type != A_FLOAT) {
            badArgList = 1;
            /*
            object_error((t_object *)x, "Bad argument in input to the second
                inlet from left (amplitude scalars). Input to this inlet can
                only be a float or a list of floats between 0.0 and 1.0,
                inclusive.");
            */
            break;
```

```
        } else {
            double f = atom_getfloat(argv + i);
            if ((f < 0.0) || (f > 1.0)) {
                badArgList = 1;
                /*
                object_error((t_object *)x, "Bad argument in input to the second
                    inlet from left (amplitude scalars). Input to this inlet can
                    only be a float or a list of floats between 0.0 and 1.0,
                    inclusive.");
                */
                break;
            }
        }
    }
    return badArgList;

}

////////////////////////////////

long phraseDetect_badDursArgList(t_phraseDetect *x, long argc, t_atom *argv)
{

    long i;
    long badArgList = 0;
    for (i = 0; i < argc; i++) {
        if ((argv + i)->a_type != A_LONG) {
            badArgList = 1;
            /*
            object_error((t_object *)x, "Bad argument in input to the third
                inlet from left (durations). Input to this inlet can only be
                a positive integer or a list of integers representing durations
                in milliseconds.");
            */
            break;
        } else {
            long l = atom_getlong(argv + i);
            if (l < 0) {
                badArgList = 1;
                /*
                object_error((t_object *)x, "Bad argument in input to the third
                    inlet from left (durations). Input to this inlet can only be
                    a positive integer or a list of integers representing
                    durations in milliseconds.");
                */
                break;
            }
        }
    }
    return badArgList;
```

```
}

///////////////////////////////

// values must be cast to float on input
void phraseDetect_frqsNumberIn(t_phraseDetect *x, double f)
{

    if ((phraseDetect_badFrqsNumber(x, f) == 0) &&
       ((x->frqs_length + 1) == x->amps_length))  {

        //post("frqs input list added to by FLOAT");

        critical_enter(0);

        if (x->frqs_length == 0) {
            x->frqs = (double*)sysmem_newptrclear(sizeof(double));
        } else {
            x->frqs = (double*)sysmem_resizeptr(x->frqs, ((x->frqs_length + 1) *
                sizeof(double)));
        }

        x->frqs[x->frqs_length] = f;
        x->frqs_length = (x->frqs_length + 1);

        critical_exit(0);

        // Once we reach or exceed analysis window size, find the phrases; save
        // the index of the end of the last phrase and move all the notes after
        // this index to the head of the input buffers.
        if (x->frqs_length >= x->analysis_window) {
            phraseDetect_crunch(x);
        }

    } else {

        /*
        object_error((t_object *)x, "Bad argument or bad input size. Clearing
            all input buffers.");
        */
        phraseDetect_clearNonEmptyInputBuffers(x);

    }

}

///////////////////////////////

// values must be cast to float on input
void phraseDetect_ampsNumberIn(t_phraseDetect *x, double f)
{
```

```
    if ((phraseDetect_badAmpsNumber(x, f) == 0) && ((x->amps_length + 1) ==
        x->durs_length))  {

        //post("amps input list added to by FLOAT");

        critical_enter(0);

        if (x->amps_length == 0) {
            x->amps = (double*)sysmem_newptrclear(sizeof(double));
        } else {
            x->amps = (double*)sysmem_resizeptr(x->amps, ((x->amps_length + 1) *
                sizeof(double)));
        }

        x->amps[x->amps_length] = f;
        x->amps_length = (x->amps_length + 1);

        critical_exit(0);

    } else {

        /*
        object_error((t_object *)x, "Bad argument or bad input size. Clearing
            all input buffers.");
        */
        phraseDetect_clearNonEmptyInputBuffers(x);

    }

}

////////////////////////////////

// values must be cast to float on input
void phraseDetect_dursNumberIn(t_phraseDetect *x, long l)
{

    if ((phraseDetect_badDursNumber(x, l) == 0))  {

        //post("durs input list added to by FLOAT");

        critical_enter(0);

        if (x->durs_length == 0) {
            x->durs = (long*)sysmem_newptrclear(sizeof(long));
        } else {
            x->durs = (long*)sysmem_resizeptr(x->durs, ((x->durs_length + 1) *
                sizeof(long)));
        }
```

```
        x->durs[x->durs_length] = 1;
        x->durs_length = (x->durs_length + 1);

        critical_exit(0);

    } else {

        /*
        object_error((t_object *)x, "Bad argument or bad input size. Clearing
            all input buffers.");
        */
        phraseDetect_clearNonEmptyInputBuffers(x);

    }

}

////////////////////////////////

void phraseDetect_frqsListIn(t_phraseDetect *x, long argc, t_atom *argv)
{

    if ((phraseDetect_badFrqsArgList(x, argc, argv) == 0) &&
       ((x->frqs_length + argc) == x->amps_length) && (x->amps_length > 0))  {

        //post("frqs input list added to by LIST");

        critical_enter(0);

        if (x->frqs_length == 0) {
            x->frqs = (double*)sysmem_newptrclear(argc * sizeof(double));
        } else {
            x->frqs = (double*)sysmem_resizeptr(x->frqs,
                ((x->frqs_length + argc) * sizeof(double)));
        }

        long i;
        t_atom *ap;
        for (i = x->frqs_length, ap = argv; i < (x->frqs_length + argc); i++,
            ap++) {
            if ((ap)->a_type == A_FLOAT) {
                x->frqs[i] = atom_getfloat(ap);
            } else {
                x->frqs[i] = (double)atom_getlong(ap);
            }
        }

        x->frqs_length = (x->frqs_length + argc);

        critical_exit(0);
```

```
        // Once we reach or exceed analysis window size, find the phrases; save
        // the index of the end of the last phrase and move all the information
        // for notes occuring after this index to the head of the input buffers.
        if (x->frqs_length >= x->analysis_window) {
            phraseDetect_crunch(x);
        }

    } else {

        /*
        object_error((t_object *)x, "Bad argument or bad input size. Clearing
            all input buffers.");
        */
        phraseDetect_clearNonEmptyInputBuffers(x);

        //post("In frqs func: x->frqs length is %ld", x->frqs_length);
        //post("In frqs func: x->amps length is %ld", x->amps_length);
        //post("In frqs func: x->durs length is %ld", x->durs_length);

    }

}

////////////////////////////////

void phraseDetect_ampsListIn(t_phraseDetect *x, long argc, t_atom *argv)
{

    if ((phraseDetect_badAmpsArgList(x, argc, argv) == 0) &&
       ((x->amps_length + argc) == x->durs_length) && (x->durs_length > 0)) {

        //post("amps input list added to");

        critical_enter(0);

        if (x->amps_length == 0) {
            x->amps = (double*)sysmem_newptrclear(argc * sizeof(double));
        } else {
            x->amps = (double*)sysmem_resizeptr(x->amps,
                ((x->amps_length + argc) * sizeof(double)));
        }

        long i;
        t_atom *ap;
        for (i = x->amps_length, ap = argv; i < (x->amps_length + argc); i++,
            ap++) {
            x->amps[i] = atom_getfloat(ap);
        }

        x->amps_length = (x->amps_length + argc);
```

```
        critical_exit(0);

    } else {

        /*
        object_error((t_object *)x, "Bad argument or bad input size. Clearing
            all input buffers.");
        */
        phraseDetect_clearNonEmptyInputBuffers(x);

        //post("In amps func: x->frqs length is %ld", x->frqs_length);
        //post("In amps func: x->amps length is %ld", x->amps_length);
        //post("In amps func: x->durs length is %ld", x->durs_length);

    }

}

////////////////////////////////

void phraseDetect_dursListIn(t_phraseDetect *x, long argc, t_atom *argv)
{

    if ((phraseDetect_badDursArgList(x, argc, argv) == 0) &&
        (argc <= x->analysis_window)) {

        //post("durs input list added to");

        critical_enter(0);

        if (x->durs_length == 0) {
            x->durs = (long*)sysmem_newptrclear(argc * sizeof(long));
        } else {
            x->durs = (long*)sysmem_resizeptr(x->durs,
                ((x->durs_length + argc) * sizeof(long)));
        }

        /*
        post("Argv + 0 ================================== %ld",
            atom_getlong(argv));
        post("Argv + 1 ================================== %ld",
            atom_getlong(argv+1));
        post("Argv + 2 ================================== %ld",
            atom_getlong(argv+2));
        */

        long i;
        t_atom *ap;
        for (i = x->durs_length, ap = argv; i < (x->durs_length + argc); i++,
            ap++) {
            x->durs[i] = atom_getlong(ap);
```

```
            // Check:
            //post("Durs arglist elt ================================= %ld",
            //atom_getlong(ap));
        }

        x->durs_length = (x->durs_length + argc);

        critical_exit(0);

    } else {
        /*
        object_error((t_object *)x, "Bad argument or bad input size. Clearing
            all input buffers.");
        */
        phraseDetect_clearNonEmptyInputBuffers(x);

        //post("In durs func: x->frqs length is %ld", x->frqs_length);
        //post("In durs func: x->amps length is %ld", x->amps_length);
        //post("In durs func: x->durs length is %ld", x->durs_length);

    }

}

////////////////////////////////

void phraseDetect_float(t_phraseDetect *x, double value)
{

    switch (proxy_getinlet((t_object *)x)) {
        case 0:
            // FREQUENCY
            //post("frq float received in leftmost inlet");
            phraseDetect_frqsNumberIn(x, value);
            break;
        case 2:
            // DURATION
            //post("dur float received in third-from-right inlet");
            /*
            object_error((t_object *)x, "Bad argument in input to the third
                inlet from left (durations). Input to this inlet can only be
                a positive integer or a list of integers representing
                durations in milliseconds.");
            */
            /*
            object_error((t_object *)x, "Bad argument. Clearing all input
                buffers.");
            */
            phraseDetect_clearNonEmptyInputBuffers(x);
            break;
        case 3:
```

```
            // AMPLITUDE
            //post("amp float received in fourth-from-right inlet");
            phraseDetect_ampsNumberIn(x, value);
            break;
    }

}

////////////////////////////////

void phraseDetect_int(t_phraseDetect *x, long value)
{

    //post("the value = %ld", atom_getlong(argv));
    //long value = atom_getlong(argv);
    switch (proxy_getinlet((t_object *)x)) {
        case 0:
            // FREQUENCY
            //post("frq int received in leftmost inlet");
            phraseDetect_frqsNumberIn(x, (double)value);
            break;
        case 2:
            // DURATION
            //post("dur int received in third-from-right inlet");
            phraseDetect_dursNumberIn(x, value);
            break;
        case 3:
            // AMPLITUDE
            //post("amp int received in fourth-from-right inlet");
            /*
            object_error((t_object *)x, "Bad argument in input to the second
                inlet from left (amplitude scalars). Input to this inlet can
                only be a float or a list of floats between 0.0 and 1.0,
                inclusive.");
            object_error((t_object *)x, "Bad argument. Clearing all input
                buffers.");
            */
            phraseDetect_clearNonEmptyInputBuffers(x);
            break;
    }

}

////////////////////////////////

/*
void phraseDetect_list(t_phraseDetect *x, t_symbol *s, long argc, t_atom *argv)
{
    defer((t_object *)x, (method)phraseDetect_dolist, s, argc, argv);
}
*/
```

```
////////////////////////////////

void phraseDetect_list(t_phraseDetect *x, t_symbol *s, long argc, t_atom *argv)
{

    switch (proxy_getinlet((t_object *)x)) {
        case 0:
            //post("frqs list received in leftmost inlet");
            phraseDetect_frqsListIn(x, argc, argv);
            break;
        case 2:
            //post("durs list received in third-from-left inlet");
            phraseDetect_dursListIn(x, argc, argv);
            break;
        case 3:
            //post("amps list received in second-from-left inlet");
            phraseDetect_ampsListIn(x, argc, argv);
            break;
    }

}

////////////////////////////////

long phraseDetect_findMaxDur(t_phraseDetect *x, long startIndex, long endIndex)
{

    long d;
    long max = 0;
    for (d = startIndex; d <= endIndex; d++) {
        if (x->durs[d] > max) {
            max = x->durs[d];
        }
    }
    return max;

}

////////////////////////////////

void phraseDetect_crunch(t_phraseDetect *x)
{

    long i, index, j;

    critical_enter(0);

    // Check:
    /*
    for (i = 0; i < x->frqs_length; i++) {
```

```
    post("%ldth frq = %f", i, x->frqs[i]);
    post("%ldth amp = %f", i, x->amps[i]);
    post("%ldth dur = %ld", i, x->durs[i]);
}
*/

double threshold_durs = phraseDetect_getDursThreshold(x);
double threshold_frqs = phraseDetect_getFrqsThreshold(x);

// Collect boundary note indices (edges of phrases) based on durations.

long *leftBoundaryIndices;
long *rightBoundaryIndices;
//leftBoundaryIndices = calloc(1,sizeof(long));
leftBoundaryIndices = (long*)sysmem_newptrclear(sizeof(long));

// Set the first leftBoundaryIndex to the first note below the threshold
// duration.
for (i = 0; i < x->durs_length; i++) {
    if (x->durs[i] < threshold_durs) {
        leftBoundaryIndices[0] = i;
        break;
    }
}
long nextLbiIndex = 1;
long nextRbiIndex = 0;
long first_time = 1;
index = (leftBoundaryIndices[0] + 1);
while (index < x->durs_length) {
    if ((double)x->durs[index] >= threshold_durs) {
        if (first_time == 1) {
            rightBoundaryIndices = (long*)sysmem_newptrclear(sizeof(long));
            first_time = 0;
        } else {
            rightBoundaryIndices =
                (long*)sysmem_resizeptr(rightBoundaryIndices,
                    ((nextRbiIndex+1) * sizeof(long)));
        }
        rightBoundaryIndices[nextRbiIndex] = index;
        nextRbiIndex = nextRbiIndex+1;
        for (j = index+1; j < x->durs_length; j++) {
            if ((double)x->durs[j] < threshold_durs) {
                leftBoundaryIndices =
                    (long*)sysmem_resizeptr(leftBoundaryIndices,
                        ((nextLbiIndex+1) * sizeof(long)));
                leftBoundaryIndices[nextLbiIndex] = j;
                nextLbiIndex = nextLbiIndex+1;
                index = j; // index = j + 1;
                break;
            }
        }
```

```
    }
    //post("j=%ld", j);
    if (j == x->durs_length) {
        break;
    }
    index = (index + 1);
}

// Size of leftBoundaryIndices could be 1 greater than rightBoundaryIndices
// or equal to it -- if it is equal, ignore the remaining notes (don't shift
// them to beg of next buffer) because they are all over the threshold and
// we just use the first duration in any string of durations that are over
// the threshold to cap the current phrase. If leftBoundaryIndices is one
// greater, we shift that index to the final index to the beginning of the
// next analysis buffer
//post("nextLbiIndex (size of leftBoundaryIndices) = %ld", nextLbiIndex);
//post("nextRbiIndex (size of rightBoundaryIndices) = %ld", nextRbiIndex);

/*
for (i = 0; i < nextRbiIndex; i++) {
    post("====================== Durs lb,rb indices: %ld,%ld",
        leftBoundaryIndices[i],rightBoundaryIndices[i]);
    post("====================== Durs lb,rb pairs: %ld,%ld",
        x->durs[leftBoundaryIndices[i]],x->durs[rightBoundaryIndices[i]]);
}
post("====================== Last durs lbiIndex: %ld",
    leftBoundaryIndices[nextLbiIndex - 1]);
*/

// Collect boundary note indices (edges of phrases) based on drops in pitch
// height.

long *registerDropIndices;
long nextRdiIndex = 0;
long first_time_drop = 1;
for (i = 0; i < (x->frqs_length - 1); i++) {
    if ((x->register_differences[i]) <= (threshold_frqs * -1)) {
        if (first_time_drop == 1) {
            registerDropIndices = (long*)sysmem_newptrclear(sizeof(long));
            first_time_drop = 0;
        } else {
            registerDropIndices = (long*)sysmem_resizeptr(
                registerDropIndices, ((nextRdiIndex+1) * sizeof(long)));
        }
        registerDropIndices[nextRdiIndex] = i+1;
        nextRdiIndex = nextRdiIndex+1;
    }
}

/*
for (i = 0; i < (x->frqs_length - 1); i++) {
```

```
    post("x->register_differences: %.2f",x->register_differences[i]);
}
for (i = 0; i < nextRdiIndex; i++) {
    post("rdi indices, values: %ld,%.2f",registerDropIndices[i],
        x->frqs[registerDropIndices[i]]);
}
post("nextLbiIndex = %ld:",nextLbiIndex);
post("nextRbiIndex = %ld:",nextRbiIndex);
post("nextRdiIndex = %ld:",nextRdiIndex);
*/

// phrase_count is 0 when obj is initialized, but you must set it to 0
// again because a previous call to this func will have incremented it
x->drd_phrase_count = 0;
x->d_phrase_count = 0;

long width;
long d;
for (i = 0; i < nextRbiIndex; i++) {
    width = (rightBoundaryIndices[i] - leftBoundaryIndices[i]);
    // Do not add phrases that are shorter than minPhraseLength or longer
    // than maxPhraseLength, or phrases that contain a duration longer
    // than the maximum allowed duration.
    if ((width >= (x->minPhraseLength - 1)) &&
        (width <= (x->maxPhraseLength - 1)) &&
        (phraseDetect_findMaxDur(x, leftBoundaryIndices[i],
            rightBoundaryIndices[i]) < x->maxDurLength)) {
        // check if there's a register change at the right boundary in
        // registerDropIndices
        if (aMemberOfB(rightBoundaryIndices[i], registerDropIndices,
            nextRdiIndex) == 1) {
            x->phrases_drd[x->drd_phrase_count].frqs =
                (double*)sysmem_newptrclear((width+1) * sizeof(double));
            x->phrases_drd[x->drd_phrase_count].amps =
                (double*)sysmem_newptrclear((width+1) * sizeof(double));
            x->phrases_drd[x->drd_phrase_count].durs =
                (long*)sysmem_newptrclear((width+1) * sizeof(long));
            long c;
            for (c = 0, d = leftBoundaryIndices[i];
                d <= rightBoundaryIndices[i]; c++, d++) {
                x->phrases_drd[x->drd_phrase_count].frqs[c] = x->frqs[d];
                x->phrases_drd[x->drd_phrase_count].amps[c] = x->amps[d];
                x->phrases_drd[x->drd_phrase_count].durs[c] = x->durs[d];
                x->phrases_drd[x->drd_phrase_count].length = width+1;
            }
            x->drd_phrase_count = (x->drd_phrase_count + 1);
        } else {
            x->phrases_d[x->d_phrase_count].frqs =
                (double*)sysmem_newptrclear((width+1) * sizeof(double));
            x->phrases_d[x->d_phrase_count].amps =
                (double*)sysmem_newptrclear((width+1) * sizeof(double));
```

```
            x->phrases_d[x->d_phrase_count].durs =
                (long*)sysmem_newptrclear((width+1) * sizeof(long));
            long c;
            for (c = 0, d = leftBoundaryIndices[i];
                d <= rightBoundaryIndices[i]; c++, d++) {
                x->phrases_d[x->d_phrase_count].frqs[c] = x->frqs[d];
                x->phrases_d[x->d_phrase_count].amps[c] = x->amps[d];
                x->phrases_d[x->d_phrase_count].durs[c] = x->durs[d];
                x->phrases_d[x->d_phrase_count].length = width+1;
            }
            x->d_phrase_count = (x->d_phrase_count + 1);
        }
    }
}

/*
for (i = 0; i < x->durs_length; i++) {
    post("%ldth dur BEFORE TRANSFER = %ld", i, x->durs[i]);
}
*/

//post("drd_phrase_count: %ld", x->drd_phrase_count);
//post("d_phrase_count: %ld", x->d_phrase_count);

if (nextLbiIndex > nextRbiIndex) {
    // If there is one more left boundary index than there are right
    // boundary indices, grab all the data from the last left boundary index
    // to the end of x->durs and shift it to the beginning of the next
    // x->durs input buffer.
    long transferIndexBeg = leftBoundaryIndices[(nextLbiIndex - 1)];

    // Actually refers to the index just AFTER the last index we want to
    // transfer.
    long transferIndexEnd = x->frqs_length;

    long newLength = (transferIndexEnd - transferIndexBeg);
    //post("transferIndexBeg = %ld", transferIndexBeg);
    //post("transferIndexEnd = %ld", transferIndexEnd);
    //post("newLength = %ld", newLength);
    for (i = 0, d = transferIndexBeg; d < transferIndexEnd; i++, d++) {
        x->frqs[i] = x->frqs[d];
        x->amps[i] = x->amps[d];
        x->durs[i] = x->durs[d];
        x->frqs_length = newLength;
        x->amps_length = newLength;
        x->durs_length = newLength;
    }
} else {

    // If the pairs of left and right boundary indices are equal in size, we
    // tranfser nothing and overwrite the input buffers.
```

```
        x->frqs_length = 0;
        x->amps_length = 0;
        x->durs_length = 0;
    }

    // Should you be able to dynamically change analysis window size?  Not for
    // now -- just use multiple phraseDetect objects -- analysis_window should
    // ONLY be set via an initialization argument.

    /*
    for (i = 0; i < x->durs_length; i++) {
        post("%ldth dur AFTER TRANSFER = %ld", i, x->durs[i]);
    }
    */

    critical_exit(0);

    // phraseDetect_output kicks out phrases one by one -- we can't put
    // critical_exit(0) after this function call because the phraseDetect_output
    // calls outlet_ functions, which are already thread safe and will cause
    // crashes if wrapped in a critical area.
    phraseDetect_output(x);

}

////////////////////////////////

void phraseDetect_clear(t_phraseDetect *x) // IS THERE A MEMORY LEAK?
{

    //post("The _clear method was called.");

    phraseDetect_clearNonEmptyInputBuffers(x);
    long i;

    critical_enter(0);

    for (i = 0; i < x->d_phrase_count; i++) {
        /*
        free(x->phrases_d[i].frqs);
        free(x->phrases_d[i].amps);
        free(x->phrases_d[i].durs);
        */
        sysmem_freeptr(x->phrases_d[i].frqs);
        sysmem_freeptr(x->phrases_d[i].amps);
        sysmem_freeptr(x->phrases_d[i].durs);
    }
    for (i = 0; i < x->drd_phrase_count; i++) {
        /*
        free(x->phrases_drd[i].frqs);
        free(x->phrases_drd[i].amps);
```

```
        free(x->phrases_drd[i].durs);
        */
        sysmem_freeptr(x->phrases_drd[i].frqs);
        sysmem_freeptr(x->phrases_drd[i].amps);
        sysmem_freeptr(x->phrases_drd[i].durs);
    }
    /*
    free(x->phrases_d);
    free(x->phrases_drd);
    */
    sysmem_freeptr(x->phrases_d);
    sysmem_freeptr(x->phrases_drd);
    x->d_phrase_count = 0;
    x->drd_phrase_count = 0;
    x->phrases_d =
        (t_phrase*)sysmem_newptrclear(x->maxPhrasesLength * sizeof(t_phrase));
    x->phrases_drd =
        (t_phrase*)sysmem_newptrclear(x->maxPhrasesLength * sizeof(t_phrase));

    x->frqs_length = 0;
    x->amps_length = 0;
    x->durs_length = 0;

    x->d_phrase_index = 0;
    x->drd_phrase_index = 0;

    critical_exit(0);

