Who is Listening?

Human and Computer Influence in Interactive Music Systems

By Gregory M. Rippin

Submitted in partial fulfillment of the requirements for the

Master of Music in Music Technology

in the Department of Music and Performing Arts Professions

in the Steinhardt School of Education

New York University

April 27, 2003

Table of Contents

1. Introduction

PD and MIDI

This paper is about interactive music systems. It will attempt to explain what interactive music systems are, and what it is about these systems that makes them useful musical tools. A limited understanding of some of the technical issues that underlie the development of such systems is necessary for an intelligent discussion of their properties to take place. Specifically, it is assumed that the reader has a working understanding of MIDI, and has had some experience with PD or similar programming environment.

MIDI is the Musical Instrument Digital Interface, a protocol from the 1980's that grew out of the desire to have synthesizers, keyboards, and computers manufactured by different companies be able to work together in a unified technical environment. MIDI's shortcomings as both a transmission protocol and as a means of musical representation have been discussed at length elsewhere,[1] and there is no need to repeat the discussion here. However, while many different schemes of transmission and representation have been developed, and many of them are in no doubt superior to MIDI, none of them have achieved the universality of MIDI. Despite its shortcomings, MIDI continues to be used by a large number of musicians and composers, and it will undoubtedly continue to be used for the foreseeable future, if for no other reason than that the large number of MIDI-based software tools that have been created over the twenty years of MIDI's existence will be difficult to replace. In most of the systems discussed in this paper, MIDI is used

1 See for example Robert Rowe's *Interactive Music Systems: machine listening and composing* (Cambridge, Mass.: MIT Press, 1993), 10, Rowe's *Machine Musicianship* (Cambridge, Mass.: MIT Press, 2001), 29, and Richard F Moore's "The Dysfunctions of MIDI." *Computer Music Journal* 12, No. 1 (Spring 1998), 19-28.

both as a means of internal representation and as the primary method of getting data to and from the outside world.

PD is a graphical programming environment specifically designed for the creation of interactive systems. It is intended for use by technically-informed composers, musicians, and artists; many of the low-level tasks (such as memory management) that can make programming a chore and a distraction from the artistic task at hand are fully automated in PD. PD uses "objects," which are boxes that "do stuff" on their own and can be connected to other objects to created complicated behavior; an object's name determines its behavior. The function of many types of objects is evident from their names (such as **+** and **integer**; the name of objects will be rendered with modified **Terminal font** throughout this paper). The roles of other objects are not so obvious and will be explained in the course of the text. An object of particular importance is **pd**, which creates a container subpatch that can be used to separate, organize, and hide groups of objects that perform a function together. A thorough explanation of PD can be found in PD's documentation and help files, which are contained in the CD-ROM appendix to this paper.

PD is the ugly cousin of Max/MSP, and much of the two environments (including many types of objects) is similar or identical. PD has several important advantages over Max, however, that bear mentioning. First, while Max runs only on Macintosh computers (and currently only on OS 9.x, not OS X), PD runs on most of the major operating systems in use today, including Windows 98, NT, and XP, Mac 9.x and X, several flavors of Linux, Unix, Palm, and IRIX. Patches (as PD programs are called) are fully portable between these operating systems. Secondly, PD contains an extensible data

structure that brings PD more closely in line with Object-Oriented Programming design philosophy allows it to be used for data-heavy "serious" programming, including graphics processing. Many of the external object libraries that can be loaded into PD to extend its capabilities make use of this data structure to create new data types; one, the *matrix* data type defined in the ZEXY library, is used extensively in this paper. Finally, while Max/MSP costs upward of several hundred dollars, PD is completely free, and is available for download from several locations on the internet. PD's name, which stands for both "Pure Data" and "Public Domain," is a direct reference to these advantages.

Interactive Music Systems

Interactive systems are, in the most obvious terms, systems that interact with the world around them. A "system" is an abstract entity that can be characterized as having a set of inputs, a set of outputs, and some discernable correlation between specific inputs and outputs. Often, these connections are referred to as "behaviors." A key point in the identity of systems is their ability to be isolated from their environment, so that these behaviors can be studied, learned, practiced, and so forth, by human users.

Systems vary immensely in their complexity. A very simple system consists of a battery, a light bulb, and a button switch. The behavior of this system is quite simple: when the switch is pushed (input), the light turns on (output). A piano is a much more complex example, since there are eighty-seven more buttons, as well as some additional controls that modify the output caused by buttons (the pedals). Despite the added complexity, a piano's input can still be mapped to its output. It is still, in theory, a closed system.

An even more complicated system consists of a computer connected to MIDI gear and an audio interface to facilitate its use in making music. Such a system still has a set of inputs (MIDI messages and audio signals, ASCII keyboard presses and mouse clicks) and outputs (MIDI messages and audio signals, and possible graphical output), as well as a large and complex, but still explicit and algorithmic, set of behaviors.

All systems are by definition "active;" that is, they have behaviors. The do things when other things are done to them. However, it is necessary to distinguish between systems that are "interactive" and ones that are "reactive." All systems are, to a limited extent, interactive, in that their behavior has some effect on the outside world, and that the world must make some adjustment as a result. This interaction generally occurs at a very low level—a violinist listens to the sounds being coming from his violin and makes tiny, unconscious adjustments to keep the instrument in tune. Similarly, a pianist listens to the instrument that he is playing and adjusts his touch to control the piano's tone and dynamics. It is this capacity that allows pianists to quickly adapt and play unfamiliar instruments, musicians to sight-read and learn new repertoire. However, once a musician is familiar with his instrument and material, this interaction is relegated to the land of "reactive" behavior—the system's behavior is entirely expected and does not cause the musician to make appreciable decisions or changes in behavior.

Traditional instruments therefore constitute what are essentially one-directional systems, since the information feedback from the instrument to the performer is minimal and low-level. Conventional MIDI performance setups, consisting of a MIDI controller such as a keyboard and a sound generator such as a synthesizer, are even more one-directional, since the amount of information that feeds back to the performer is even

smaller—the keys do not vibrate, the timbre does not change organically with changes in keyboard touch, and so forth.

As another example of a system with a one-directional informational flow, consider a piece of electronic music that has been recorded to tape and is being replayed. If you add a live performer who, for instance, plays a piano score in synch with the tape, you end up with a more interesting performance, but the system effectively hasn't changed. The tape player isn't effected by the presence of the performer, who is nonetheless listening to the tape and adjusting his playing (tempo, dynamics, etc.). Note that the presence of a computer doesn't necessarily make the system interactive, or even more complex. For example, a MIDI sequencer or digital audio workstation could easily be substituted for the tape player in the system described above, without any significant changes in complexity or information flow.

More complex systems do involve, however, the insertion of a computer at some point in the information path. As a very basic example of a system that is more complex, a computer can arpeggiate the notes of the chord being played by the performer. Immediately, the system becomes a more active part of the performance. The system is actively supplying musical material, and the performer must pay careful attention to what the system is doing in order to produce a musical result. The essence or interactive music systems, then, is that they, along with one or more human performers, cooperatively participate in a musical performance, in a way that makes both the human and the computer components active and essential. The similarity to human-to-human musical interaction is not accidental. In reality, "computers simulate interaction... by allowing users to change aspects of their current state and behavior. This interactive loop is

completed when the computers, in turn, affect the further actions of the users." [2] The design of interactive music systems, "assumes an implicit recursive element, namely a loop between the 'sound' and the 'performer': the computer output somehow affects the performer's next action (reaction), which in turn will eventually affect the computer system in some way." [3]

This is a common description of interactive music systems. Robert Rowe states that "interactive computer music systems are those systems whose behavior changes in response to musical input. Such responsiveness allows these systems to participate in live performances, of both notated and improvised music." [4] Note the emphasis on behavior, musicality, and responsiveness. An alternate definition is suggested by Winkler: "Interactive music is defined here as a music composition or improvisation where software interprets a live performance to affect music generated or modified by computers." [5] More specifically, interactive music systems "can be viewed as dedicated compositional tools capable of reacting in some way upon changes they detect in their 'external conditions,' namely in the initial input and the run-time control data. Such data are usually set and adjusted by some agency -- a performer, a group of performers (it could be a composer, too, either working in the studio of experimenting on stage, improvising) using some control device." [6]

All of these definitions of interactive music systems are correct, because many different types of interactive systems exist. Rowe suggests the use of three categorical

---

2 Todd Winkler's *Composing Interactive Music: Techniques and Ideas Using Max* (Cambridge, Mass.: MIT Press, 2001), 3

3 Agostino Di Scipio's "Sound is the Interface: Sketches of a Constructivistic Ecosystem View of Interactive Signal Processing." Proceedings of the Colloquium on Musical Informatics (Firenze 8-10 May 2003), 1.