}

//////////////////////////////// eof
```

B Max/MSP Patches

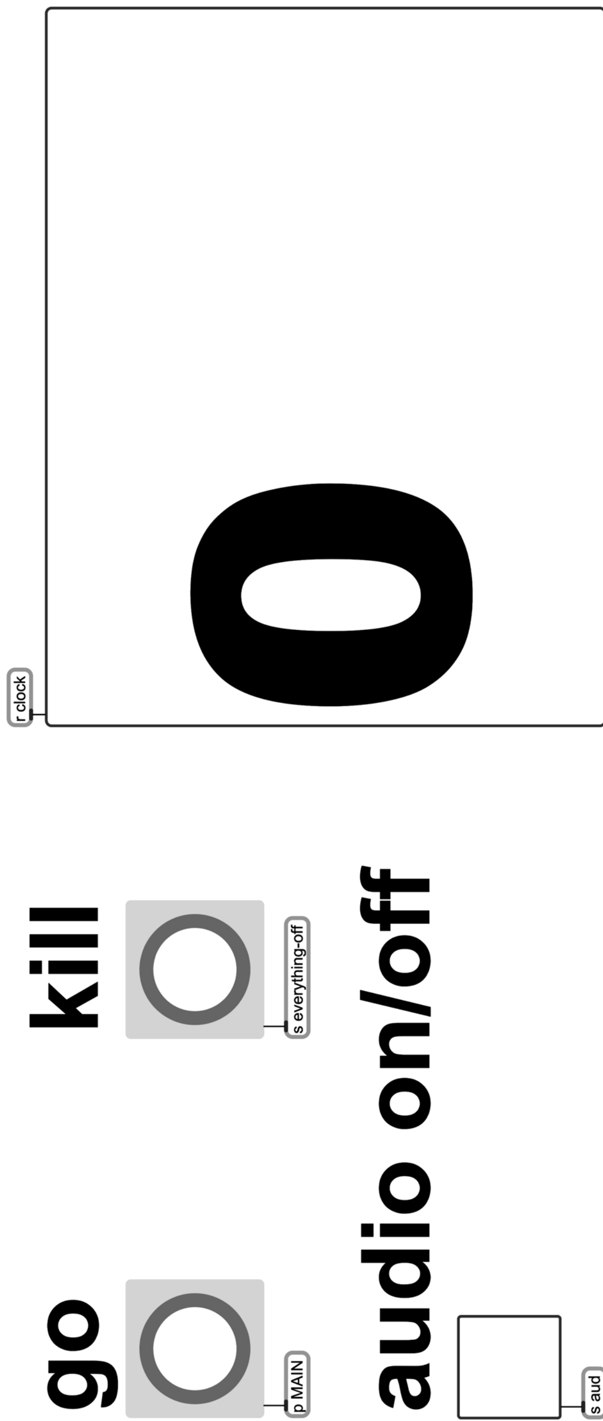

Figure B.1: Main control window.

Figure B.2: Main patch (p MAIN)

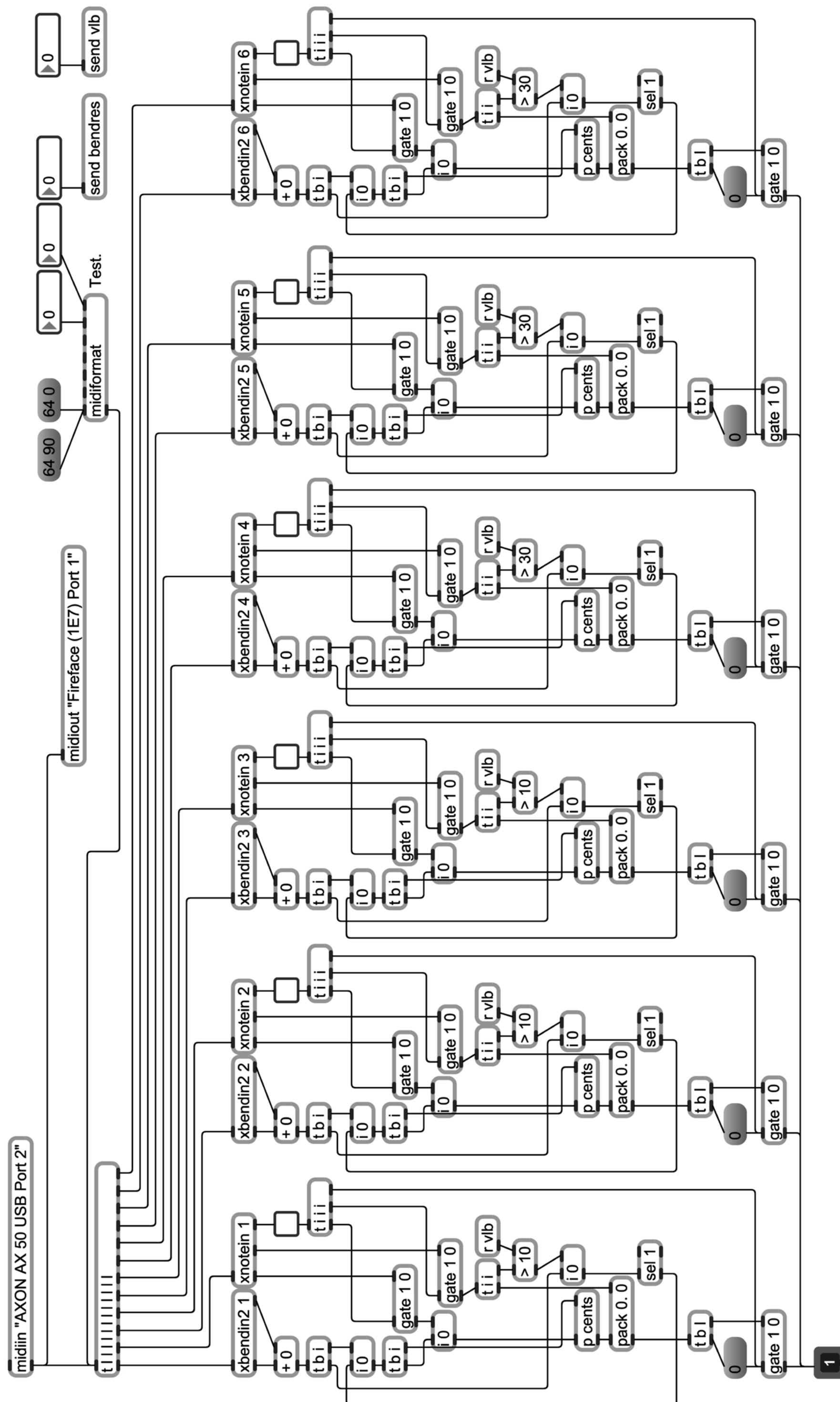

Figure B.3: Axon input filter (p axon_input)

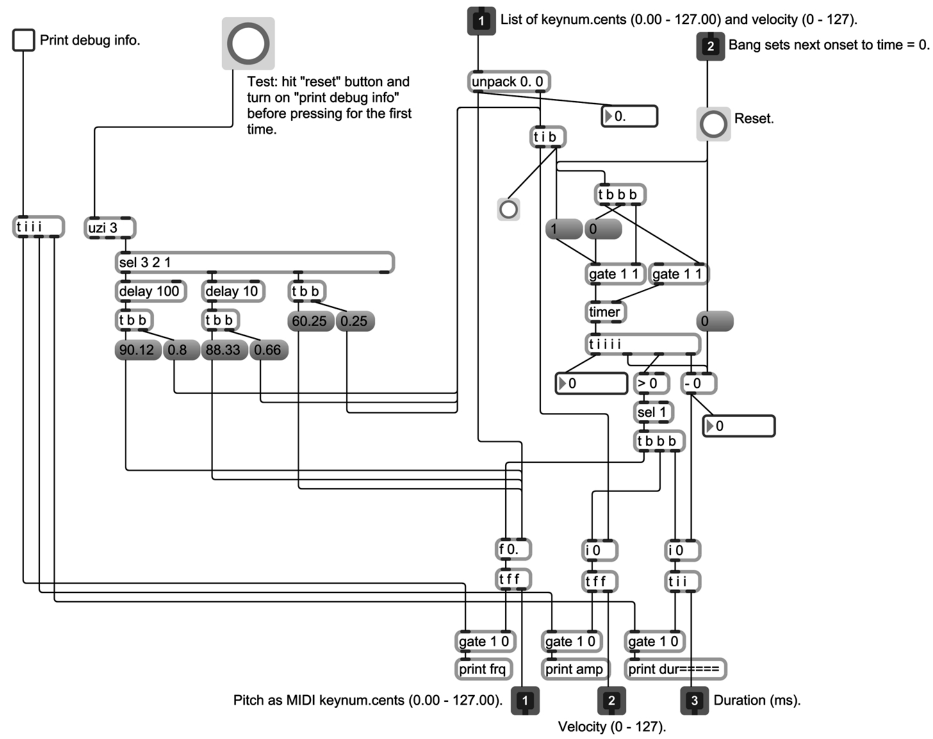

Figure B.4: Absolute time to time interval conversion (p onsets_to_durations)

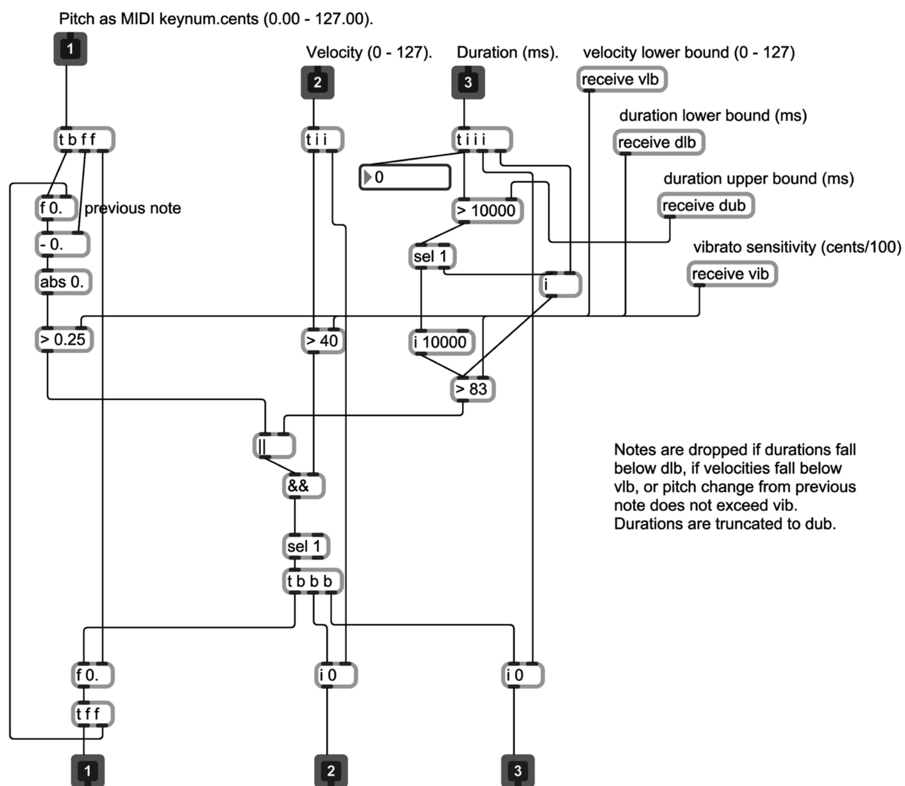

Figure B.5: MIDI filters (p filters)

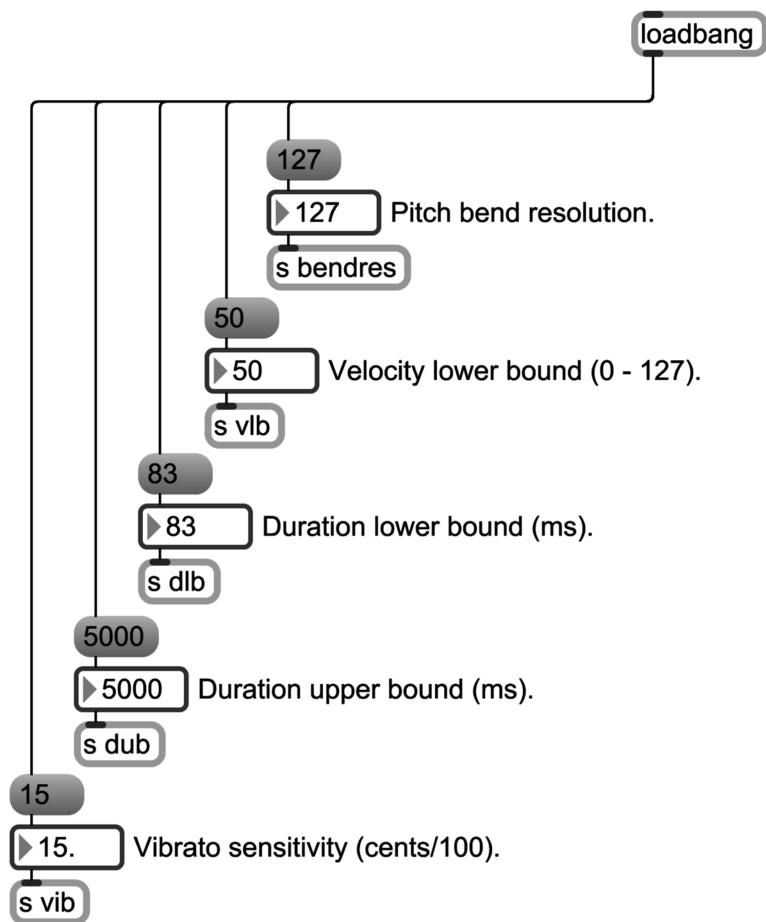

Figure B.6: Input settings for MIDI filters (p MIDI_input_settings)

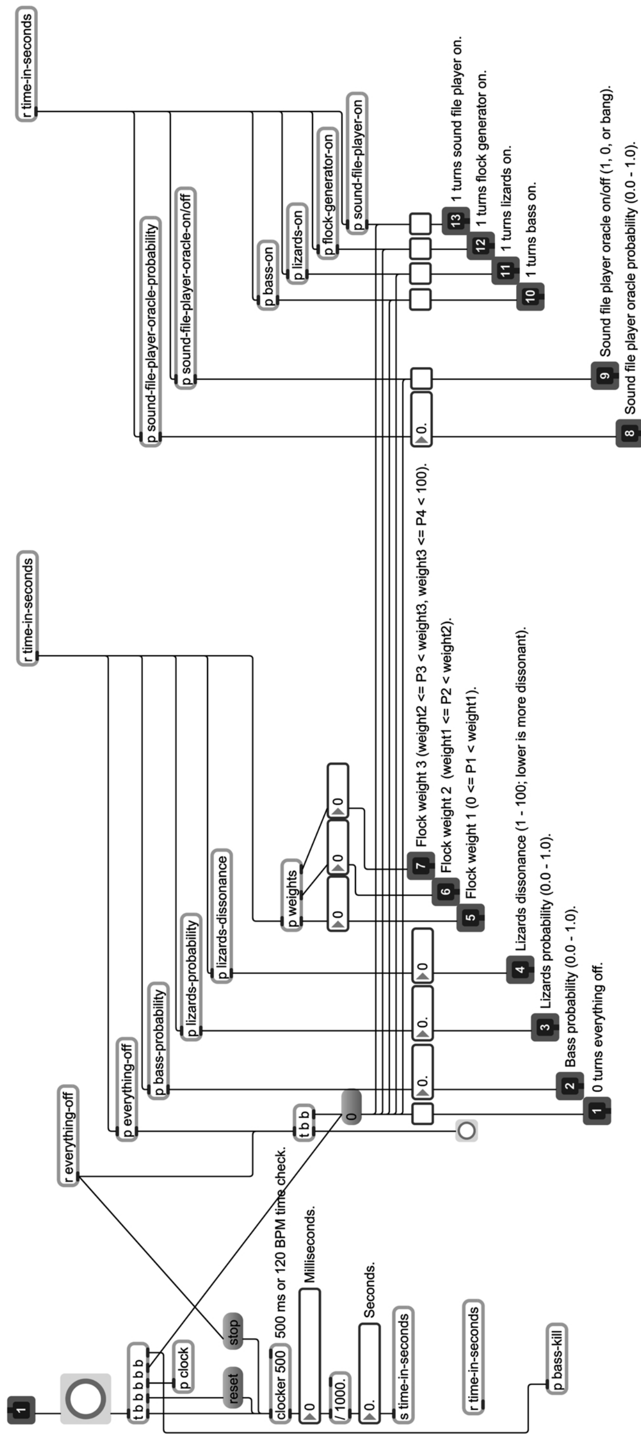

Figure B.7: Automated event scheduling (p score_playback)

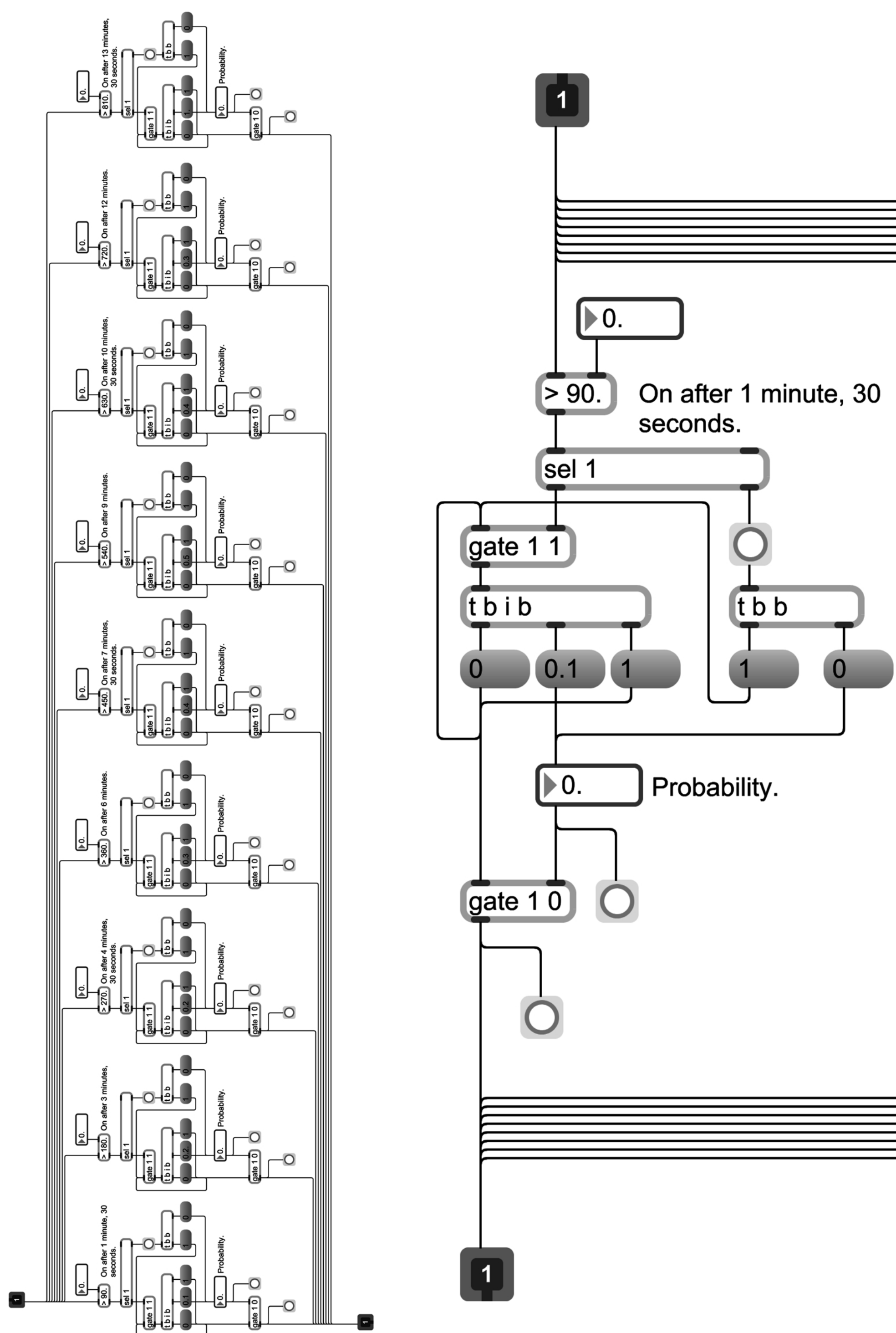

Figure B.8: Factor oracle probabilities for bass accompaniment (p bass-probability)

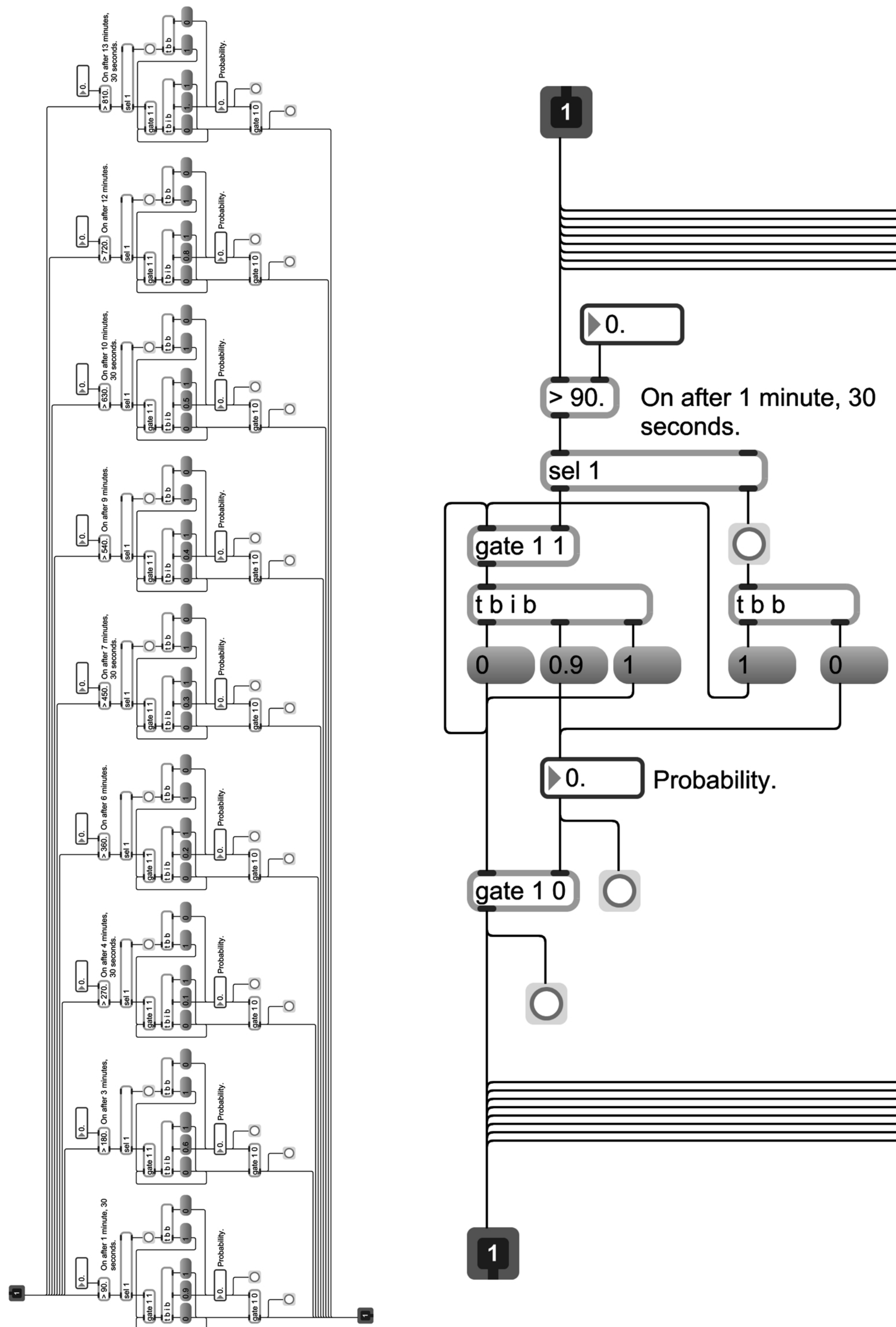

Figure B.9: Factor oracle probabilities for harmonic accompaniment (p lizards-probability)

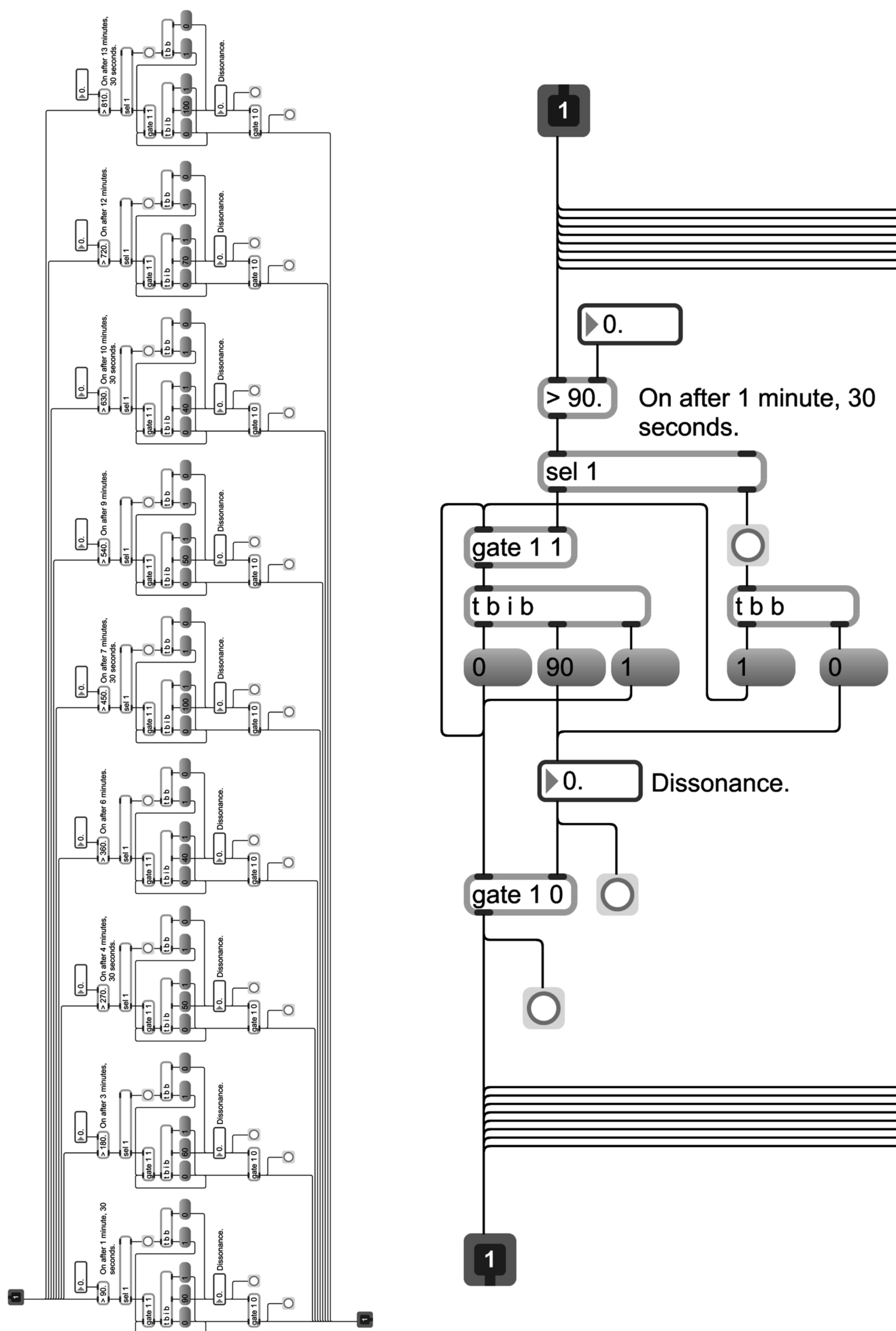

Figure B.10: Dissonance values for chord voicings (p lizards-dissonance)

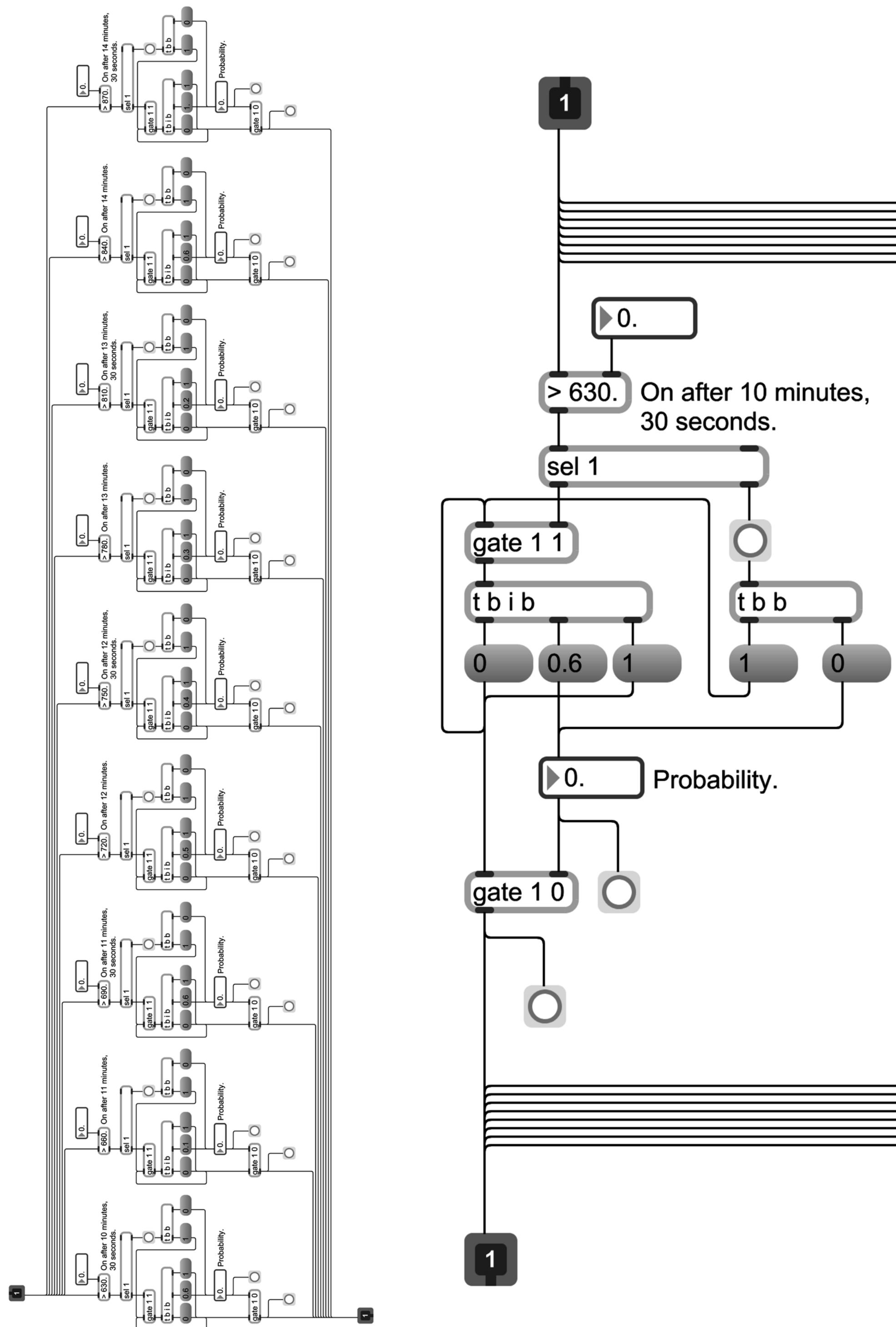

Figure B.11: X-alphabet factor oracle probabilities (p sound-file-player-oracle-probability)

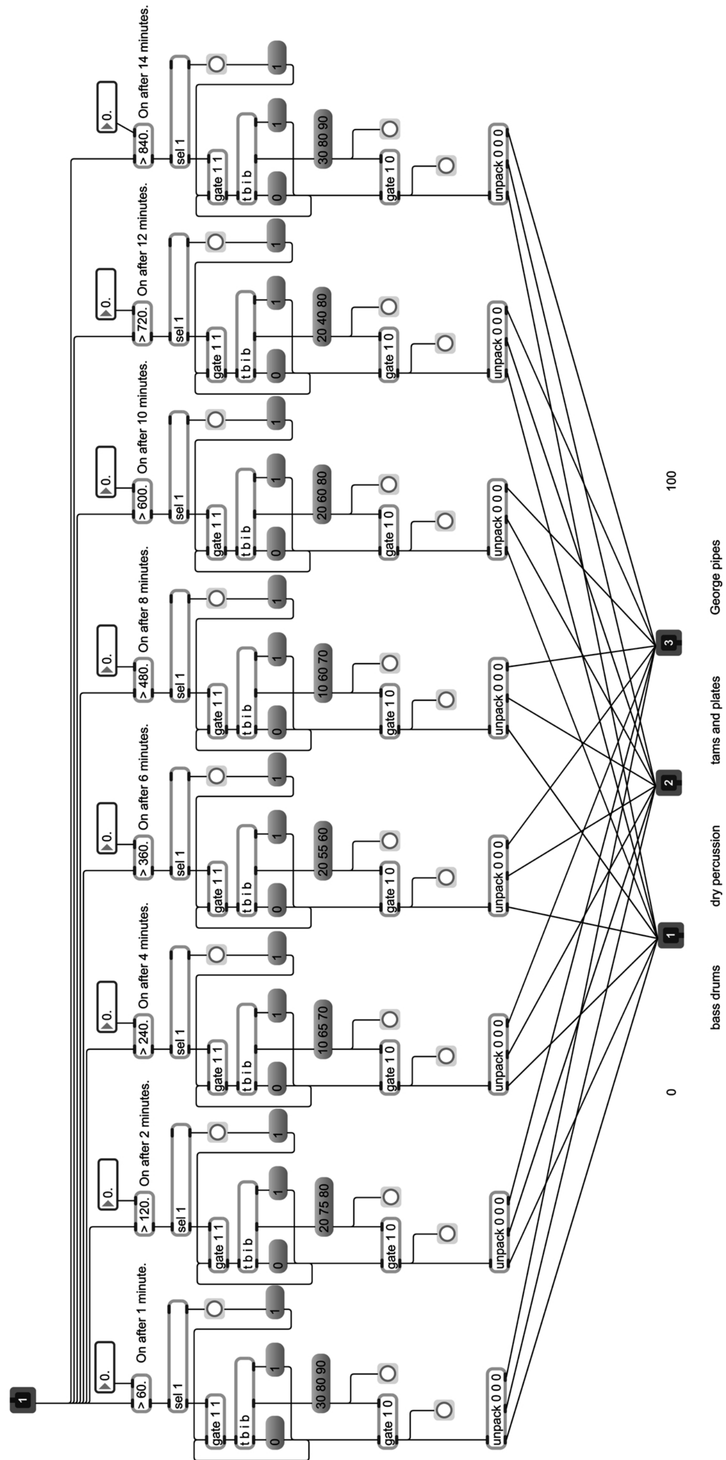

Figure B.12: Weights for percussion flock timbral group selection (p weights)

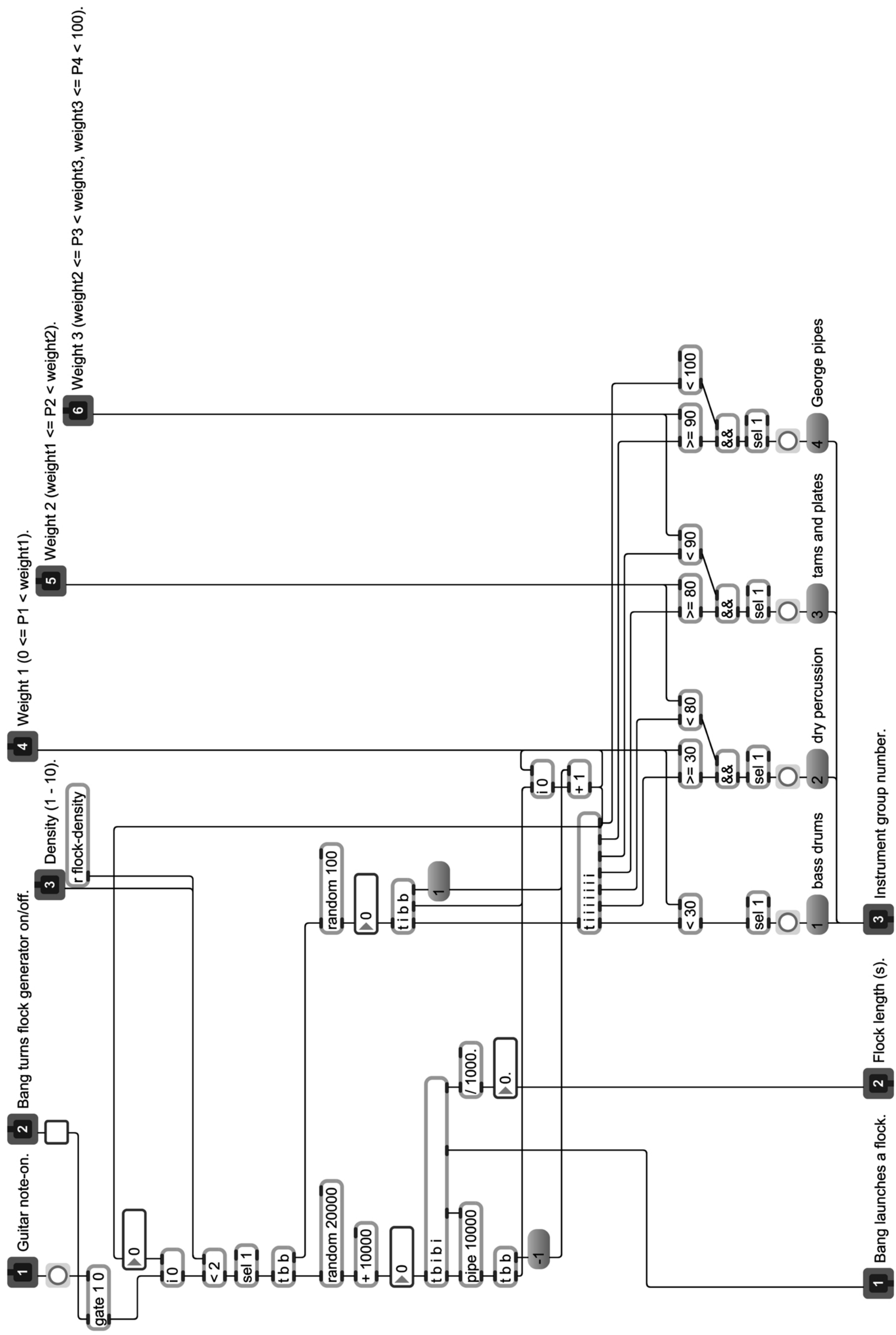

Figure B.13: Timbral group selection for percussion flocks (p selectFlock)

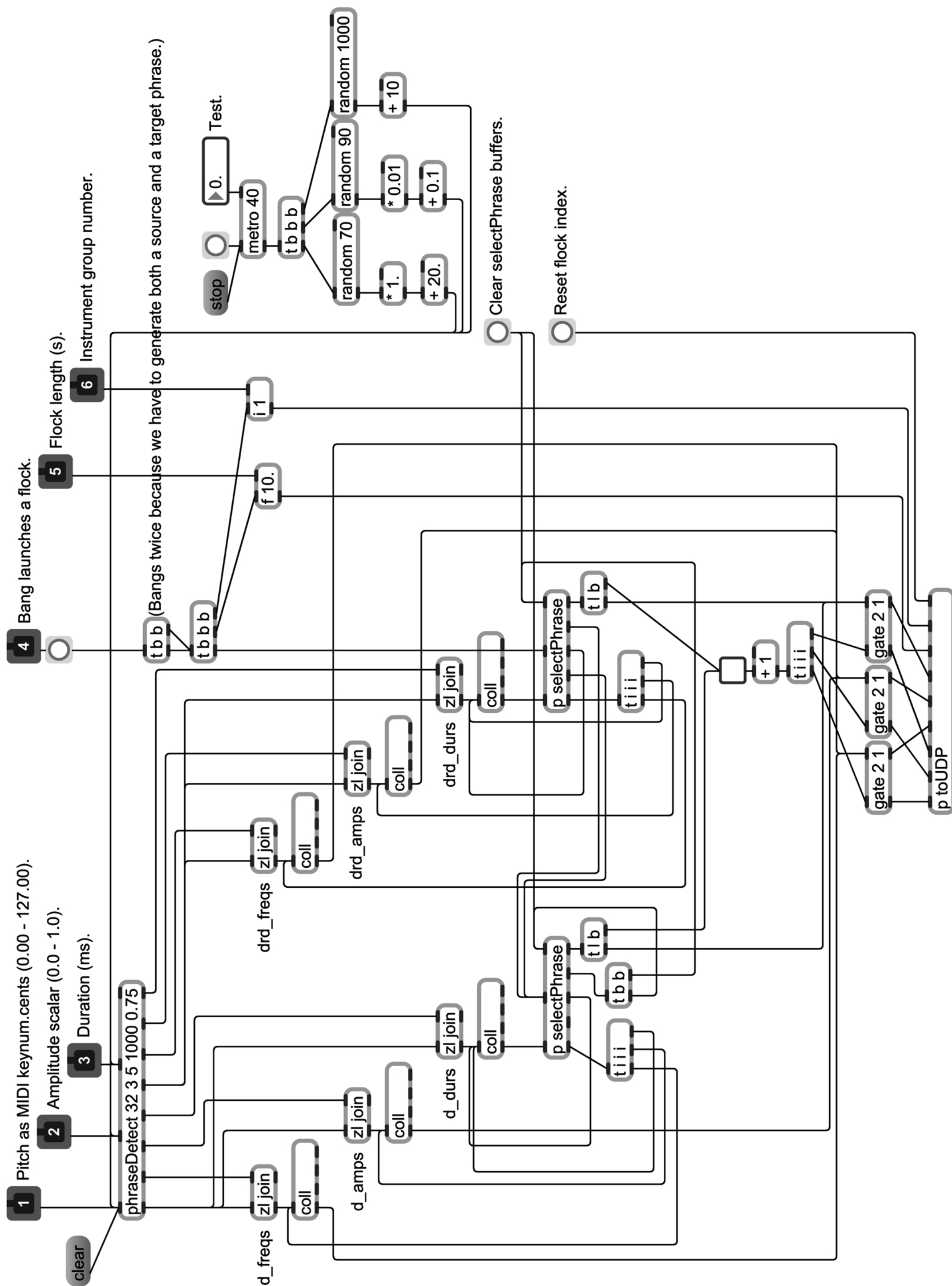

Figure B.14: Phrase detection (p phrase-detection-for-robots)

Figure B.15: Phrase selection (p selectPhrase)

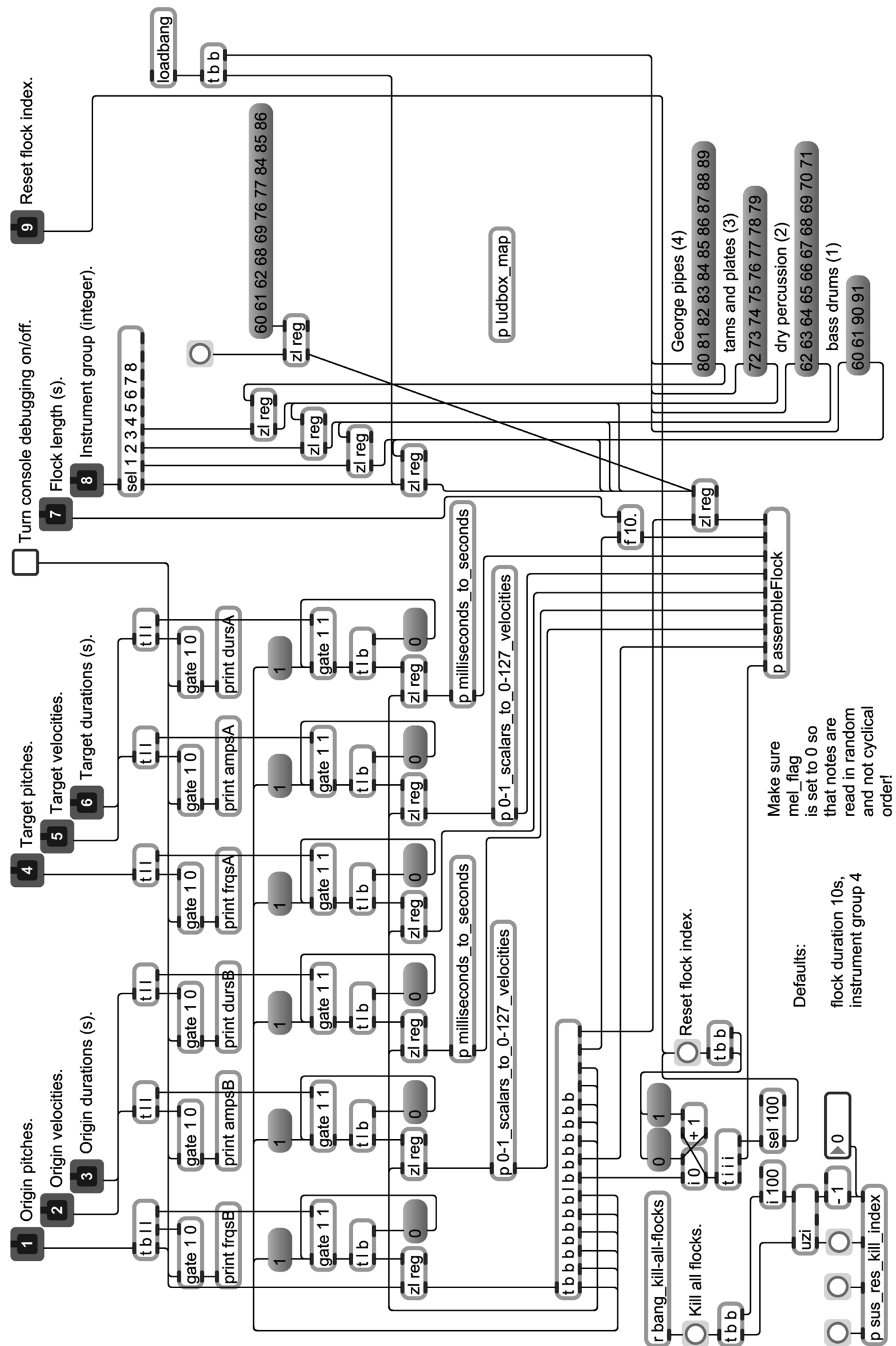

Figure B.16: Flocking information sent via OSC over UDP to Supercollider (p toUDP)

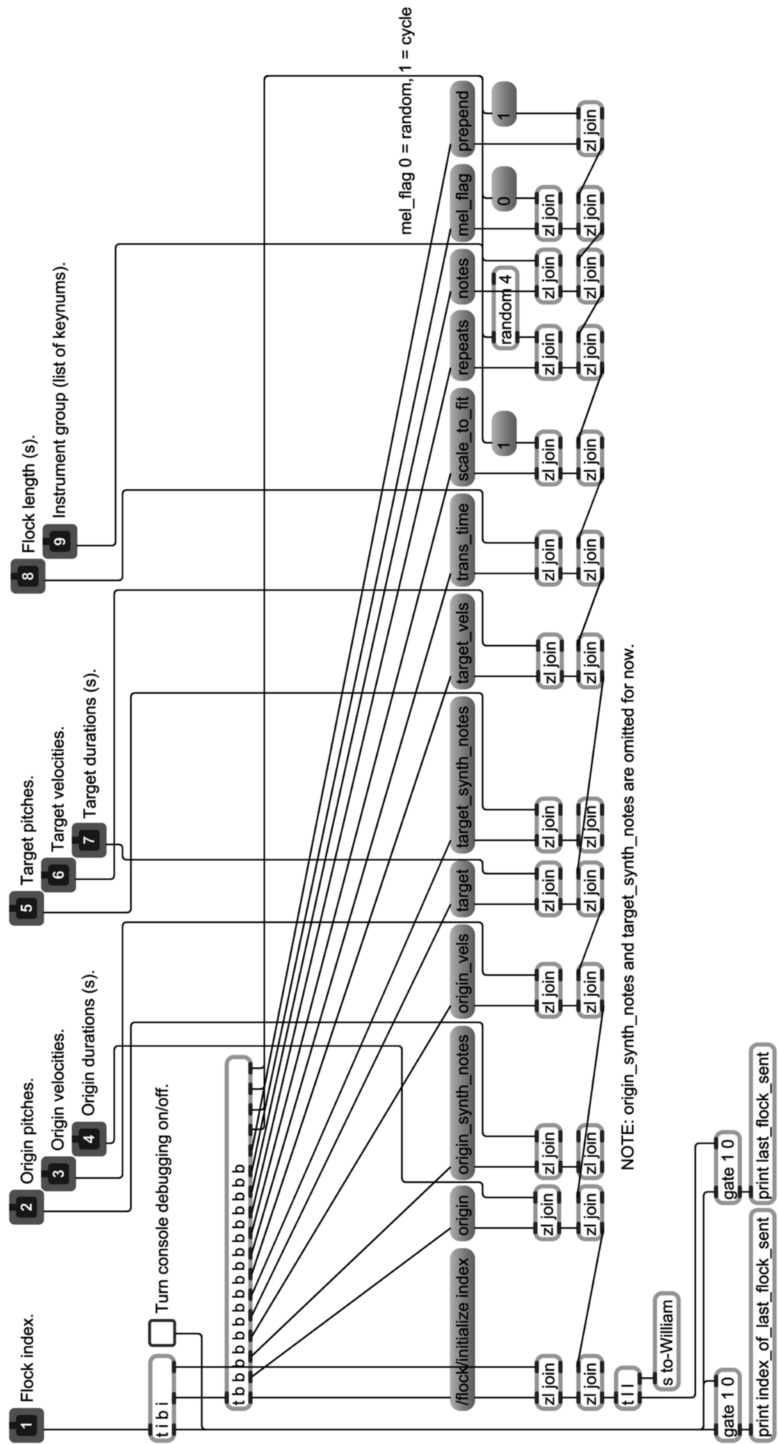

Figure B.17: OSC message formatting for flock initialization (p assembleFlock)

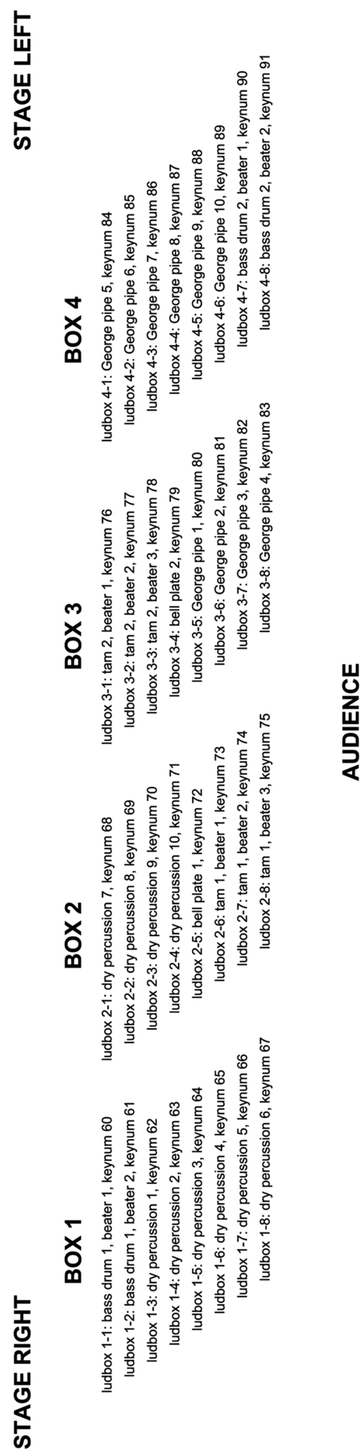

Figure B.18: Ludbot MIDI trigger assignments (p ludbox_map)

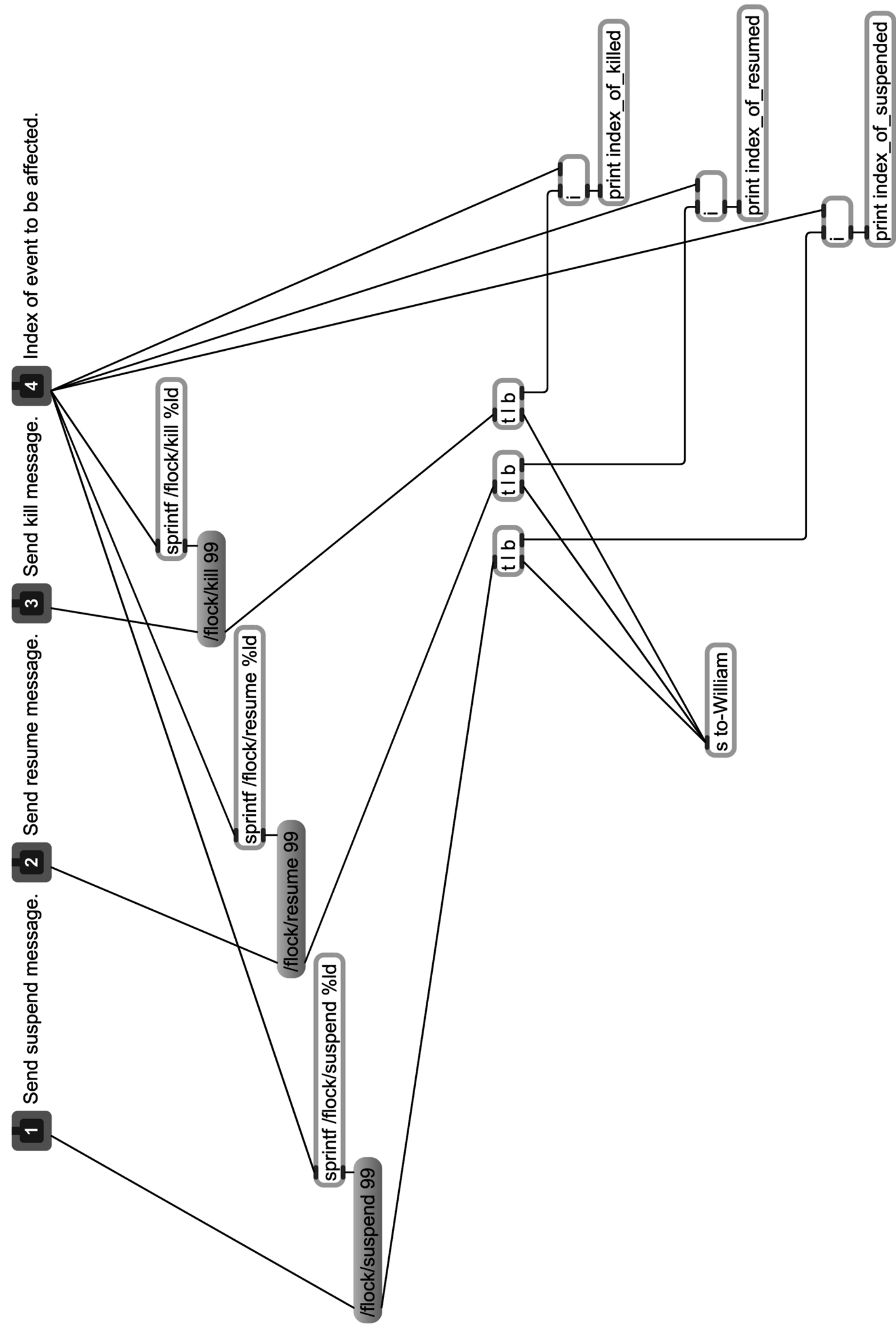

Figure B.19: OSC for flock suspension, resumption, and destruction (p sus_res_kill_index)

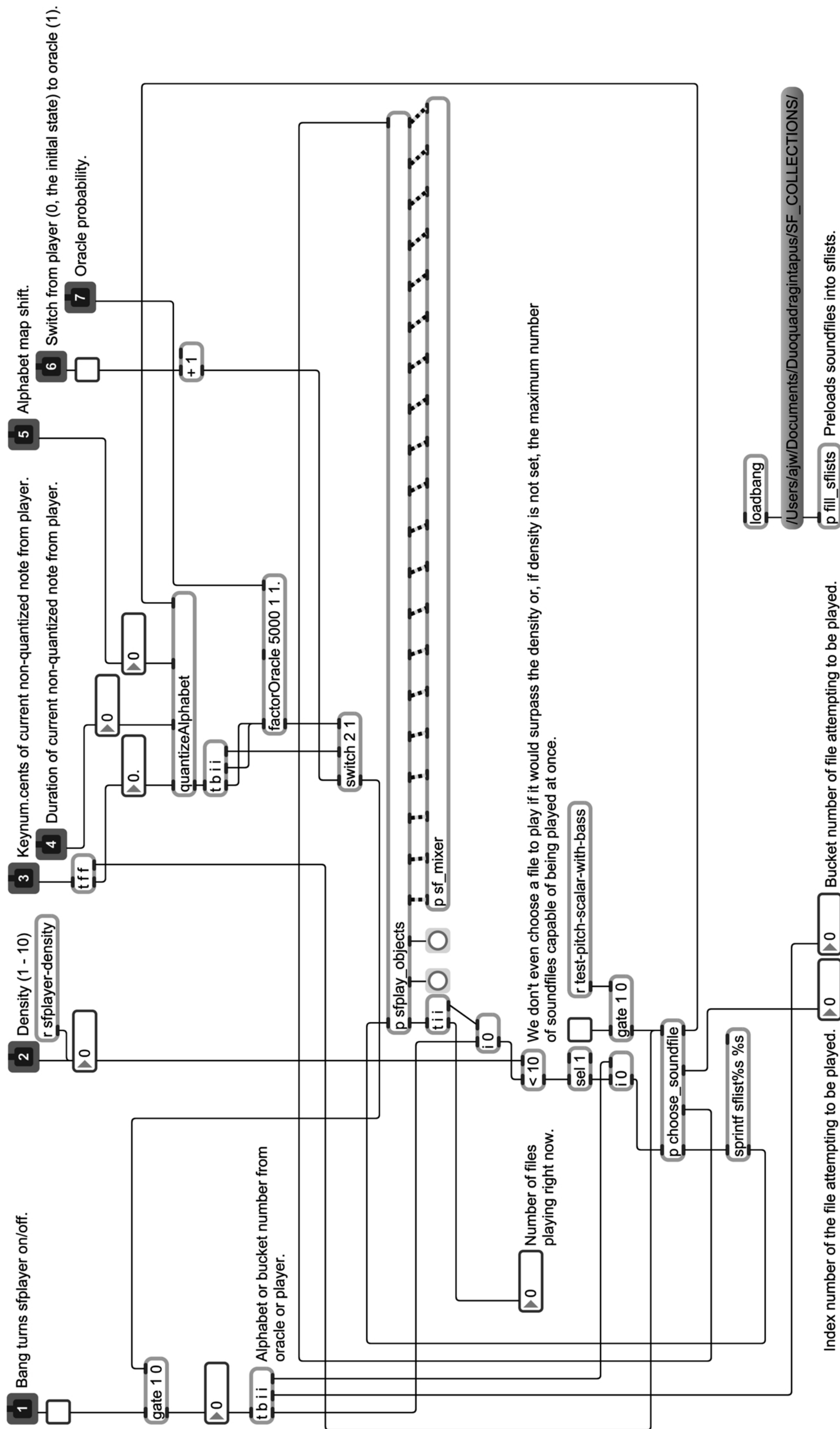

Figure B.20: X-alphabet pitch-duration pairs mapped to sound file “buckets” (p sfplayer)

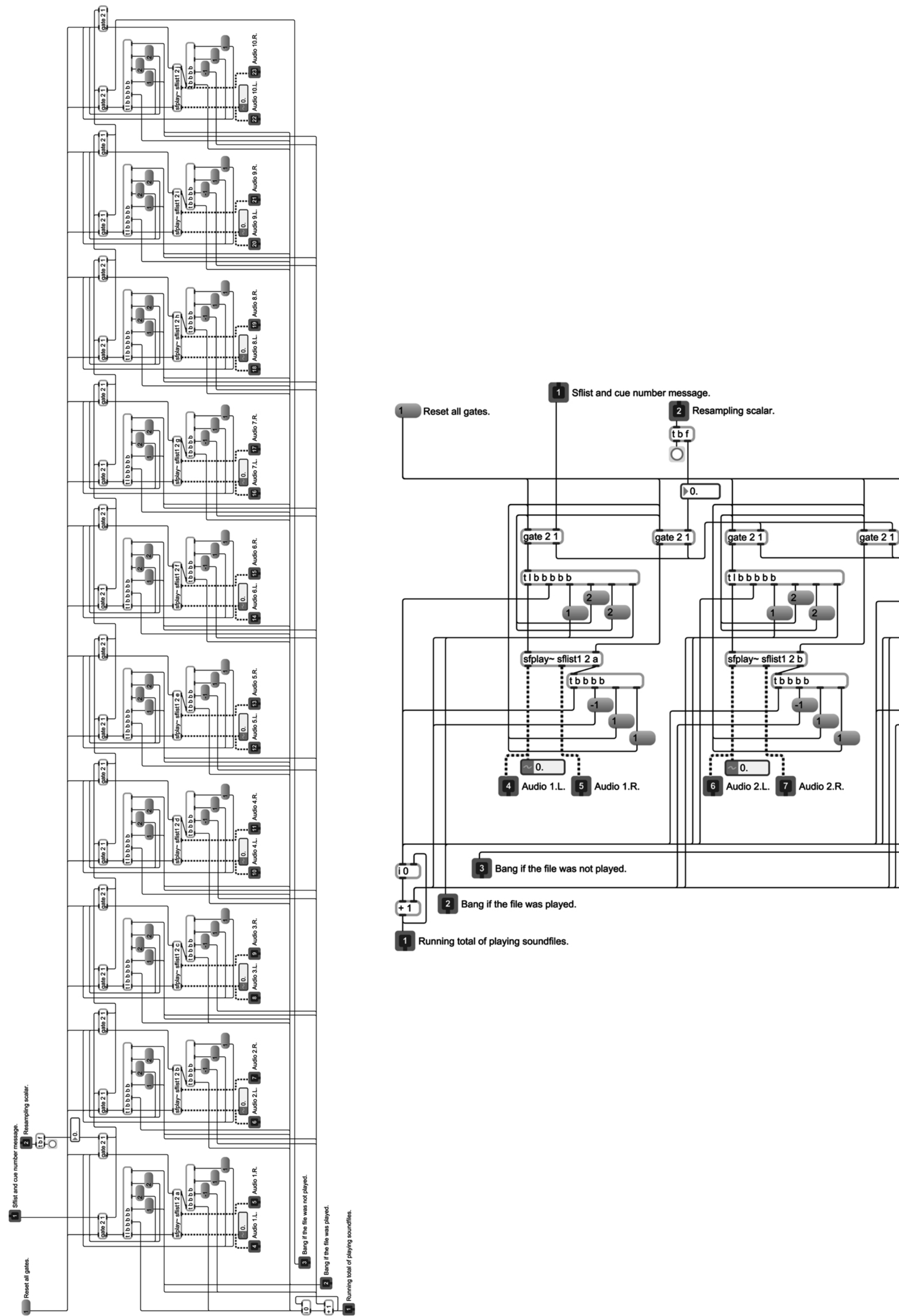

Figure B.21: Sound file players (p sfplay_objects)

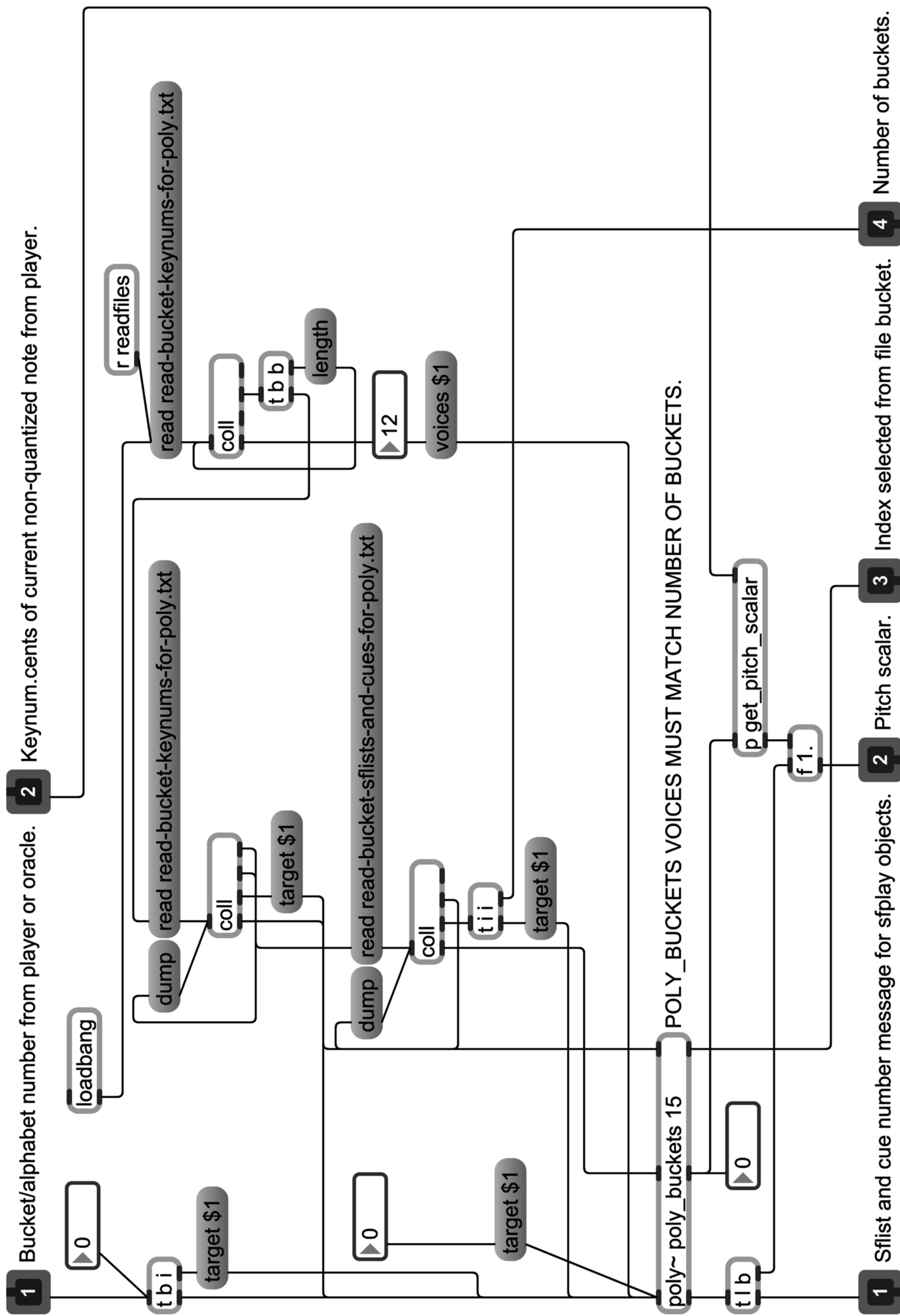

Figure B.22: Sound file bucket selection (p choose_soundfile)

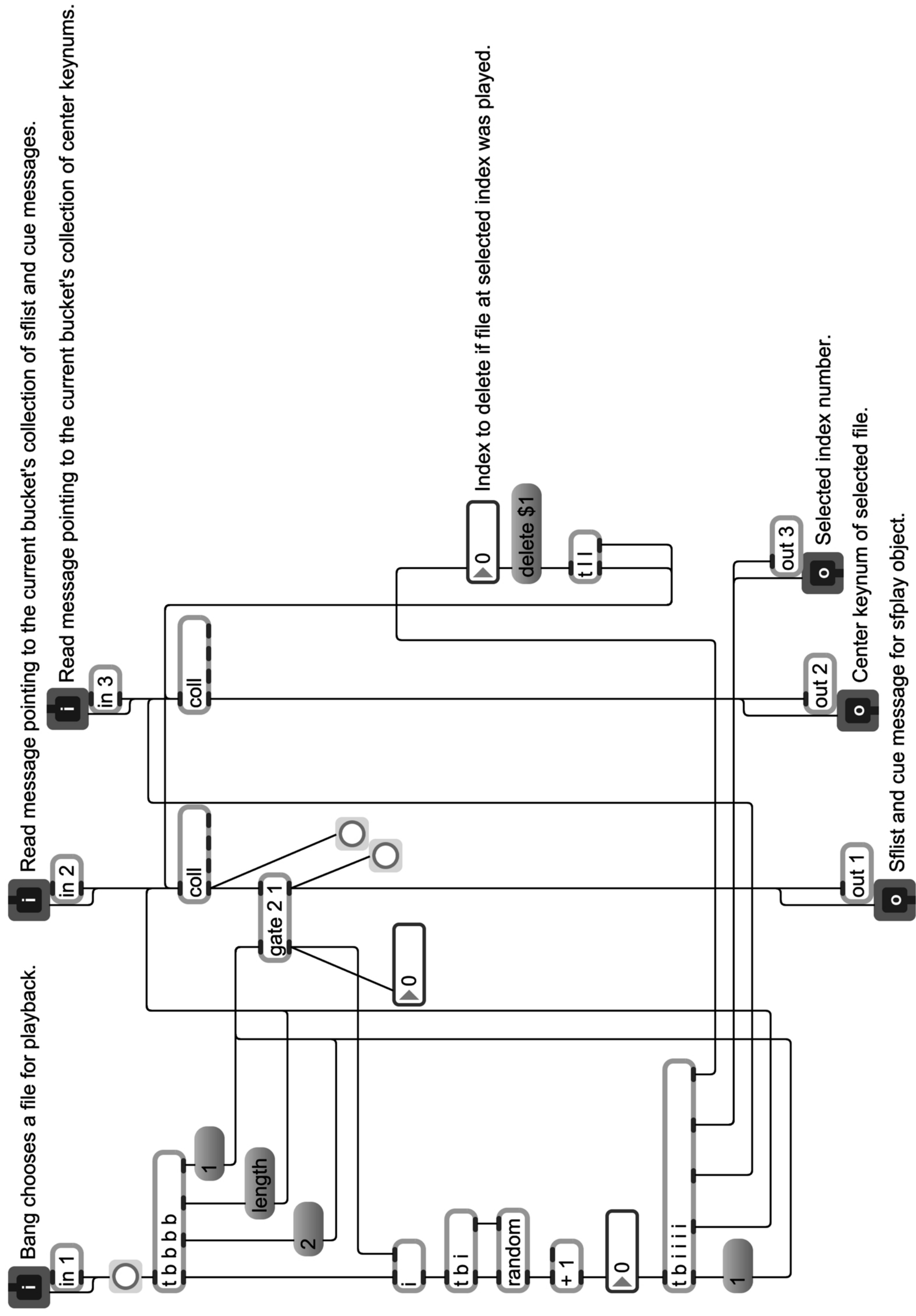

Figure B.23: Sound file selection (poly~ poly_buckets)

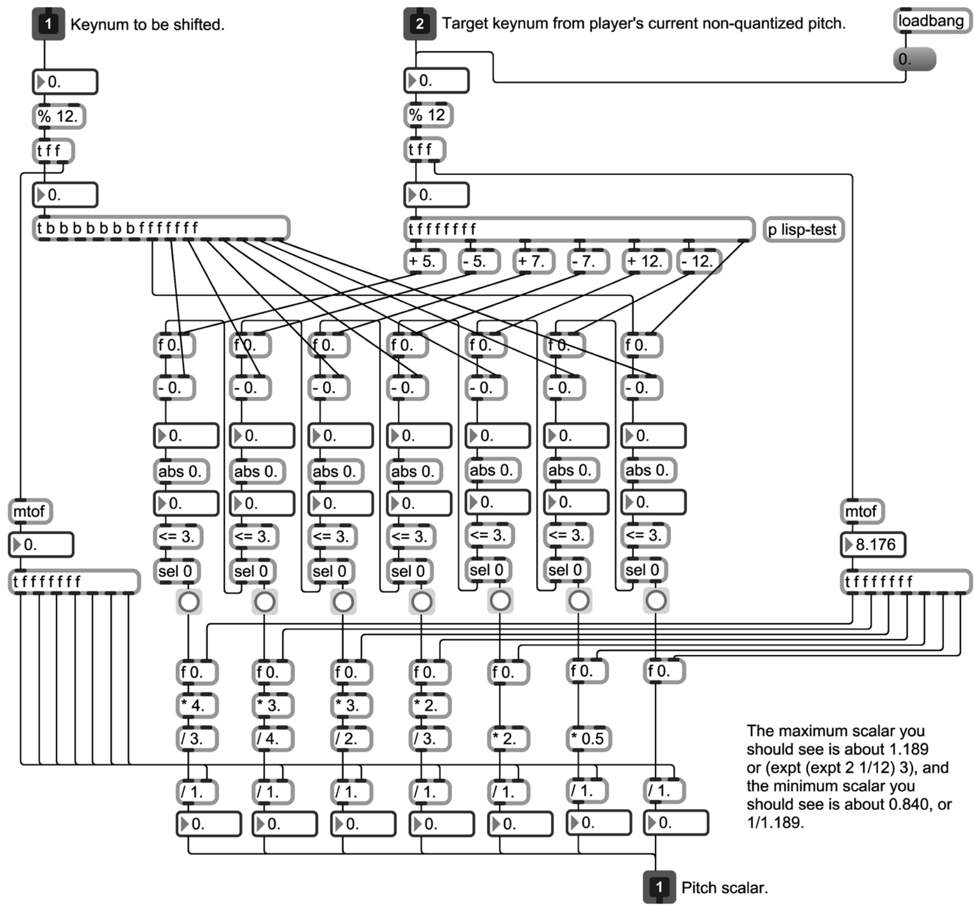

Figure B.24: Sound file transposition (p get_pitch_scalar)

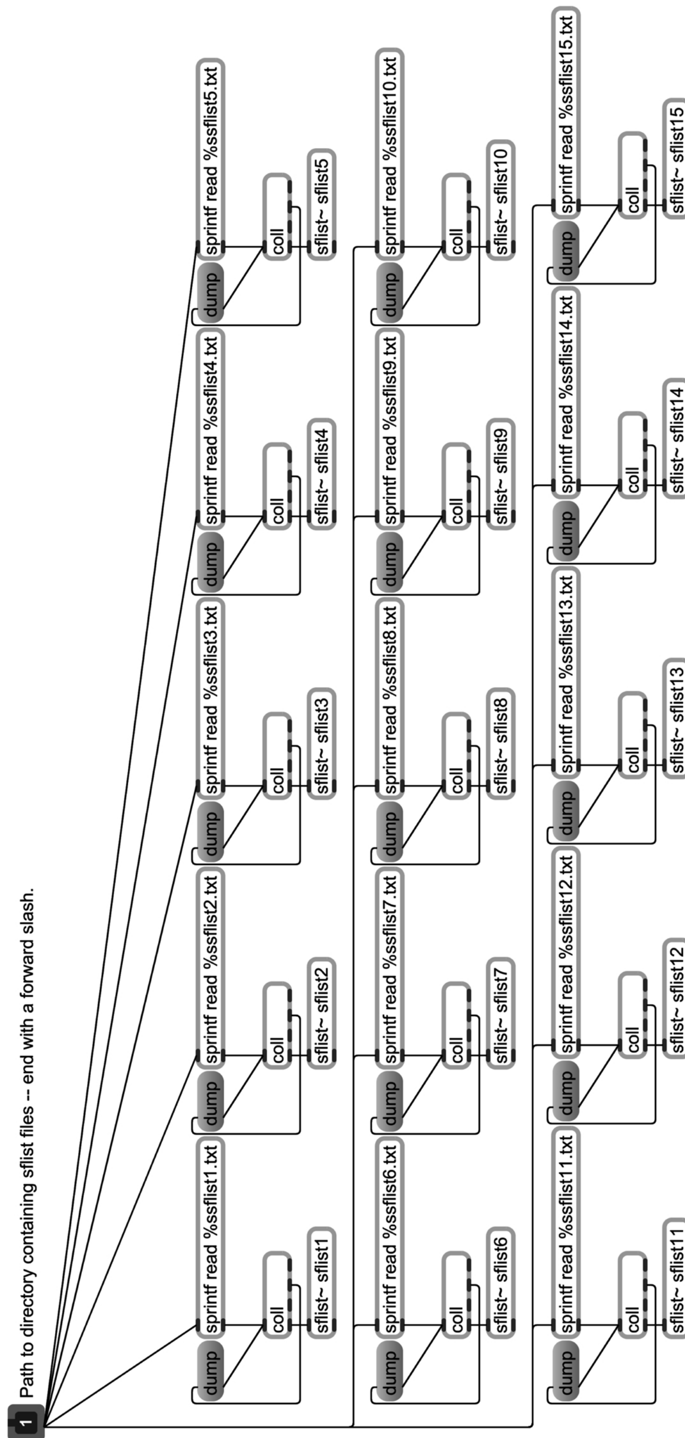

Figure B.25: Sound file → sound file list assignments (p fill_sflists)

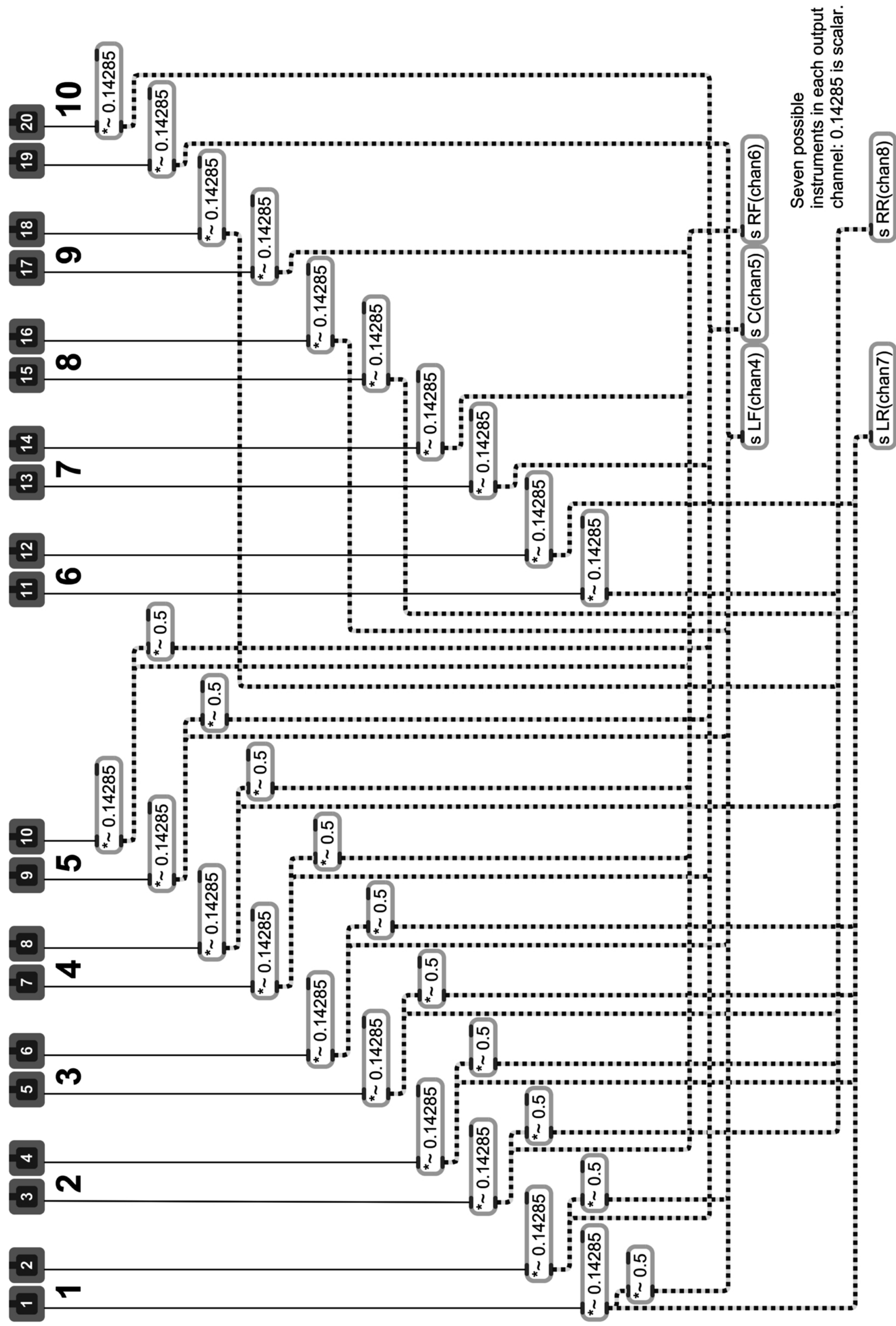

Figure B.26: 5.1 surround mix for pre-rendered sound files (p sf_mixer)

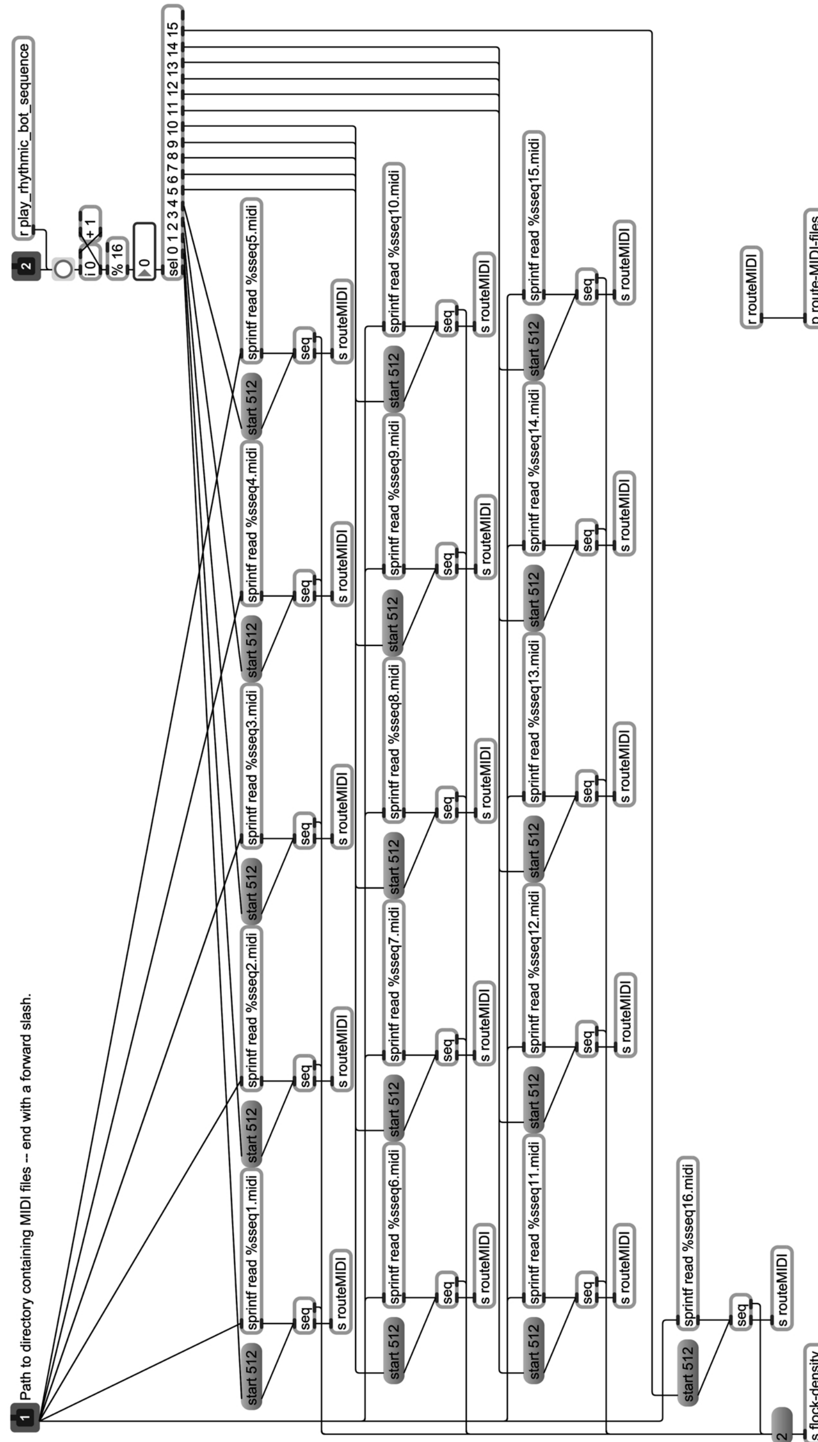

Figure B.27: Player for pre-composed ludbot sequences (p MIDI_bot_player)

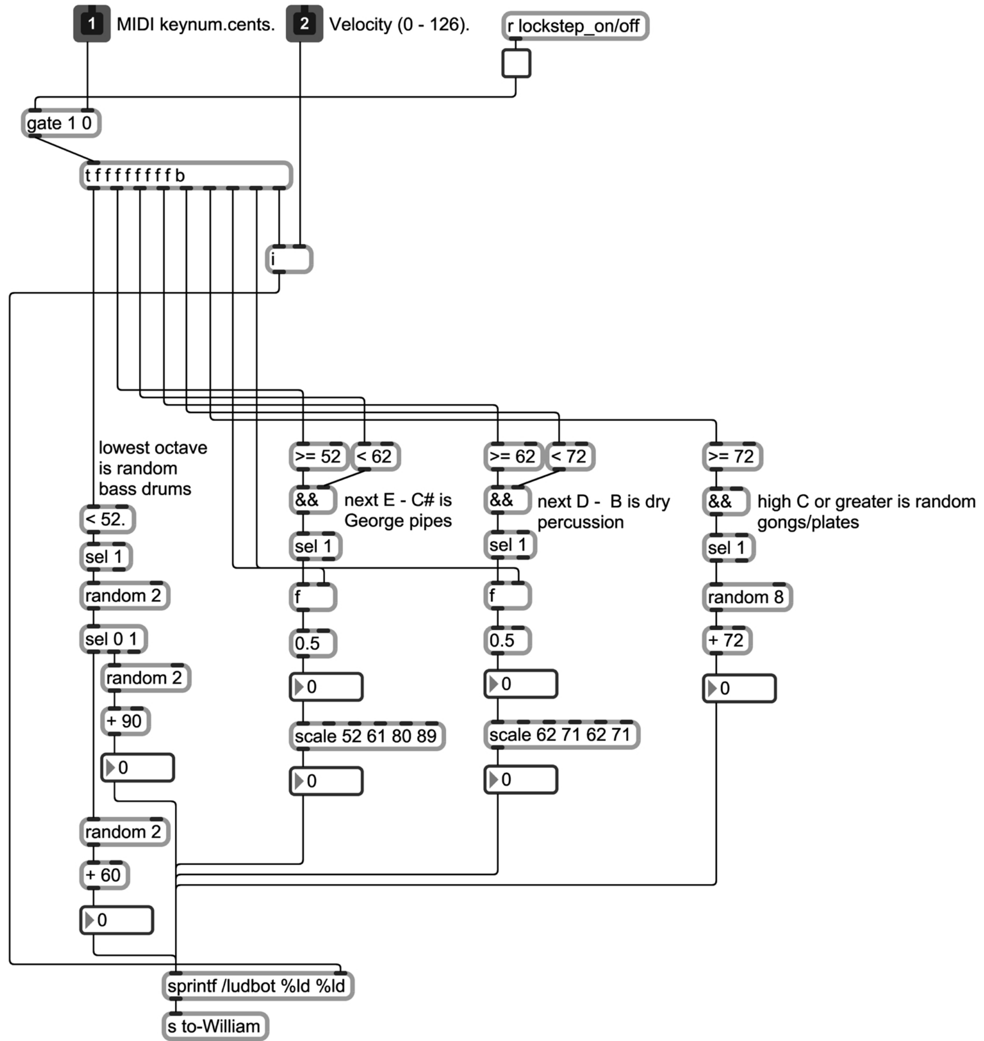

Figure B.28: Guitar → ludbot mappings for ludbot "lockstep" mode (p ludbot-lockstep)

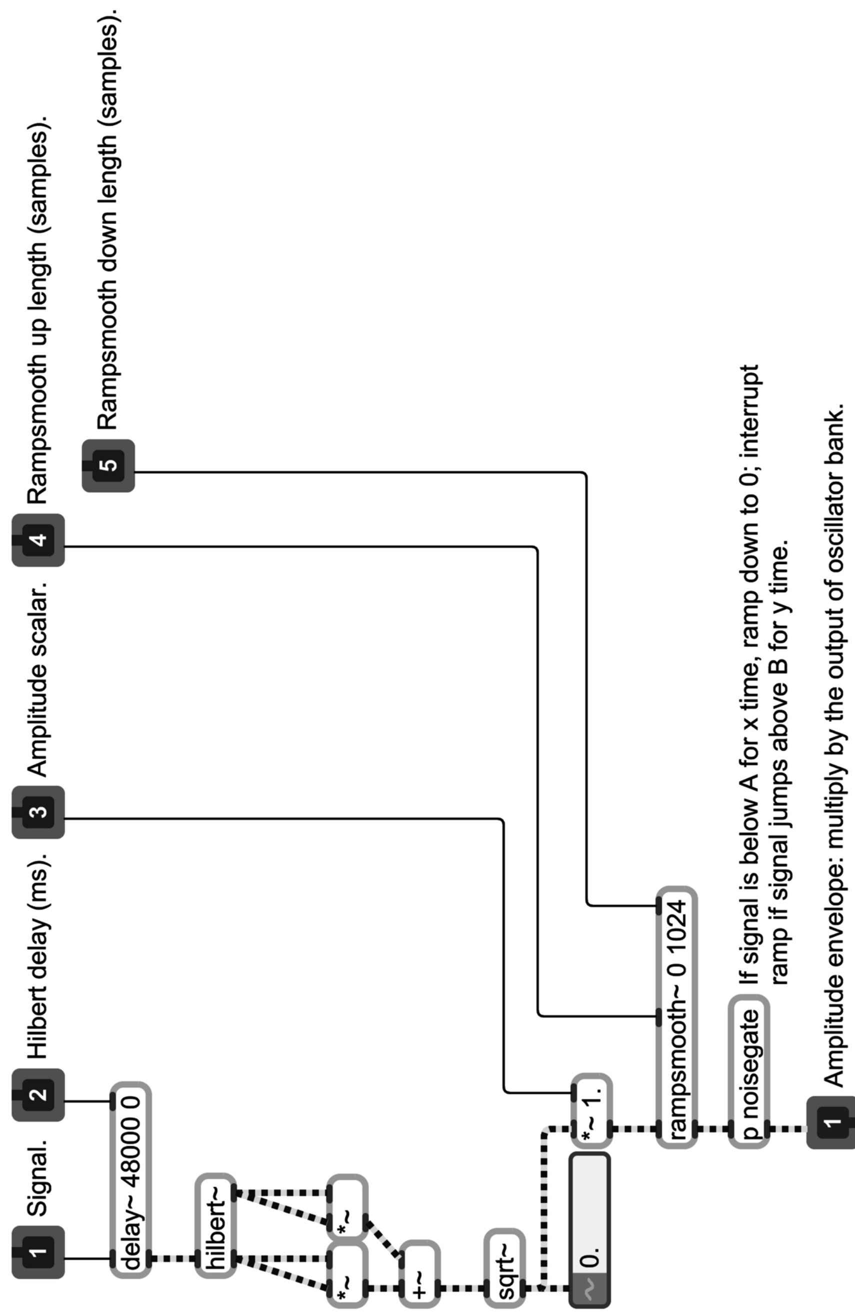

Figure B.29: Separation of guitar signal phase and amplitude (p hilbert)

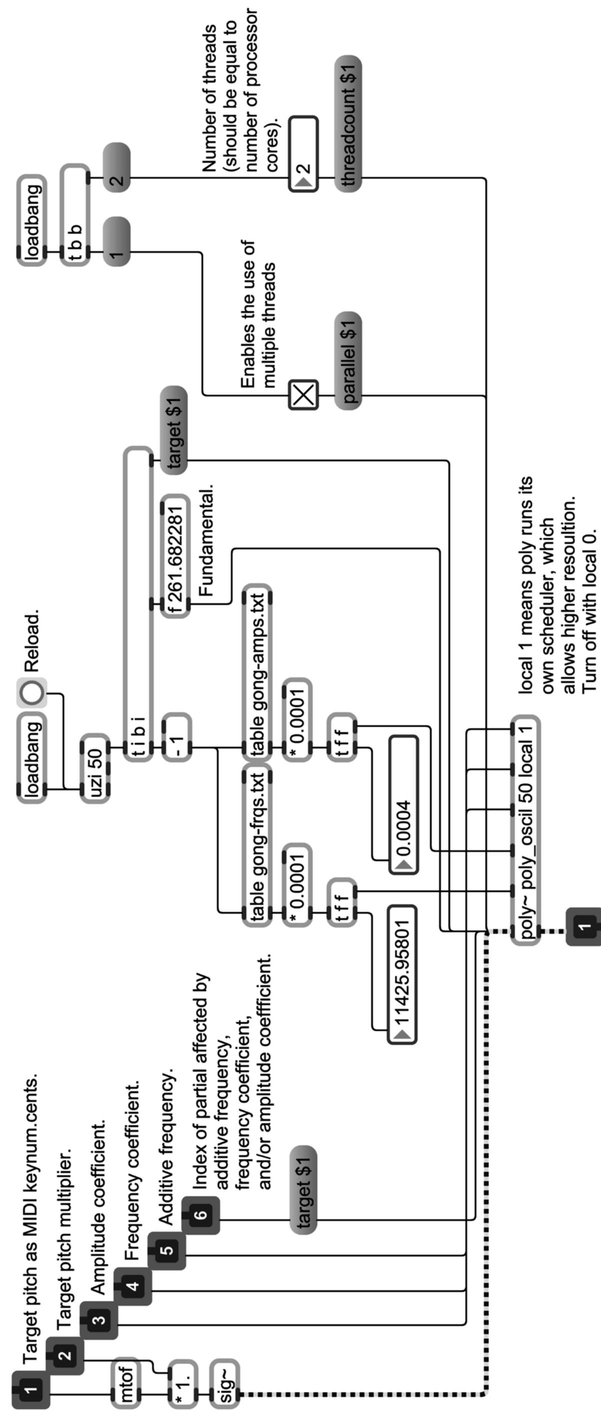

Figure B.30: Oscillator bank for gong timbre simulation (p oscilbank)

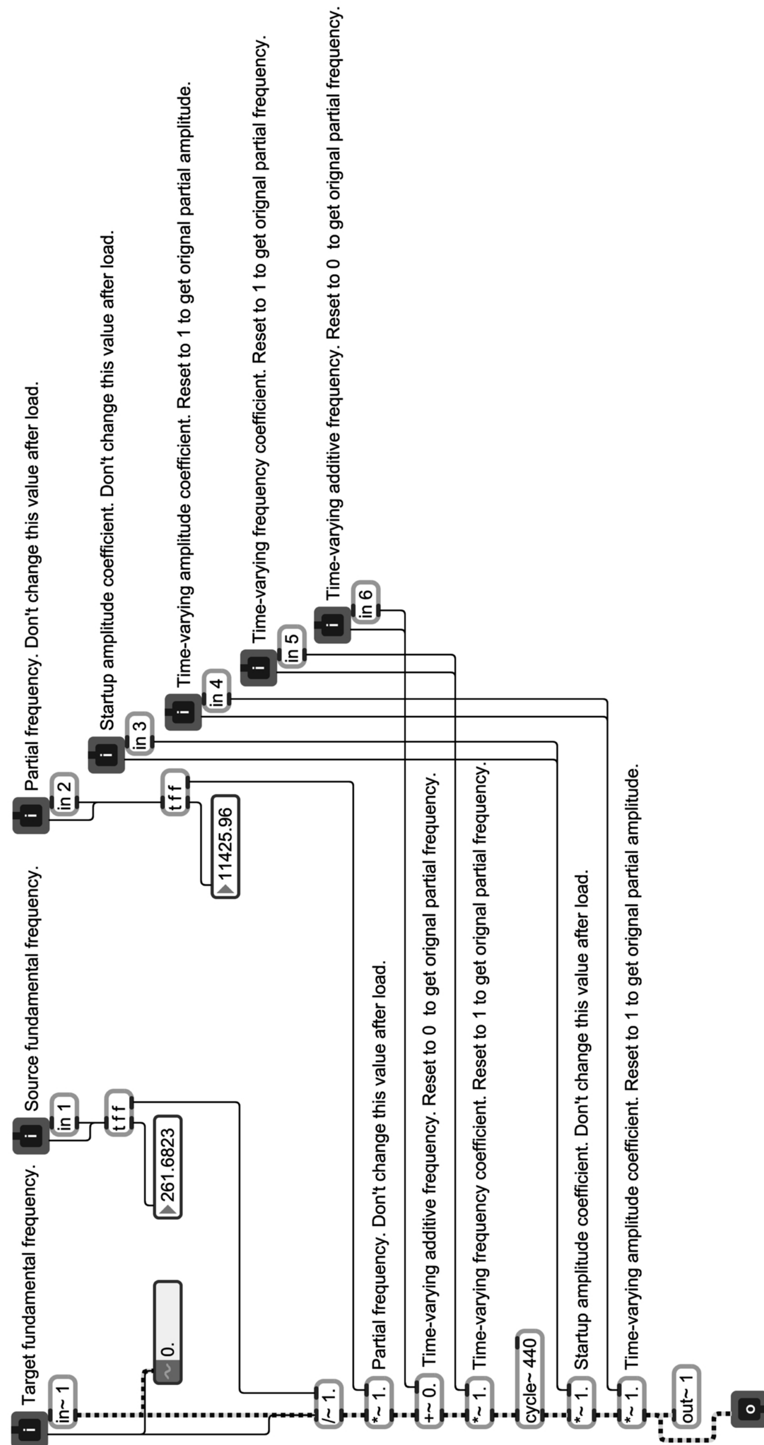

Figure B.31: Parametric control of individual oscillators (poly~ poly_oscil)

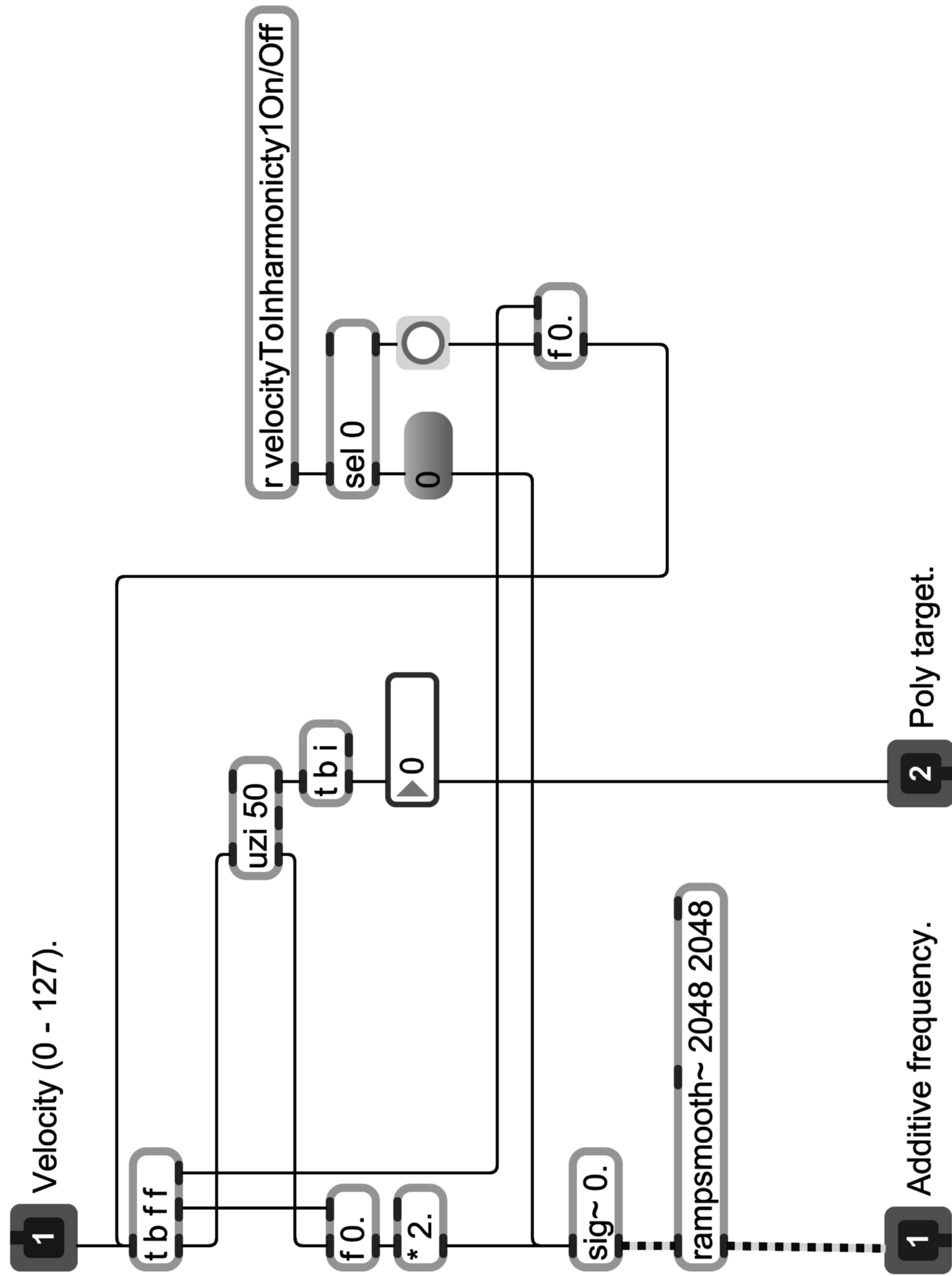

Figure B.32: Guitar velocity mapped to gong inharmonicity (p velocityToInharmonicity)

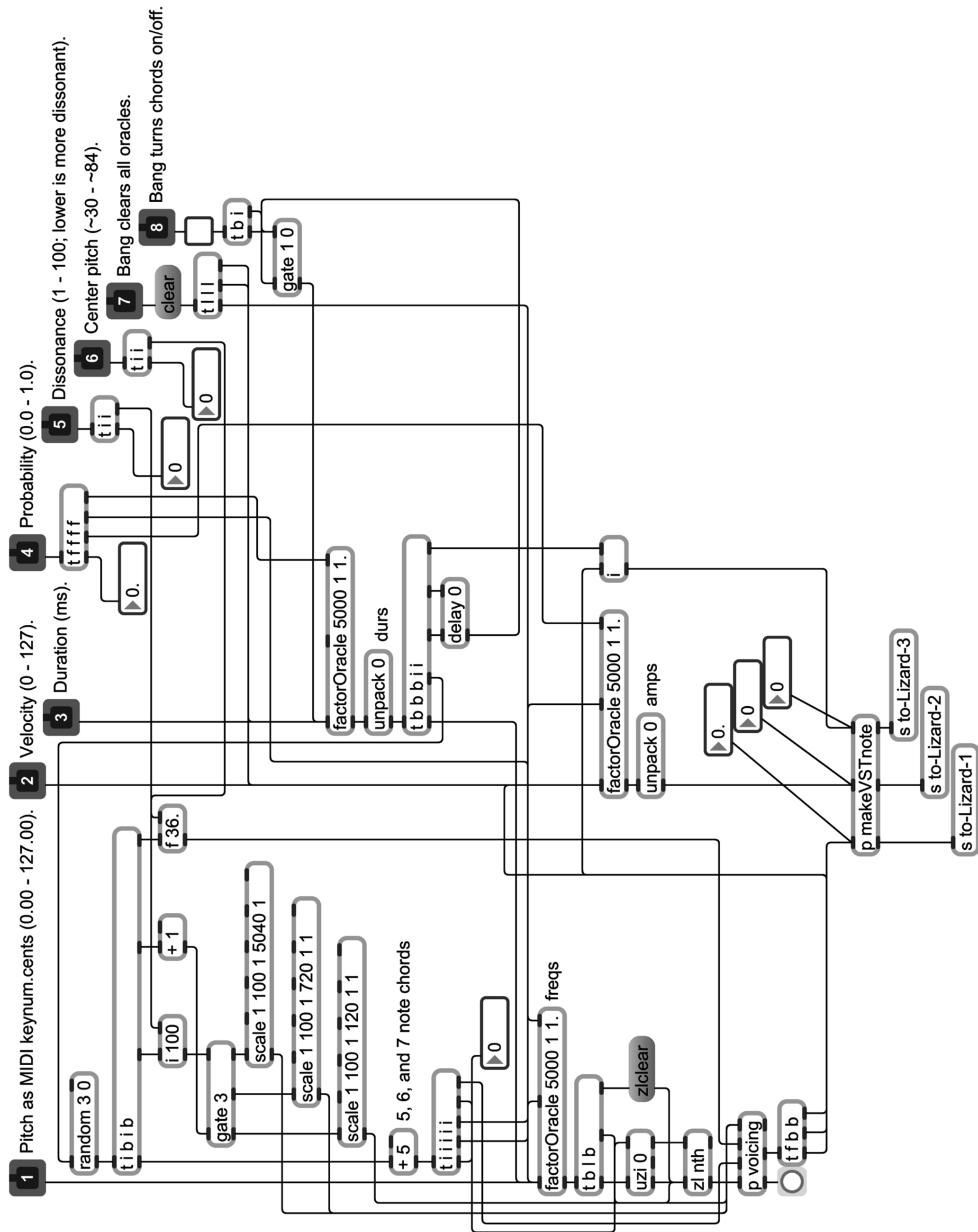

Figure B.33: Factor oracle generation of harmonic accompaniment (p harmony-oracle)

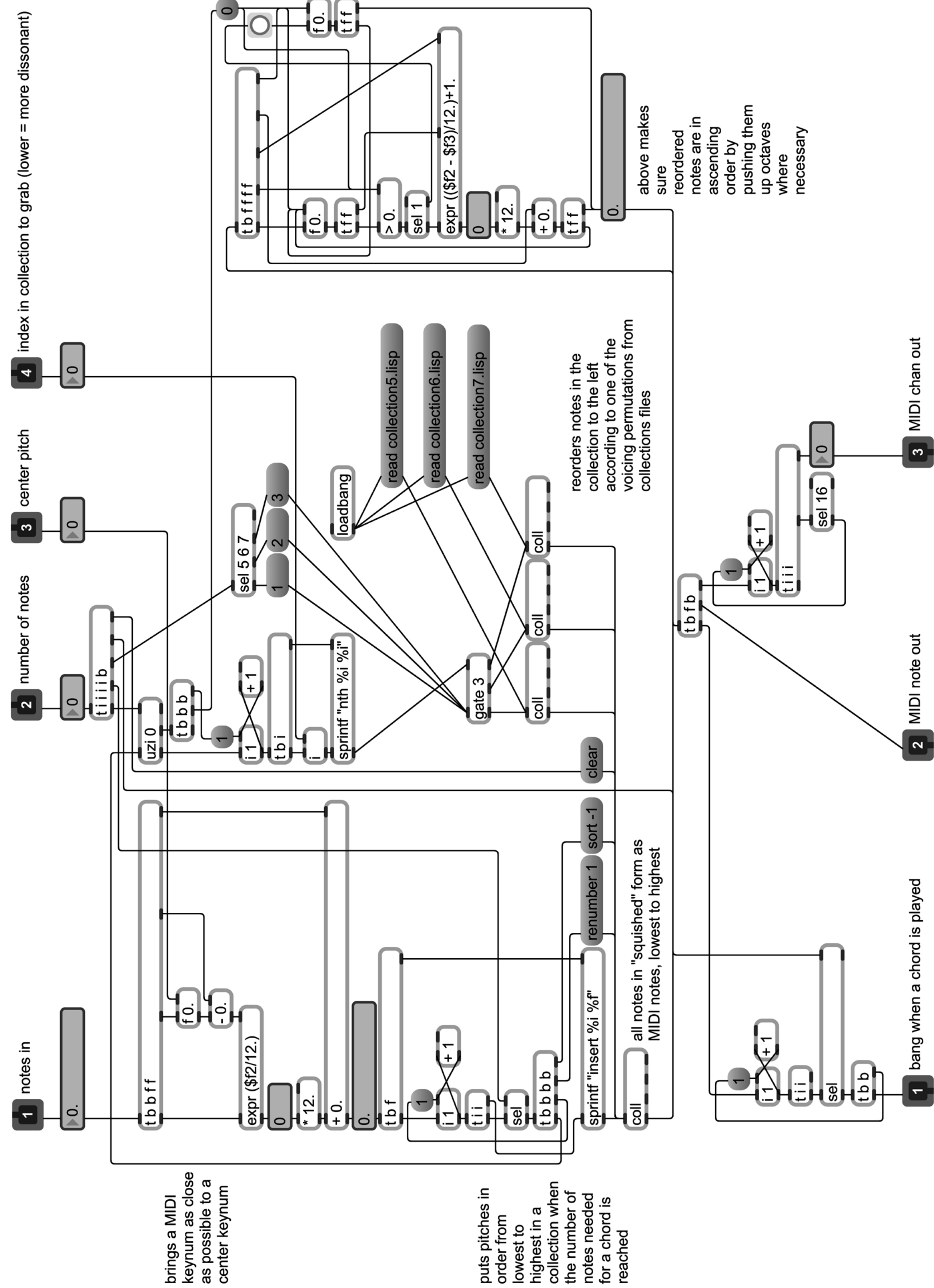

Figure B.34: Voicing selection for harmonic accompaniment (p voicing)

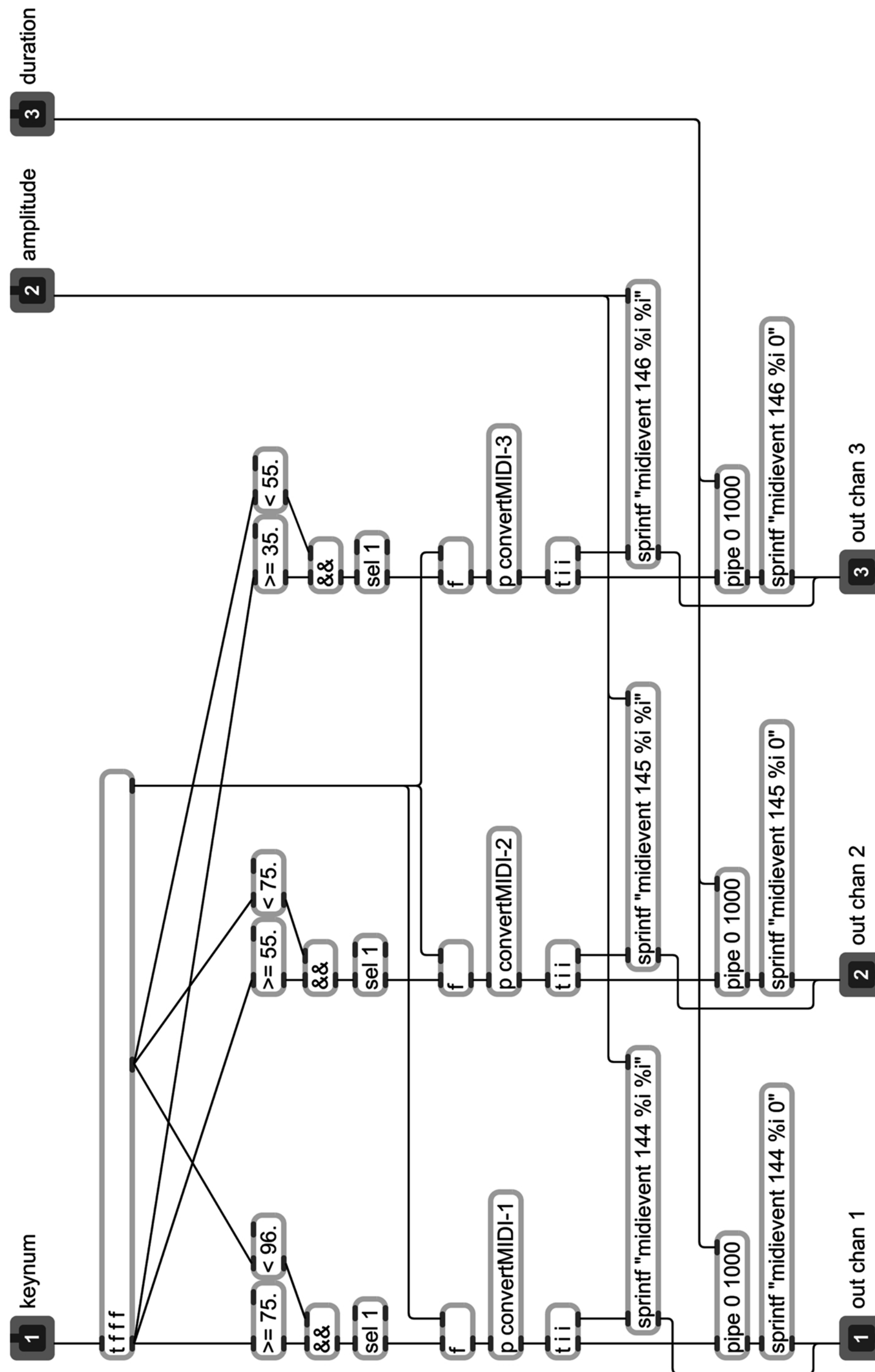

Figure B.35: MIDI information formatted for VST synthesizer plug-ins (p makeVSTnote)

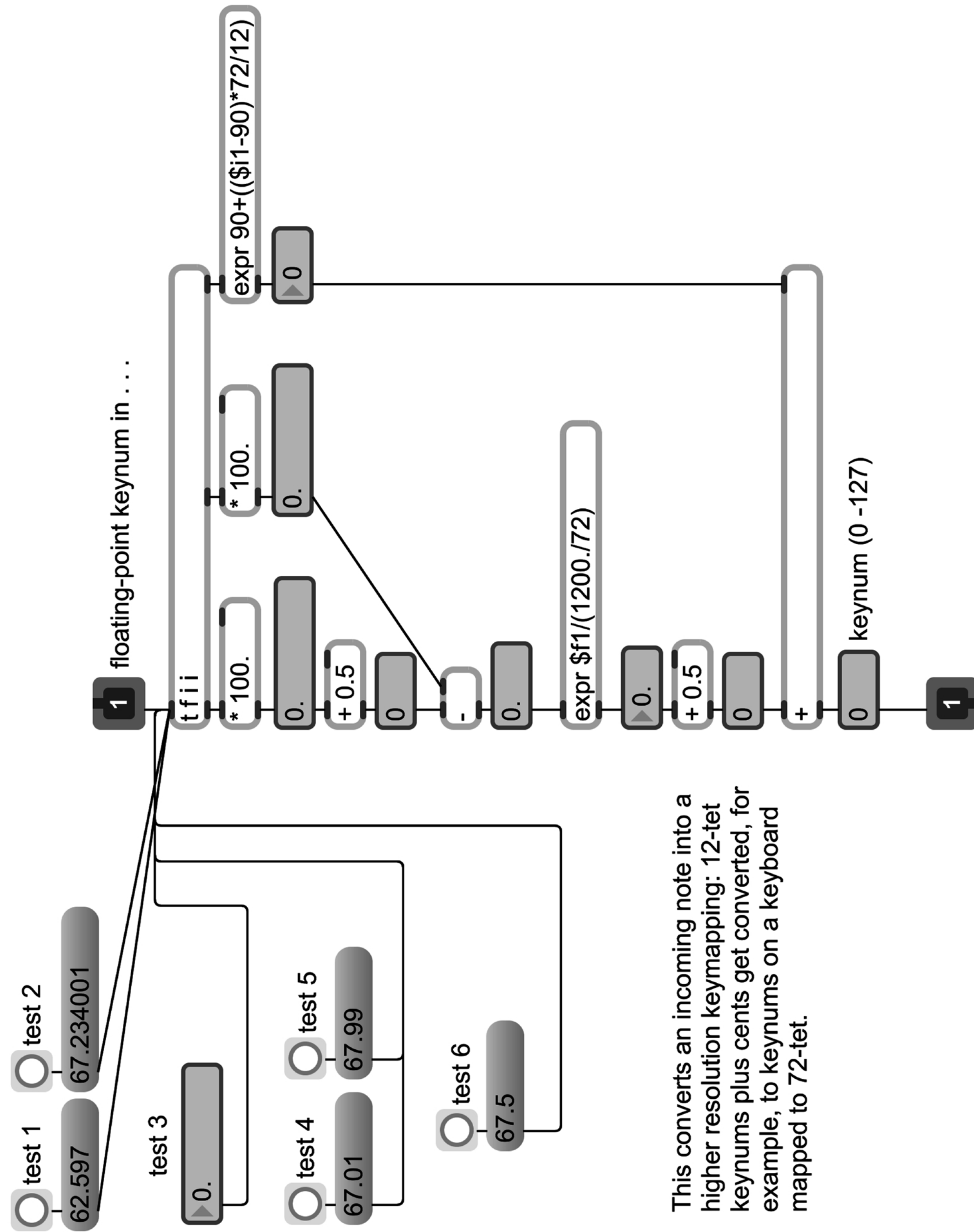

Figure B.36: Pitch values formatted for AAS synthesizers (p convertMIDI-1)

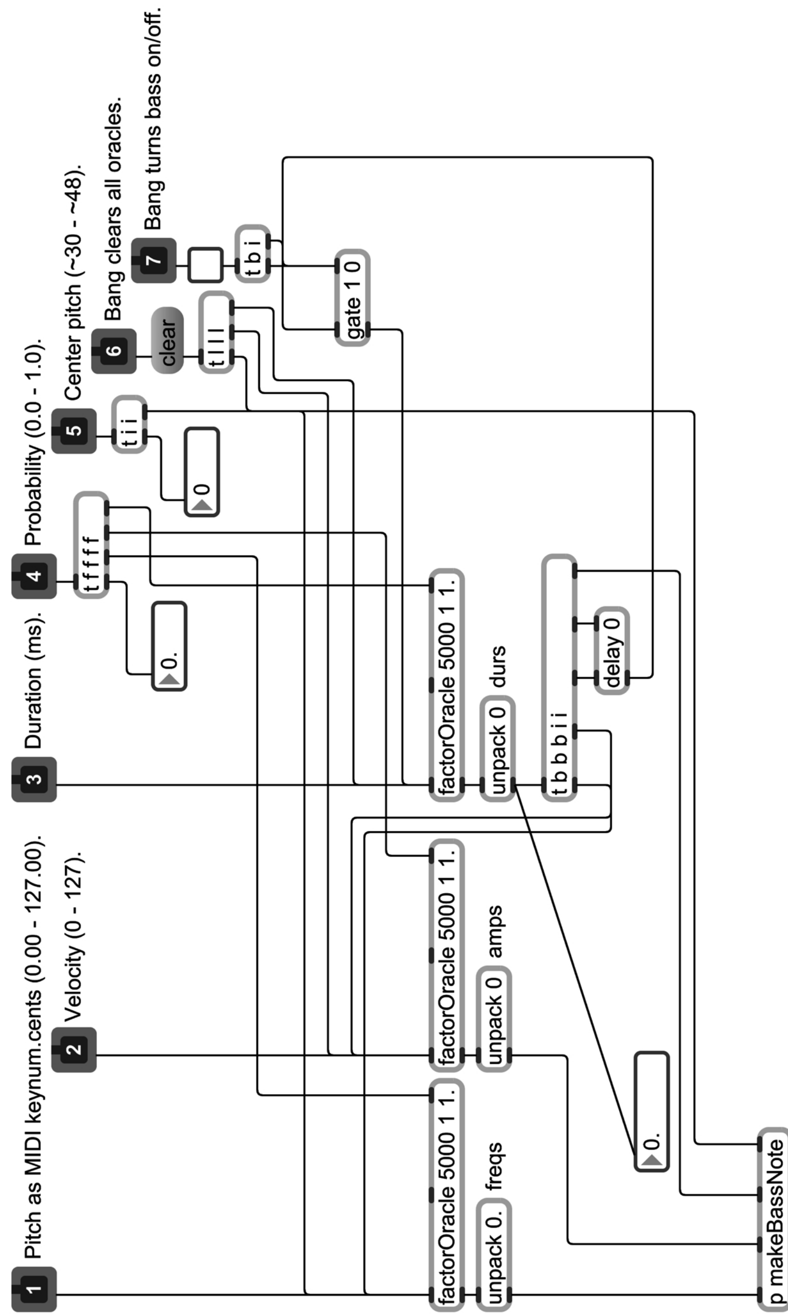

Figure B.37: Factor oracle generation of bass accompaniment (p bass-oracle)

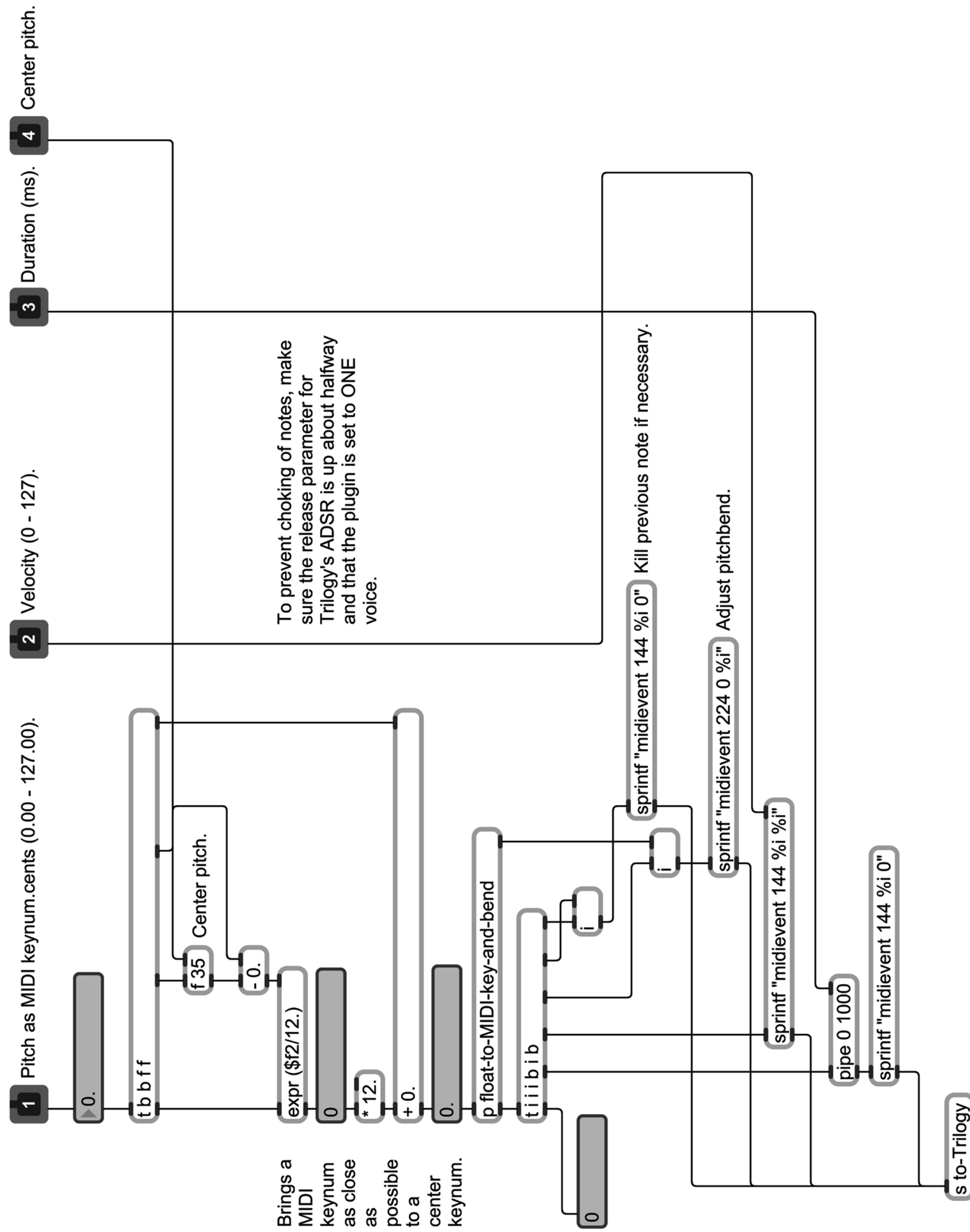

Figure B.38: MIDI information formatted for VST sampler plug-in (p makeBassNote)

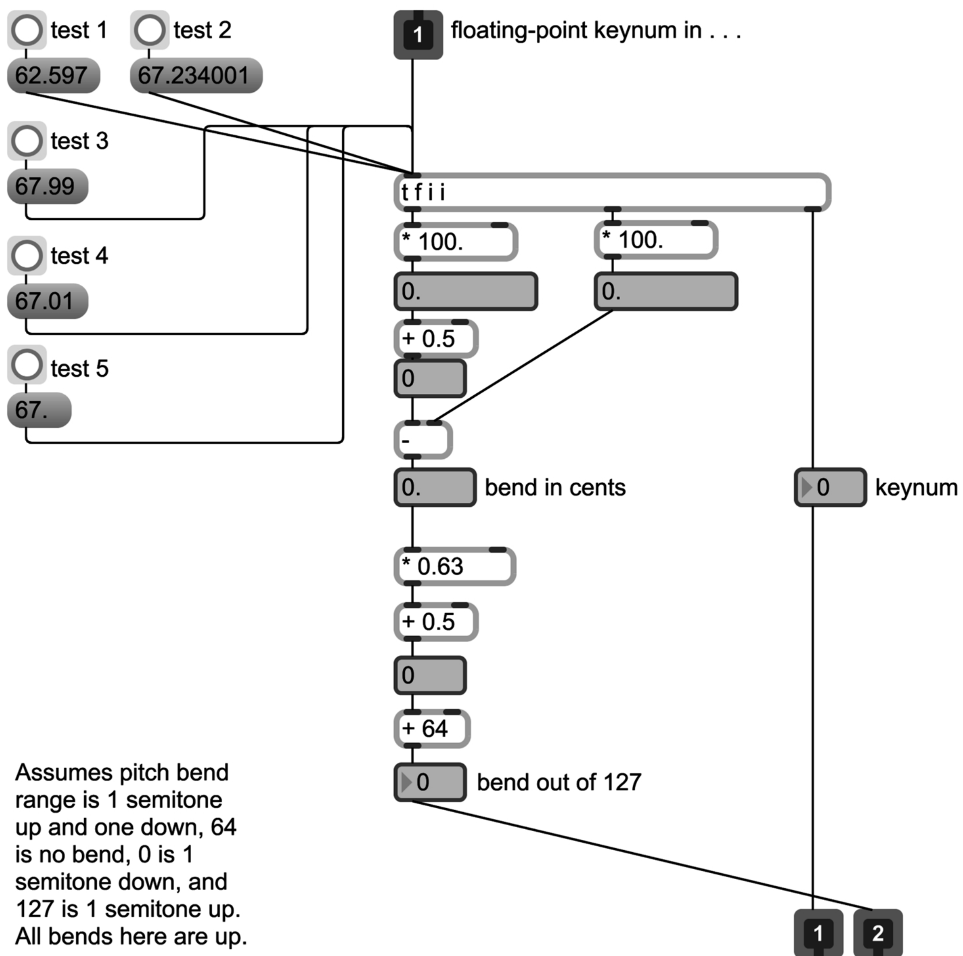

Figure B.39: Pitch values formatted for Trilogy sampler (p float-to-MIDI-key-and-bend)

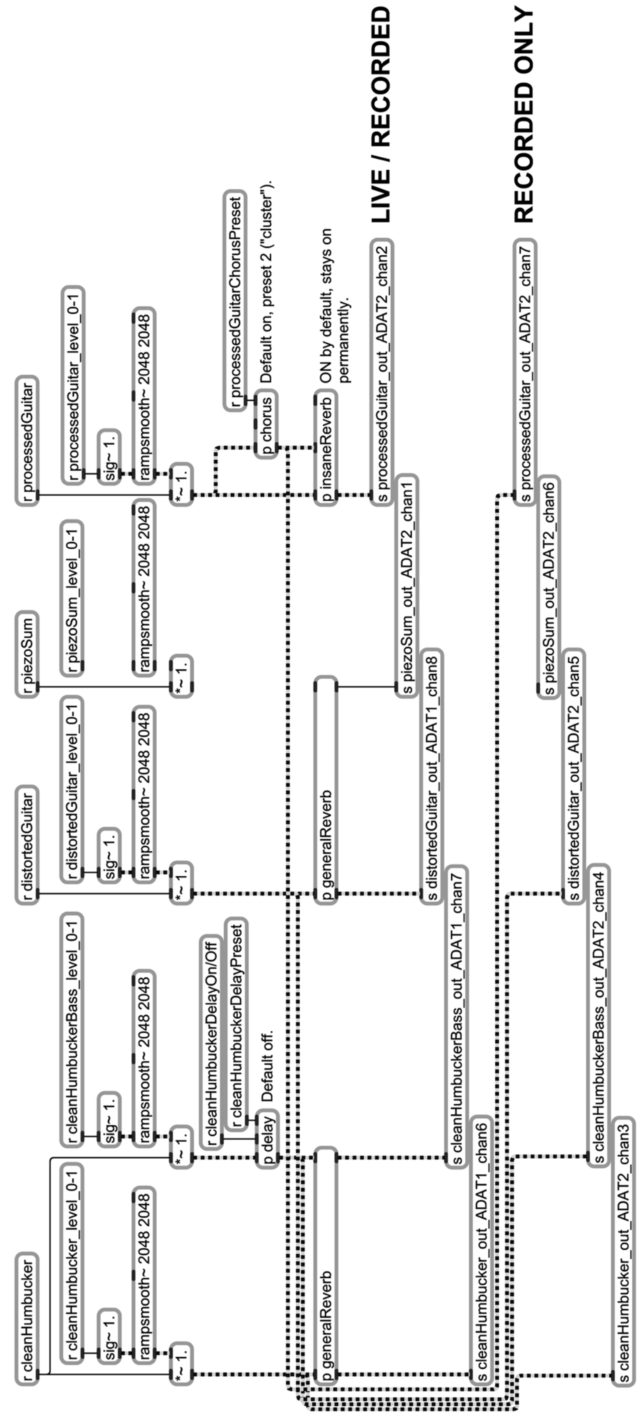

Figure B.40: Pitch-shift, chorus, and reverb effects (p guitarEffects)

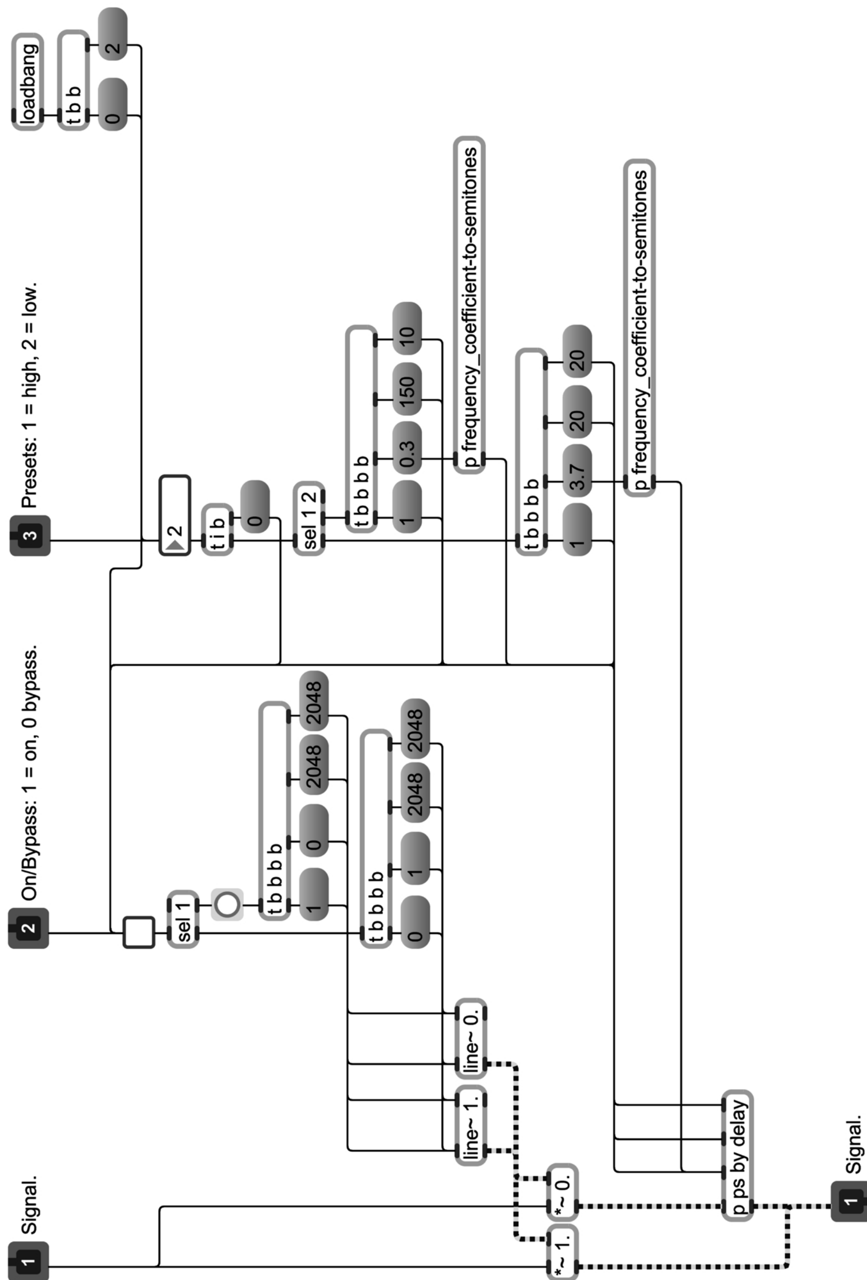

Figure B.41: Pitch-shift settings (p delay)

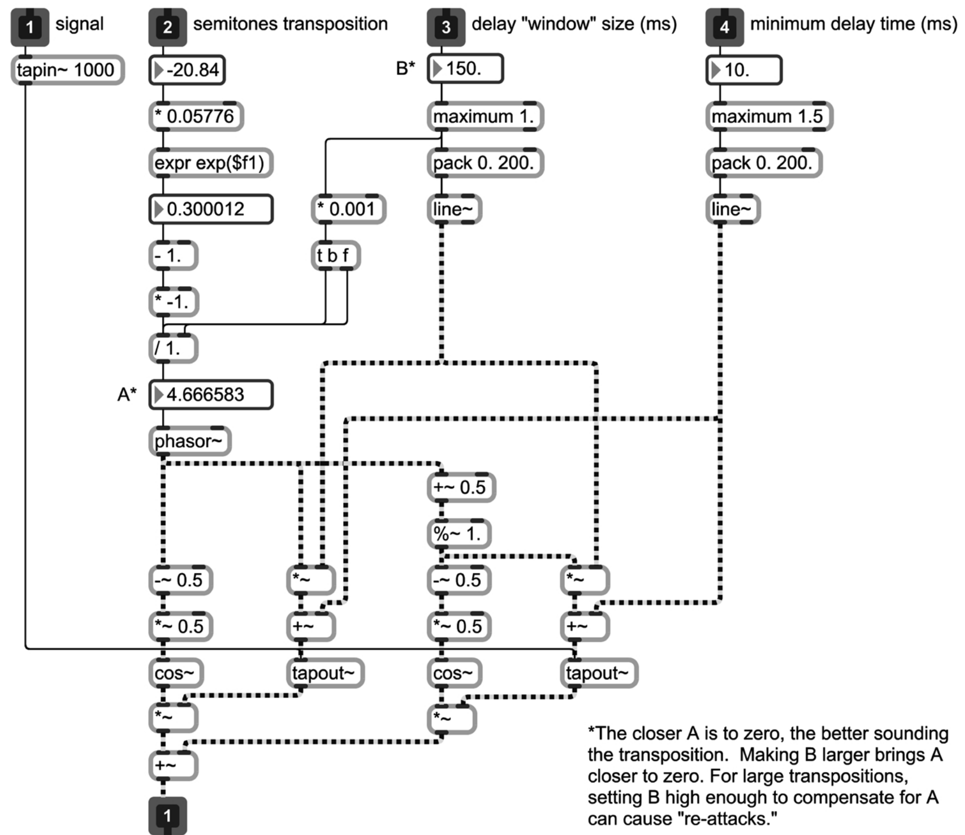

Figure B.42: Pitch shifter (p ps_by_delay), adapted from Puckette's G09.pitchshift.pd

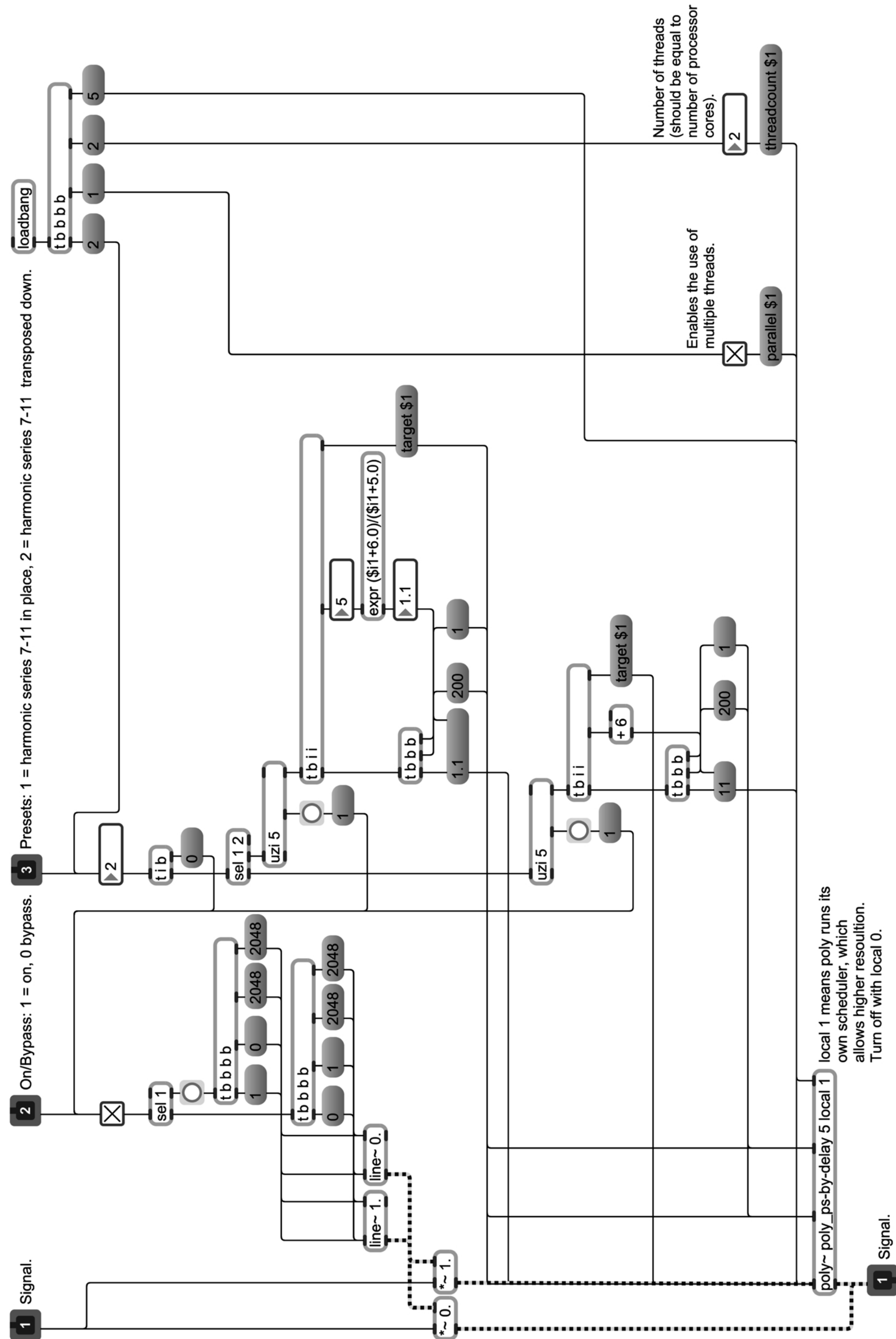

Figure B.43: Chorus settings (p chorus)

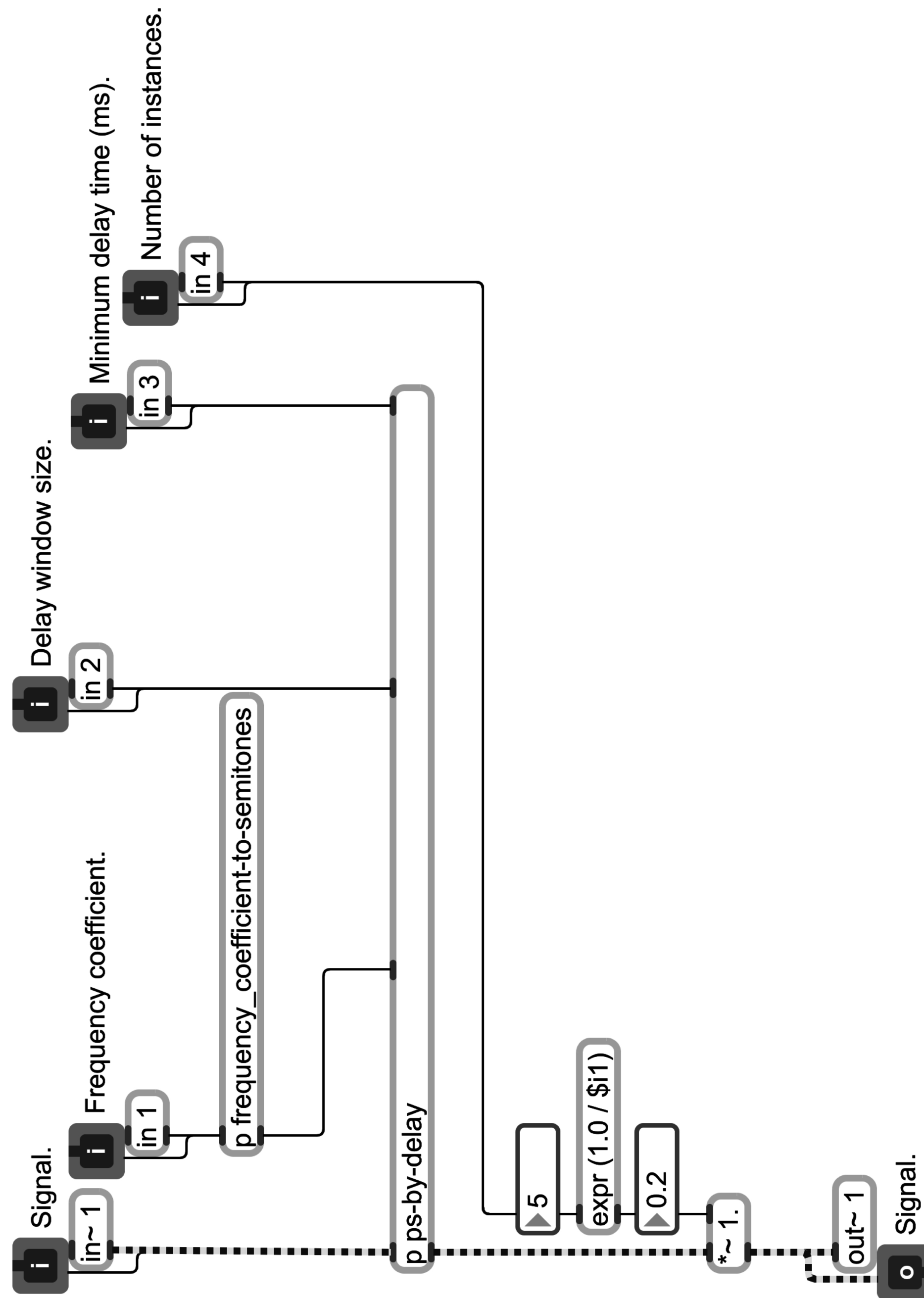

Figure B.44: Grouped pitch shifters (poly~ poly_ps-by-delay)

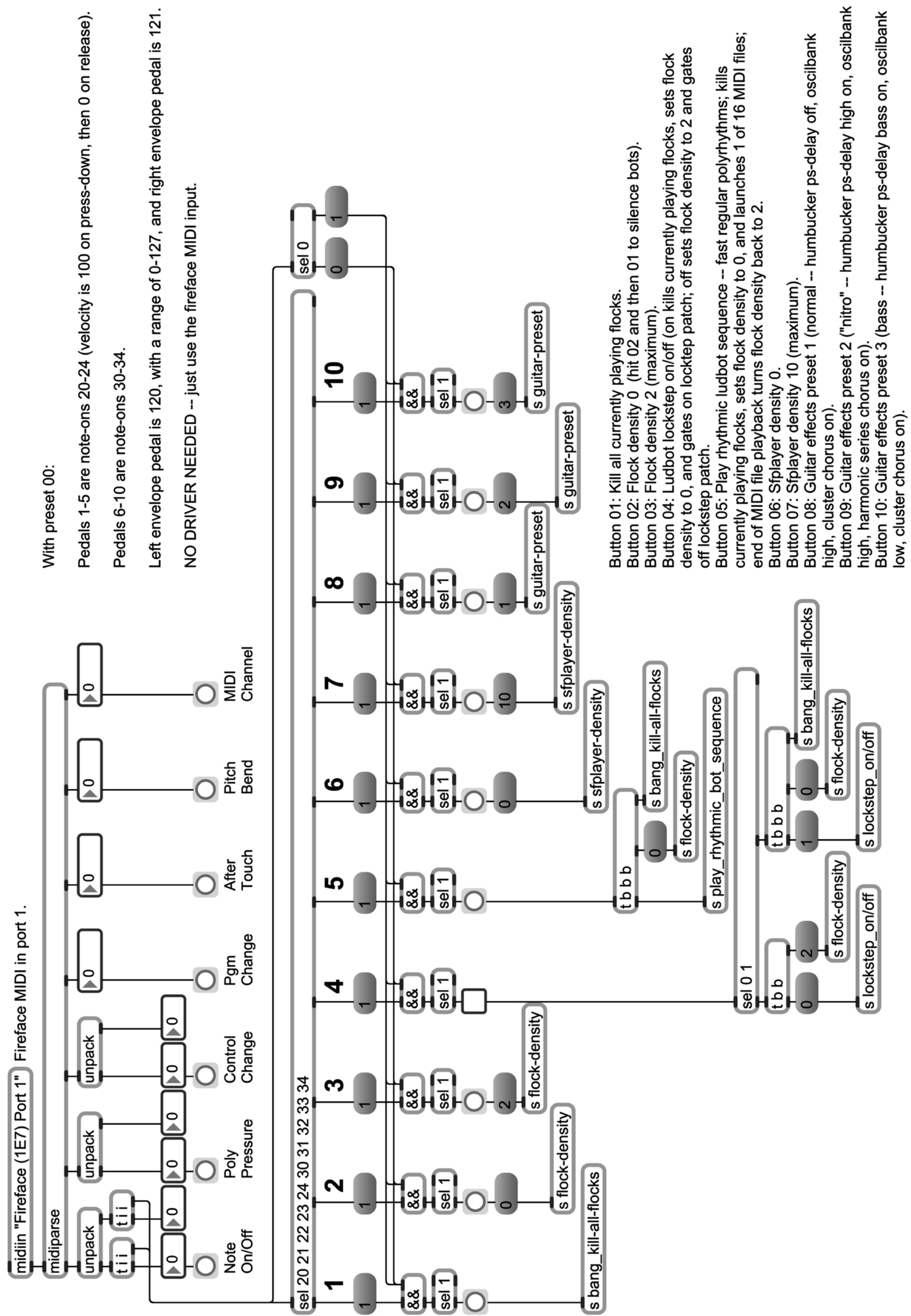

Figure B.45: MIDI footswitch controller mappings (p footswitch)

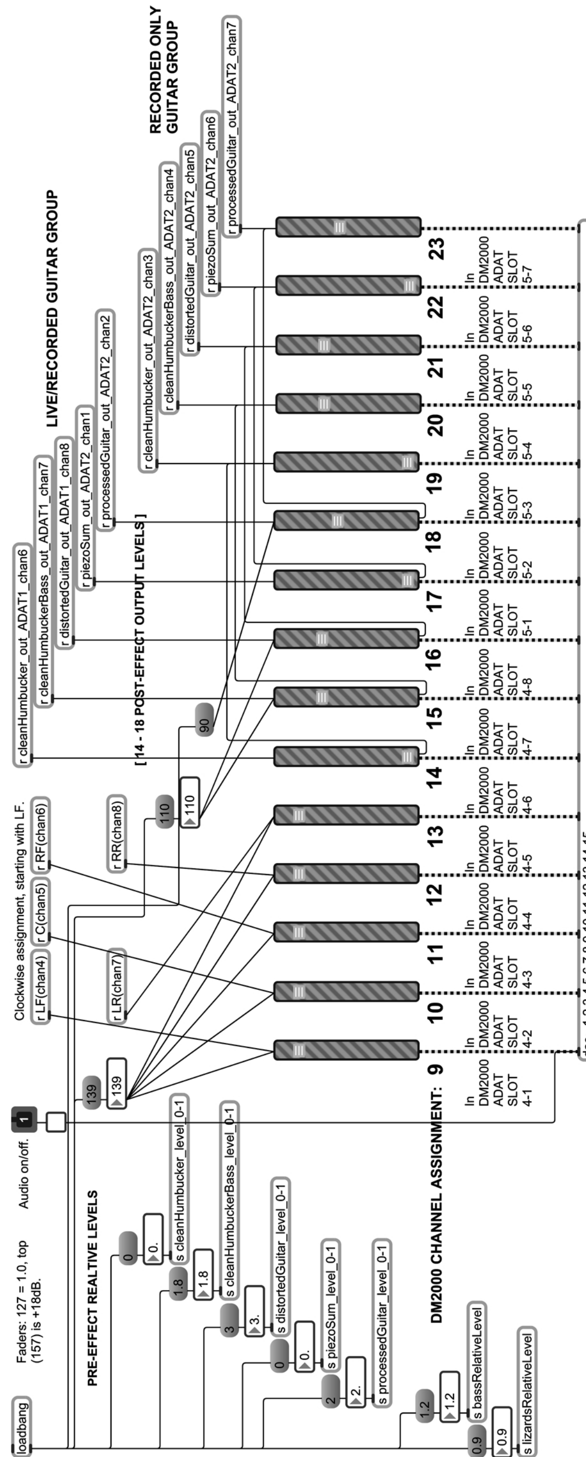

Figure B.46: Mixer (p mixer)

C CL-SYS

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;******************************************************************************
;; Copyright (C) 2007-9 Adam James Wilson (ajwilson@ucsd.edu). This program is
;; free software; you can redistribute it and/or modify it under the terms of
;; the GNU General Public License as published by the Free Software Foundation;
;; either version 2 of the License, or (at your option) any later version. This
;; program is distributed in the hope that it will be useful, but WITHOUT ANY
;; WARRANTY - without even the implied warranty of MERCHANTABILITY or FITNESS
;; FOR A PARTICULAR PURPOSE. See the GNU General Public License for more
;; details.
;;******************************************************************************
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; CONSTANTS
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar ppp 0.125)