4 Rowe's *Interactive Music Systems*, 1.

5 Todd Winkler's *Composing Interactive Music*, 4.

dimensions to classify systems. The "score-driven—performance-driven" dimension addresses the amount of improvisation that is expected of the performer and the computer. The "transformative-generative-sequenced" dimension examines the source of the musical material being used in a work, especially the material being used by the computer. Finally, the "instrument-player" dimension addresses the musical roles of the performer and the computer. [7]

Rowe makes an important point as to the purpose of such a classification: it is to recognize the similarities between systems and identify their common predecessors. [8] Such classification is, however, quite troublesome. Definitions that create categories like those above are approximate. When is a score following system more than a score follower? If it handles periods of improvisation and adapts its scored output to better fit the local improvisatory context, is it still to be considered a score follower? If an instrument system spontaneously begins to play notes of its own from time to time, is it still an instrument system, or has it become something different?

This last example suggests a further problem with categorical classification: in the course of a piece, many systems shift roles and behaviors drastically. Such internal changes are often the focus of interactive compositions. "A central issue that confronts all composers of interactive music is the drama of human-computer interaction. What is the relationship between humans and computers?" [9] In fact, "it becomes clear that the system design itself, and particularly the actions mediated by the user interface (interdependencies among control variables), become the very matter of composition and

---

6 Di Scipio's "Sound is the Interface," 1.

7 Rowe's *Interactive Music Systems*, 6-7.

8 Ibid., 6.

can no longer be separated by the internal development of sound." [10] As "distinctions fade between instrument and music," composers "compose" the systems that make the music as much as they compose the music itself. [11] Indeed, interactive composers such as George Lewis begin to collectively refer to their software systems, compositional strategies, and musical output as "compositions." [12]

This development suggests an additional motivation for classifying interactive systems: it is necessary to do so to even understand many interactive works. Questions like those asked above by Winkler lead one to think that a parametric, rather than categorical, classification would be appropriate when one is trying to construct a taxonomy of interactive music works. What is the relationship between humans and computers? Who is making what decisions? And how, and with what certainty? What would happen if one stopped playing? Would the other stop, take a solo, or even notice that the other had fallen silent? Who is leading? Who is listening?

---

9 Todd Winkler's *Composing Interactive Music*, 21.

10 Di Scipio's "Sound is the Interface," 1.

11 Joel Chadabe's *Electric Sound : the past and promise of electronic music* (Upper Saddle River, NJ : Prentice Hall, 1997), 291.

12 George E. Lewis's "Too Many Notes: Computers, Complexity, and Culture in Voyager." (*Leonardo Music Journal* 10 (2000): 33.

2. Proposed Taxonomy of Interactive Music Systems

Who is Playing?

The first way we can examine the information flow in a system is to look at who is playing, a description that I shall call *Participation*. The term *Participation* shall be defined to include only those times when a computer or performer is actively producing output, and shall exclude the kind of activity that effects the output of another performer but does not result in any actual output of its own accord. A special case must be made for systems that involve a performer and a computer working collaboratively to produce sounds from a single source—what Rowe calls "instrument" systems. Since activity on the part of both the performer and the computer is necessary in order to produce any sound at all, it seems logical to say that their Participation is about equal.

Since the amount of Participation on the part of human and computer performers in a piece are largely uncorrelated—a computer doesn't necessarily start to make more sound when the human makes less—Participation should be pictured as a two-dimensional description, with human and computer performer participation varying independently. See **Figure 1**, where the amount of human Participation is depicted horizontally, and the amount of computer Participation is pictured vertically.

To cite the extremes of this description, take two solo performances, one by a human musician and another by a computer system. The first make take the form of a piano sonata, the second, a prerecorded score of electronic music being replayed. Using the graphical scheme introduced above, the piano sonata would be marked off at (1, 0), and the scored computer piece would be at (0,1). An example of a piece with nearly

equal amounts of human and computer Participation is Salvatore Martirano's "YahaSALmaMAC Orchestra" (1986). In this piece, portions of the performer's playing on a MIDI keyboard are captured by the computer, and are then processed, mutated, delayed, and then played back in canon with the performer's ongoing performance. [13] Using the graphical scheme, this piece would be roughly (0.75, 0.75) (see **Figure 2**), with the Participation of both the human and the computer dipping and peaking throughout the course of the piece.