(defvar pp  0.25)
(defvar p   0.375)
(defvar mf  0.5)
(defvar f   0.7)
(defvar ff  0.9)
(defvar fff 1.0)

;; Set below and above to adjust amplitude relationships in trees.

(defun retrieve-dynamic (iteration)
    (cond
        ((=  iteration 0) ff)
        ((=  iteration 1) f)
        ((=  iteration 2) p)
        ((=  iteration 3) pp)
        ((=  iteration 4) pp)
        ((>= iteration 5) ppp)
```

```
    )
)

(defun dynamic-from-float (float)
    (cond
        ((< float pp)
         "ppp"
        )
        ((< float p)
         "pp"
        )
        ((< float mf)
         "p"
        )
        ((< float f)
         "mf"
        )
        ((< float ff)
         "f"
        )
        ((< float fff)
         "ff"
        )
        ((>= float fff)
         "fff"
        )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; UTILITY FUNCTIONS: ROUND-TO-NEAREST-HUNDREDTH, COLLAPSE, SORT-LISTA-BY-LISTB,
;; GET-FIRST-INDEX-OF-VALUE, MEDIAN1
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun round-to-nearest-hundredth (float)
    (float (+ (floor float) (/ (round (* (remainder float (floor float)) 100))
        100)))
)

#|

(round-to-nearest-hundredth 54.344437)
(round-to-nearest-hundredth 54.995867)
(round-to-nearest-hundredth 54.23)

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun collapse (list index)
    (concatenate 'list
        (butlast list (length (nthcdr index list)))
        (nthcdr (+ index 1) list)
    )
)

#|

(collapse '(1 2 3 4 5) 2)

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun sort-lista-by-listb (lista listb &key (new-lista '()) (new-listb '()))
    (let ((lowest nil)
          (index-of-lowest nil)
         )
        (loop for b in listb with count = 0 do
            (cond
                ((not lowest)
                 (setf lowest b)
                 (setf index-of-lowest count)
                )
                ((< b lowest)
                 (setf lowest b)
                 (setf index-of-lowest count)
                )
            )
            (incf count)
        )
        (setf new-lista (append new-lista (list (nth index-of-lowest lista))))
        (setf new-listb (append new-listb (list lowest)))
        ;(print new-listb)
        ;(print new-lista)
        (if (> (length listb) 1)
            (sort-lista-by-listb
                (collapse lista index-of-lowest)
                (collapse listb index-of-lowest)
                :new-lista new-lista
                :new-listb new-listb
            )
            (list new-lista new-listb)
        )
    )
)

#|

(sort-lista-by-listb '(c a a b d e) '(4 2 2 3 5 9))
```

```
|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun sort-two-lists-by-lista (lista listb listc &key (new-lista '())
    (new-listb '()) (new-listc '()))
    (let ((lowest nil)
          (index-of-lowest nil)
         )
        (loop for a in lista with count = 0 do
            (cond
                ((not lowest)
                 (setf lowest a)
                 (setf index-of-lowest count)
                )
                ((< a lowest)
                 (setf lowest a)
                 (setf index-of-lowest count)
                )
            )
            (incf count)
        )
        (setf new-lista (append new-lista (list lowest)))
        (setf new-listb (append new-listb (list (nth index-of-lowest listb))))
        (setf new-listc (append new-listc (list (nth index-of-lowest listc))))
        ;(print new-lista)
        ;(print new-listb)
        ;(print new-listc)
        (if (> (length lista) 1)
            (sort-two-lists-by-lista
                (collapse lista index-of-lowest)
                (collapse listb index-of-lowest)
                (collapse listc index-of-lowest)
                :new-lista new-lista
                :new-listb new-listb
                :new-listc new-listc
            )
            (list new-lista new-listb new-listc)
        )
    )
)

#|

;; The first example below should produce an error.
(sort-two-lists-by-lista '(c a a b d e) '(4 2 2 3 5 9) '(2 0 0 1 3 4))
(sort-two-lists-by-lista '(4 2 2 3 5 9) '(c a a b d e) '(2 0 0 1 3 4))

|#
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-first-index-of-value (list value)
    (loop with index = 0 do
        (when (= (nth index list) value)
            (return index)
        )
        (incf index)
    )
)

#|

(get-first-index-of-value '(1 2 3 4 4 4 5) 4)

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun median1 (list)
    (let* ((vals (sort list #'< ))
           (len (length vals))
          )
        (if (oddp len)
            (elt vals (ash len -1))
            (let ((p (nthcdr (1- (ash len -1)) vals)))
                (/ (+ (car p) (cadr p)) 2)
            )
        )
    )
)

#|

(median1 '(1 2 3 4))

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; GET-TREE-RESOLUTION
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Divides length of the tree by the smallest time interval between onsets to
;; get the "resolution" of the tree; takes ceiling of division, because you want
;; any new resolution to be equal to or greater than inherent resolution.

;; Note: you can't currently count time-intervals of zero (or less than that) as
;; "smallest-interval."
```

```
(defun get-tree-resolution (onsets)
    (let ((smallest-interval nil)
          (total-duration (- (car (last onsets)) (car onsets)))
          (test-interval nil)
         )
        (loop with x = 0 with y = 1 until (= y (length onsets)) do
            (setf test-interval (- (nth y onsets) (nth x onsets)))
            (if smallest-interval
                (when (and (< test-interval smallest-interval)
                    (> test-interval 0))
                    (setf smallest-interval test-interval)
                )
                (setf smallest-interval test-interval)
            )
        (print
            (format nil "GET-TREE-RESOLUTION: Current smallest time interval =
                ~A" smallest-interval)
        )
            (incf x)
            (incf y)
        )
        (ceiling total-duration smallest-interval)
    )
)

#|

(get-tree-resolution '(0 2 15 16 16.5 17))
(get-tree-resolution '(-2 1 2 15 17 20))
(get-tree-resolution '(0 1 2 15 17 20))

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; FREQUENCY-TO-MIDI-KEYNUM
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; MIDI keynum 0 (C-1), the lowest MIDI key number, is 8.175798 hz in STANDARD
;; 12-tet; MIDI keynum 127 (G9), the highest MIDI key number, is 12543.852 hz in
;; STANDARD 12-tet.

(defun frequency-to-MIDI-keynum (frequency)
    (if (or (< frequency 8.175798) (> frequency 12543.852))
        (error "FREQUENCY-TO-MIDI-KEYNUM: Input frequency must be between
            8.175798hz and 12543.852hz, inclusive.")
        (round-to-nearest-hundredth (log (/ frequency 8.175798) (expt 2 1/12)))
    )
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; ANGLE-TO-ET-MIDI-KEYNUM
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; NOTE  this assumes that the pitch bend range of the target device is one
;; semitone up and one semitone down. In Tassman, for example, set Polyvkeys
;; pitch wheel amount to .083 (1/12 of a volt, where 1 volt equals an octave).

(defun angle-to-et-MIDI-keynum

    (fundamental        ;; Frequency in hertz from which pitch deviation by
                        ;; ANGLE occurs.
     angle              ;; Amount of pitch deviation from FUNDAMENTAL.
     equal-temperament  ;; Denominator for the exponent to the frequency
                        ;; coefficient represented by TEMPERAMENT-BASE.
     temperament-base   ;; The upper-limit coefficient of FUNDAMENTAL, divided
                        ;; into increments equal to TEMPARAMENT-BASE^(1/EQUAL
                        ;; TEMPERAMENT).
     &key
     (debugging? nil)
    )

    (let* ((fundamental fundamental)
           (angle angle)
           (equal-temperament equal-temperament)
           (temperament-base temperament-base)
           (dir nil)
           (coeff nil)
           (freq nil)
           (keynum nil)
           (keynum-fraction nil)
           (keynum-MIDI-fraction nil)
          )

        ;; Debugging.
        (when debugging?
            (print
                (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Input angle = ~A" angle)
            )
        )

        ;; Make sure ANGLE lies between -180 and 180 degrees, inclusive.

        (when (> angle  360) (setf angle (mod angle  360)))
        (when (< angle -360) (setf angle (mod angle -360)))
        (when (> angle  180) (setf angle (- angle 360)))
        (when (< angle -180) (setf angle (+ 360 angle)))

        ;; Debugging.
```

```
(when debugging?
    (print
       (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Angle within +/- 180
           degrees = ~A" angle)
    )
)

;; If ANGLE is negative, pitch change is downward; if positive, pitch
;; change is upward.

(if (> angle 0) (setf dir 1) (setf dir -1))
(setf angle (abs angle))

;; Rescale ANGLE to frequency range and quantize frequency to a step
;; in EQUAL-TEMPERAMENT.

(setf coeff (expt temperament-base (/ (round (* (/ angle 180)
    equal-temperament)) equal-temperament)))

;; If direction of pitch change is upward, multiply FUNDAMENTAL by the
;; pitch change coefficient; if the direction is downward, multiply by
;; the inverse of the pitch change coefficient.

(if (> dir 0) (setf freq (* fundamental coeff))
    (setf freq (* fundamental (/ 1 coeff))))

;; Debugging.
(when debugging?
    (print (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Fundamental frequency
        = ~A" fundamental))
    (print (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Fundamental frequency
        coefficient = ~A" coeff))
    (print (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Frequency deviation
        from fundamental, based on angle = ~A" freq))
)

;; Turn logarithmic frequency into a linear floating-point MIDI key
;; number.

(setf keynum (frequency-to-MIDI-keynum freq))

;; Debugging.
(when debugging?
    (print (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Initial keynum = ~A"
        keynum)
    )
)

;; Separate the fractional part of the key number from the key number.
;; Numbers below are rationalized (put into fractions) before arithmetic
;; operations so that there is no floating point error in the
```

```
        ;; calculation of pitch bends.

        (setf keynum-fraction (float (- (rationalize keynum) (rationalize (floor
            keynum)))))

        ;; If a smaller pitch bend is possible by coming DOWN from the key
        ;; number one greater than the current key number, add one to the key
        ;; number, and subtract one from the fractional part to get the
        ;; appropriate negative or downward pitch change from the new key
        ;; number.

        (when (> keynum-fraction 0.5)
            (setf keynum (+ (floor keynum) 1))
            (setf keynum-fraction (float
            (- (rationalize keynum-fraction) 1)))
        )

        ;; Rescale the fractional change to a value in the MIDI pitch-bend
        ;; range.

        (setf keynum-MIDI-fraction (round (cm:rescale keynum-fraction -1.0 1.0
            -8192 8192)))

        ;; Debugging.
        (when debugging?
            (print (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Keynum w/smallest
                required pitch-bend = ~A" keynum))
            (print (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Keynum fraction
                (- .99 to .99) = ~A" keynum-fraction))
            (print (format nil "ANGLE-TO-ET-MIDI-KEYNUM: Keynum MIDI
                fraction (-8192 to 8191) = ~A~%" keynum-MIDI-fraction))
         )

        ;; Return a list containing the key number, the key number's fractional
        ;; part, and the key number's fractional part rescaled to the MIDI
        ;; pitch-bend range of values.

        (list (floor keynum) keynum-fraction keynum-MIDI-fraction)

    )
)

#|

(angle-to-et-midi-keynum 440.00974 0 12 2 :debugging? t)