Figure 1: Example Participation Diagram

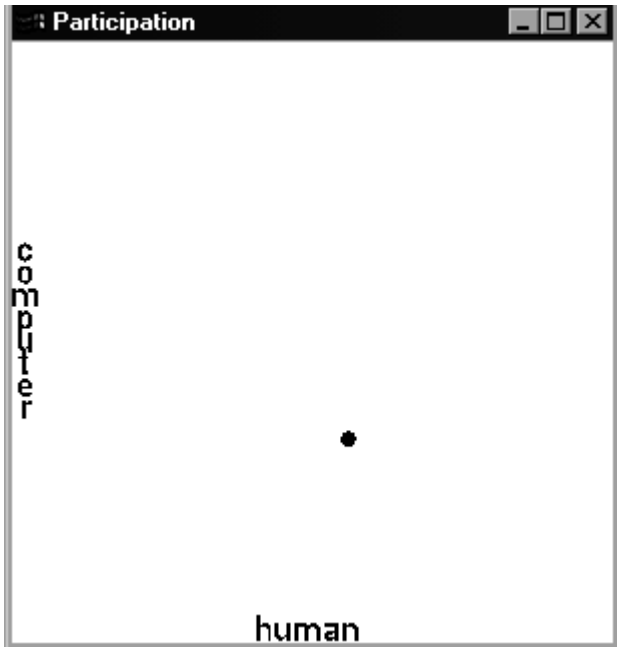
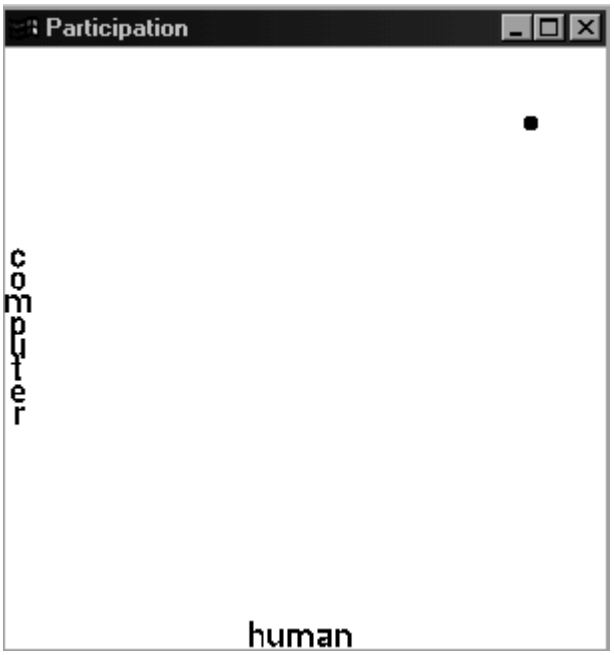


Figure 2: YahaSALmaMAC Participation



Admittedly, this is not at all a rigorous or scientific description, since Participation as I have defined it is difficult to quantify in a musically meaningful way. The Participation diagrams are based on authors' descriptions of their systems rather than on measurements of the systems' functioning. However, this description shall prove to

13 Chadabe's *Electric Sound*, 213-214.

be useful for comparing relative amounts of Participation, both between pieces and within individual pieces.

Who is listening?

One of the major challenges faced by early electronic music composers who wished to write music in the concert tradition was one of drama and presentation: how does one perform music that had been created on electronics in a way that is interesting to an audience? Of the many answers to this question, one of the most widespread was and continues to be the inclusion of a live musician performing onstage, on electronics or on a traditional instrument, alongside a previously realized taped electronic part. A Participation diagram places this piece again somewhere near (0.75, 0.75), with fluctuations throughout the piece.

The problems with this approach are immediate and obvious: the tape part and the live musician are essentially two separate systems operating in the same time and acoustical space. While deeper, musical connections between the two parts could be worked out in advance, the presence of any realtime connection between the parts is illusory. Real musical interaction does not occur. The performer's musical flexibility and spontaneity is greatly restricted. The restraints, especially the temporal ones, are rigid in such a setup.

As composers moved from working with analog electronics to working with computers, the problem persisted:

> Computer music compositions written for live performance often include performers playing traditional acoustic instruments. In the early days, the only viable method was to have the performer play along with a previously

> recorded tape of an electronically realized part. As computers became faster, some performers began to play with parts synthesized in real time. Either approach represents a one-way interaction in which performers' actions are influenced by what they hear from the electronic source.[14]

Dodge points out the pertinent and limiting feature of a system such as this: the flow of information is one-directional. The performer must react to the tape part, especially with relation to changes in tempo, sudden dynamic and tonal shifts, but the tape player (or realtime synthesis algorithm) chugs along, oblivious to the actions (or inaction) of the performer. The human is listening. The electronics are not.

A separate visualization is needed to describe what's being discussed here: cross-influence between the various actors in a performance, which I shall call simply *Influence*. Again, two dimensions are needed, since human and computer Influence will vary independently. The result is another two-dimensional graph, with computer→human Influence increasing to the right, and with human→computer Influence increasing upward on the graph. Once again, the diagrams are based on authors' descriptions of their systems rather than on measurements of the systems' functioning.