(angle-to-et-midi-keynum 440 0 12 2 :debugging? t)
(angle-to-et-midi-keynum 440 180 12 2 :debugging? t)
(angle-to-et-midi-keynum 440 -30 12 2 :debugging? t)
(angle-to-et-midi-keynum 440 -390 12 2 :debugging? t)
(angle-to-et-midi-keynum 440.43 390. 12 2 :debugging? t)
(angle-to-et-midi-keynum 260 30 12 2 :debugging? t)
```

```
(angle-to-et-midi-keynum 260 -30 12 2 :debugging? t)

;; Note: equal temperaments here are constructed based on FUNDAMENTAL, so 12-tet
;; may be slightly different from standard 12-TET based on C0 = 8.175798hz.

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; EXPAND-ANGLES-AND-NODES
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun expand-angles-and-nodes
    (angles
     nodes
     node-scalar
     iterations
     &key
     (debugging? nil)

     ;; Keyword arguments below "debugging?" are not to be overridden except by
     ;; the function itself.

     (original-angles angles)
     (original-nodes nodes)
     (cumulative-list-of-angles angles)
     (cumulative-list-of-nodes nodes)
     (cumulative-list-of-associated-iterations '())
     (iteration-count 0)
    )

    ;; Some error checking:

    (when (not (= (length angles) (length nodes)))
        (error "EXPAND-ANGLES-AND-NODES: You must input an equal number of
            angles and nodes.")
    )

    (let ((new-angles '())
          (new-nodes '())
         )

        ;; Debugging.
        (when debugging? (print (format nil "Iteration = ~A~%"
            iteration-count)))

        (cond
            ((> iterations 0)
             (loop for w in angles for x in nodes do
```

```
    (setf cumulative-list-of-associated-iterations
        (append cumulative-list-of-associated-iterations
            (list iteration-count)))

   ;; Debugging.
   (when debugging?
       (print (format nil "Angle = ~A" w))
       (print (format nil "Node == ~A~%" x))
   )

    (loop for y in original-angles for z in original-nodes do
        (setf new-angles
            (append new-angles
                (list
                    (+ w y)
                )
            )
        )
        (setf new-nodes
            (append new-nodes
                (list

                    ;; The below works for angles greater than 90
                    ;; degrees because, for example,
                    ;; (0.5 - 0.4cos(180 - 95)) =
                    ;; (0.5 + 0.4cos(95)).

                    (+ x (* (* z node-scalar) (cos (/ (* w pi)
                        180))))
                )
            )
        )
    )
)
(setf cumulative-list-of-angles (append cumulative-list-of-angles
    new-angles))
(setf cumulative-list-of-nodes (append cumulative-list-of-nodes
    new-nodes))
(expand-angles-and-nodes
    new-angles
    new-nodes
    (* node-scalar node-scalar)
    (- iterations 1)
    :debugging? debugging?
    :original-angles original-angles
    :original-nodes original-nodes
    :cumulative-list-of-angles cumulative-list-of-angles
    :cumulative-list-of-nodes cumulative-list-of-nodes
    :cumulative-list-of-associated-iterations
        cumulative-list-of-associated-iterations
    :iteration-count (+ iteration-count 1)
```

```
             )
            )
            ((= iterations 0)

             (loop for x in angles for y in nodes do
                 (setf cumulative-list-of-associated-iterations
                     (append cumulative-list-of-associated-iterations
                         (list iteration-count)))

                 ;; Debugging.
                 (when debugging?
                     (print (format nil "Angle = ~A" x))
                     (print (format nil "Node == ~A~%" y))
                 )

             )

             (list cumulative-list-of-angles cumulative-list-of-nodes
                 cumulative-list-of-associated-iterations)
            )
        )
    )
)

#|

(expand-angles-and-nodes '(30 -40) '(.5 .9) 0.8 2 :debugging? t)
(expand-angles-and-nodes '(95 -40) '(.5 .9) 0.8 2 :debugging? t)

;; Note: it is possible to have negative time values, as some chains of nodes
;; angles will put new nodes below zero; output must be rescaled to a range of
;; positive values.

;; Also, it is necessary to re-shuffle nodes and angles so that the times are in
;; order from lowest to highest.

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; QUANTIZE-TO-AVERAGE-DUR
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun quantize-to-average-dur (onsets amount-of-quantization)
    (when (or (<= amount-of-quantization 0) (> amount-of-quantization 1))
        (error "QUANTIZE-TO-AVERAGE-DUR: amount-of-quantization must satisfy
            (0 < aoq <= 1).")
    )
    (let* ((onsets (sort onsets #'<)) ;; Sort onsets low to high.
            (total-time (- (car (last onsets)) (car onsets)))
```

```
            (avg (/ total-time (- (length onsets) 1)))
        )
        (append
            (list (car onsets))
            (loop with x = 1 until (= x (- (length onsets) 1)) collect
                (+ (nth x onsets) (* amount-of-quantization
                    (- (+ (* x avg) (car onsets)) (nth x onsets)))) do
                (incf x)
            )
            (last onsets)
        )
    )
)

#|

(quantize-to-average-dur '(0 2 5 7 8 10) 1)
(quantize-to-average-dur '(-2 0 2 5 7 8) 1)
(quantize-to-average-dur '(0 2 5 7 8 10) .8)
(quantize-to-average-dur '(1 2 5 7 8 11) 1)
(quantize-to-average-dur '(1 2 5 7 8 11) .8)
(quantize-to-average-dur '(0 2 5 7 8 11) 1)
(quantize-to-average-dur '(0 2 5 7 8 11) 1.2) ;; Should produce an error.
(quantize-to-average-dur '(0 2 5 7 8 11) 0)   ;; Should produce an error.

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; QUANTIZE-ONSET-TO-EQL-DIVS
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Quantizes onset within a given duration (length) to the nearest time-point
;; out of the set of time points that equally divide the duration into
;; divisions-of length parts. IT WILL QUANTIZE POINTS "OUT OF RANGE" (0 -
;; length), but this is not desirable for make-tree-1; a function calling this
;; iteratively over a list of onsets should first make sure the list is pushed
;; into the range of positive numbers.

(defun quantize-onset-to-eql-divs (onset length divisions-of-length)
    (let ((smallest-interval (/ length divisions-of-length)))
        (* (round (/ onset smallest-interval)) smallest-interval)
    )
)

#|

(quantize-onset-to-eql-divs 3.5 20 10) ;; 4
(quantize-onset-to-eql-divs -1.5 20 10) ;; -2, but -2 is out of range (0 - 10)
(quantize-onset-to-eql-divs -2 20 10) ;; -2, but -2 is out of range (0 -10)
```

```
(float (quantize-onset-to-eql-divs 3.6 10 20)) ;; 3.5
(float (quantize-onset-to-eql-divs 20 10 20)) ;; 20; out of range (0 - 10)
(quantize-onset-to-eql-divs 0 10 25) ;; 0

|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; MAKE-TREE-1
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-tree-1

    (&key
     trunk-frequency              ;; "Trunk" frequency.
     angles                       ;; Initial changes in pitch for branches;
                                  ;; positive (up) or negative (down) within
                                  ;; 180 degrees.
     nodes                        ;; Where along the length of the trunk each
                                  ;; of the branches attach (out of length ==
                                  ;; 1.0).
     node-scalar                  ;; Changes location of nodes on each branch
                                  ;; by multiplying original nodes by (node-
                                  ;; scalar)^iteration.
     equal-temperament            ;; Number of equal divisions of
                                  ;; temperament-base.
     temperament-base             ;; Interval divided into equal steps by
                                  ;; equal temperament.
     iterations                   ;; Number of levels of repetition of
                                  ;; branching structure.
     (channelize? nil)            ;; Do you want to cycle through available
                                  ;; channels for each note? If so, how many
                                  ;; channels?
     (quantize-durs-to-avg? nil)  ;; Equalize spacing between onsets by
                                  ;; specifying amount of quantization to
                                  ;; average spacing (0 < aoq <= 1).
     (new-divisions? nil)         ;; A number of divisions of the tree
                                  ;; duration to quantize onsets to.
     (override-res-test? nil)     ;; New time resolution (set by new
                                  ;; divisions?) must equal or exceed current
                                  ;; res. unless override-res-test? == true.
     (new-length? nil)            ;; Specify a new total duration in seconds
                                  ;; for the complete tree.
     (debugging? nil)             ;; Display debugging information.
     (tree-start 0.0)             ;; You may need to set the "tree-start
                                  ;; note-on delay for MIDI sequencers that
                                  ;; cant negotiate simultaneous pitch-
                                  ;; bend and note-on events.
     (outfile "test.midi")        ;; Path to MIDI output file.
```

```
)

(let* ((angles-nodes-levels (expand-angles-and-nodes angles nodes
           node-scalar iterations :debugging? debugging?))

       ;; Add trunk angle deviation (always zero) to angles prior to
       ;; sorting.

       (angles (push 0 (car angles-nodes-levels)))

       ;; Add trunk start time (always zero) to nodes prior to sorting.

       (nodes (push 0 (cadr angles-nodes-levels)))

       ;; Add trunk iteration level (always zero) to iterations prior to
       ;; sorting.

       (levels (push 0 (caddr angles-nodes-levels)))

       ;; Sort angles and nodes in order of NODES from lowest to highest
       ;; (there may be some negative nodes since angles and branch
       ;; lengths may produce new nodes BEFORE trunk frequency onset at
       ;; time = 0.0).

       (sorted-nodes-angles-levels (sort-two-lists-by-lista nodes angles
           levels)) ;; Returns sorted lists in the order received.
       (sorted-angles (nth 1 sorted-nodes-angles-levels))
       (sorted-nodes (nth 0 sorted-nodes-angles-levels))
       (sorted-levels (nth 2 sorted-nodes-angles-levels))

       (bends '())
       (bend-ons '())
       (bend-offs '())

       ;; Turn angles into two lists: keynums and associated pitch bends.

       (keynums
           (loop for x in sorted-angles with midi-info = nil do
               (setf midi-info (angle-to-et-MIDI-keynum trunk-frequency
           x equal-temperament temperament-base :debugging?
           debugging?))
               (setf bends (append bends (list (nth 2 midi-info))))
               collect (nth 0 midi-info)
           )
       )

       ;; Turn iteration levels into amplitudes based on association
       ;; list.

       (amplitudes (loop for x in sorted-levels collect
           (retrieve-dynamic x)))
```

```
;; Shift note onsets into the positive range.
(onsets
    (if (< (car sorted-nodes) 0)
           (loop for x in sorted-nodes collect
               (+ x (abs (car sorted-nodes)) tree-start)
           )
           (loop for x in sorted-nodes collect
               (+ x tree-start)
           )
    )
)

;; quantize-durs-to-avg?

(onsets (if quantize-durs-to-avg? (quantize-to-average-dur onsets
    quantize-durs-to-avg?) onsets))
(tree-dur (car (last onsets)))
(tree-resolution (get-tree-resolution onsets))

;; Scale tree onsets to new divisions of tree dur (must check
;; first to make sure resolution is adequate); check is done
;; unless you want to override the resolution check and are
;; willing to allow some harmonically presented intervals.

(onsets
    (if new-divisions?
        (if (and (< new-divisions? tree-resolution)
    (not override-res-test?))
            (error (format nil "New divisions should be at
        least ~A." tree-resolution))
            (loop for x in onsets collect
        (quantize-onset-to-eql-divs x tree-dur
            new-divisions?))
        )
        onsets
    )
)

;; Scale onsets to new-length?; a new-length? longer than tree-dur
;; will stretch distance between attacks, a shorter length will
;; shrink them.

(onsets (if new-length? (loop for x in onsets collect
    (* x (/ new-length? tree-dur))) onsets))

;; Make durations 1/2 the amount of time between consecutive
;; onsets; make bend-ons start 1/30000th of a second before note
;; onsets and bend-offs stop 1/30000th of a second before the
;; onset of the next note. [Maybe quantize these 1/30000 intervals
;; to 32-bit floats for older sequencers; they are doubles now.]
```

```
(durations
    (if channelize?
        (loop with x = 0 with y = 1 with z = channelize? until
            (= x (length onsets)) collect
            (if (nth z onsets)
                (/ (- (nth z onsets) (nth x onsets)) 2)
                20.0 ;; Make final duration quite long.
            )
            do
            (when (nth y onsets)
                (setf bend-ons
                    (append bend-ons
                        (list
                            (- (nth y onsets)
                    1/30000)
                        )
                    )
                )
            )
            (if (nth z onsets)
                (setf bend-offs
                    (append bend-offs
                        (list
                            (- (nth z onsets)
                    1/30000)
                        )
                    )
                )

             ;; No bend-off required for last note, but you need to
             ;; add it so lists being looped over in parallel are
             ;; equal in length.

                (setf bend-offs
                    (append bend-offs
                        (list
                            (+ (nth x onsets) 1.0 1/30000)
                        )
                    )
                )
            )
            (incf x)
            (incf y)
            (incf z)
        )
        (loop with x = 0 with y = 1 until (= x (length onsets))
            collect
            (if (nth y onsets)
                (/ (- (nth y onsets) (nth x onsets)) 2)
                20.0 ;; Make final duration quite long.
```

```
               )
               do
               (if (nth y onsets)
                   (setf bend-offs
                       (append bend-offs
                           (list
                               (- (nth y onsets) 1/30000)
                           )
                       )
                   )

                ;; No bend-off required for last note, but you need to
                ;; add it so lists being looped over in parallel are
                ;; equal in length.

                   (setf bend-offs
                       (append bend-offs
                           (list
                               (+ (nth x onsets) 1.0 1/30000)
                           )
                       )
                   )
               )
               (incf x)
               (incf y)
        )
    )
   )

   (MIDI-data '())

  ) ;; matches let* argument list

(if channelize?
    (setf bend-ons (append '(0) bend-ons))
    (setf bend-ons (append '(0) bend-offs))
)

;; Debugging.
(when debugging?
    (print (format nil "MAKE-TREE-1: unsorted angles:~%~A~%"
        angles))
    (print (format nil "MAKE-TREE-1: unsorted nodes:~%~A~%" nodes))
    (print (format nil "MAKE-TREE-1: sorted angles:~%~A~%"
        sorted-angles))
    (print (format nil "MAKE-TREE-1: sorted nodes:~%~A~%"
        sorted-nodes))
    (print (format nil "MAKE-TREE-1: bends-ons:~%~A~%" bend-ons))
    (print (format nil "MAKE-TREE-1: note-ons:~%~A~%" onsets))
    (print (format nil "MAKE-TREE-1: amplitudes:~%~A~%"
        amplitudes))
```

```
    (print (format nil "MAKE-TREE-1: keynums:~%~A~%" keynums))
    (print (format nil "MAKE-TREE-1: bends:~%~A~%" bends))
    (print (format nil "MAKE-TREE-1: durations:~%~A~%" durations))
    (print (format nil "MAKE-TREE-1: bend-offs:~%~A~%" bend-offs))
)

(loop
    for on in onsets
    for dur in durations
    for key in keynums
    for bend in bends
    for b-on in bend-ons
    for b-off in bend-offs
    for amp in amplitudes
    with chan = 0 do

    ;; Debugging.
    (when debugging?
        (print (format nil "MAKE-TREE-1: channel = ~A" chan))
        (print (format nil "MAKE-TREE-1: bend on = ~A" b-on))
        (print (format nil "MAKE-TREE-1: note on = ~A" on))
        (print (format nil "MAKE-TREE-1: amplitude = ~A" amp))
        (print (format nil "MAKE-TREE-1: keynum = ~A" key))
        (print (format nil "MAKE-TREE-1: bend = ~A" bend))
        (print (format nil "MAKE-TREE-1: duration = ~A" dur))
        (print (format nil "MAKE-TREE-1: bend-off = ~A~%" b-off))
    )

    (setf MIDI-data
        (append MIDI-data
            (list
                ;;bend on
                (new midi-pitch-bend
                    :time b-on
                    :channel chan
                    :bend bend
                )
                ;; note
                (new midi
                    :time on
                    :channel chan
                    :duration dur
                    :amplitude amp
                    :keynum key
                )
                ;; bend off
                (when b-off
                    (new midi-pitch-bend
                        :time b-off
                        :channel chan
                        :bend bend
```

```
                        )
                    )
                )
            )
        )

        (when channelize?
            (setf chan (mod (+ chan 1) channelize?))
        )
    )

    (events MIDI-data outfile :channel-tuning nil :divisions 30000)

    ) ;; matches let*

)

#|

(make-tree-1
    :trunk-frequency 285.00
    :angles '(30 -40 67)
    :nodes '(0.1 0.5 0.9)
    :node-scalar 0.8
    :equal-temperament 17
    :temperament-base 2.1
    :iterations 3
    :channelize? 4
    :new-length? 10
)

|#
```

D Duoquadragintapus DVD-R

The attached DVD-R contains video and audio files of the premiere performance of *The Duoquadragintapus*, which took place on May 19 in Atkinson Hall at the University of California, San Diego, as part of the UCSD Music Department 2009 Spring Festival of New Music.

Production credits:
Audio recording by Dustin Raphael and Adam James Wilson.
Audio editing by Adam James Wilson.
Lighting by Todd Margolis.
Video recording and editing by Alex Matthews.

Section titles below refer to the names of electronic files on the accompanying DVD-R.

D.1 Duoquadragintapus.mp4

MPEG 4 video.

D.2 Duoquadragintapus.wav

Stereo interleaved 44.1kHz/16-bit WAV file.

D.3 Duoquadragintapus.wav.L, Duoquadragintapus.wav.R

Separate left- and right-channel mono 96kHz/24-bit WAV files.

E Performance Program

Attached is a PDF facsimile of an excerpt from the program notes of the UCSD Music Department 2009 Spring Festival of New Music, during which the premiere performance of *The Duoquadragintapus* took place.

Bibliography

Allauzen, Cyril, Crochemore, Maxime, and Raffinot, Matheiu. "Factor Oracle: A New Structure for Pattern Matching." *Lecture Notes in Computer Science* 1725 (1999): 295–310.

Assayag, Gerard and Dubnov, Shlomo. "Using Factor Oracles for Machine Improvisation." *Soft Computing* 8 (2004): 1–7.

Gervasi, Vincenzo and Principe, Giuseppe. "Coordination without Communication: the Case of the Flocking Problem." *Discrete Applied Mathematics* 144 (2004): 324–344.

Krumhansl, Carol and Jusczyk, Peter W. "Infants' Perception of Phrase Structure in Music." *Psychological Science* 1 (1990).1: 70–73.

Puckette, Miller S. "Patch for Guitar." Tech. rep., Second International PureData Convention, Montreal, Canada, 2007.
URL `http://crca.ucsd.edu/~msp/Publications/pd07-reprint.pdf`

Wilson, Adam James. "Flocking in the Time-Dissonance Plane." *Proceedings of the International Computer Music Conference*. Montreal, Canada, 2009a, 387–390.

———. "A Symbolic Sonification of L-Systems." *Proceedings of the International Computer Music Conference*. Montreal, Canada, 2009b, 203–206.