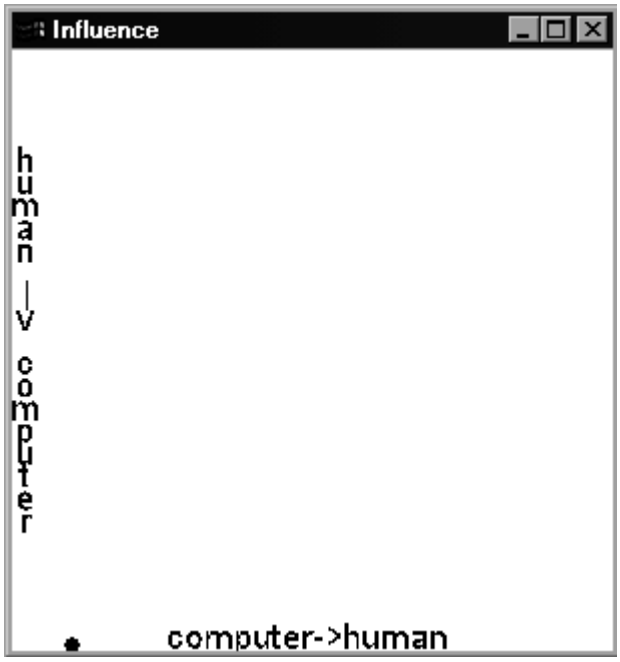
In the system being discussed here, the human is, at most, adjusting to expected changes in tempo and dynamics on the part of the computer, so the computer→human influence is very low. Meanwhile, nothing that the human does (short of pressing the stop button) will have any effect on the computer: the human→computer Influence is

---

14 Charles Dodge and Thomas A. Jerse's *Computer Music: synthesis, composition, and performance*, Second Edition. (United States: Wadsworth Publishing Company, 1997), 412.

zero. The Influence diagram of this system, pictured in **Figure 3**, would place it at around (0.1, 0).[15]

Figure 3: Example Influence Diagram



Take, by way of contrast, the work of George Lewis, whose *Voyager* system "has its own behavior, which is sometimes influenced by the performance of the humans."[16] Lewis's pieces are mostly improvisations, in which the humans and the computer play off of each other in various ways. "I conceive a performance of *Voyager* as multiple parallel streams of music generation, emanating from both the computers and the humans—a nonhierarchical, improvisational, subject-subject model of discourse." (Lewis "Too Many Notes" 34). The computer makes its own musical decisions, often based on analyses of the performer's playing, and "decisions taken by the computer have consequences for the music that must be taken into account by the improviser." [17] The large amount of Influence that the computer and human exert on each others' Participation, allows real, musical Interaction to occur. The Influence diagram in **Figure 4** places this piece at around (0.5, 0.5).

We therefore have two ways of describing interactive computer music systems, two graphs that depict, albeit roughly, information about who is making sound and who is

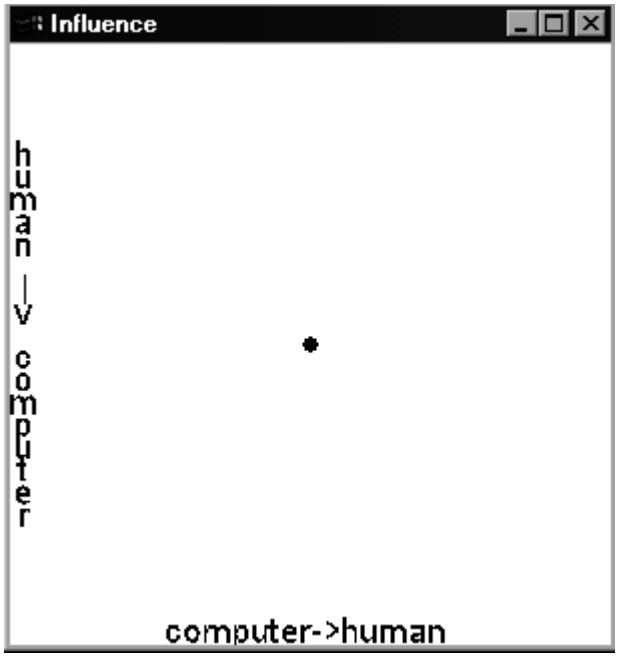
15 The Participation and Influence graphics are rendered using GEM, PD's Graphics Environment for Multimedia.

16 Rowe's *Interactive Music Systems*, 80.

17 George Lewis's "Too Many Notes," 26.

deciding when and how the sound is going to be made. These two descriptions vaguely describe the information flow in an interactive music system.

Figure 4: George Lewis Influence



3. Envisioning Influence: Time Synchronization

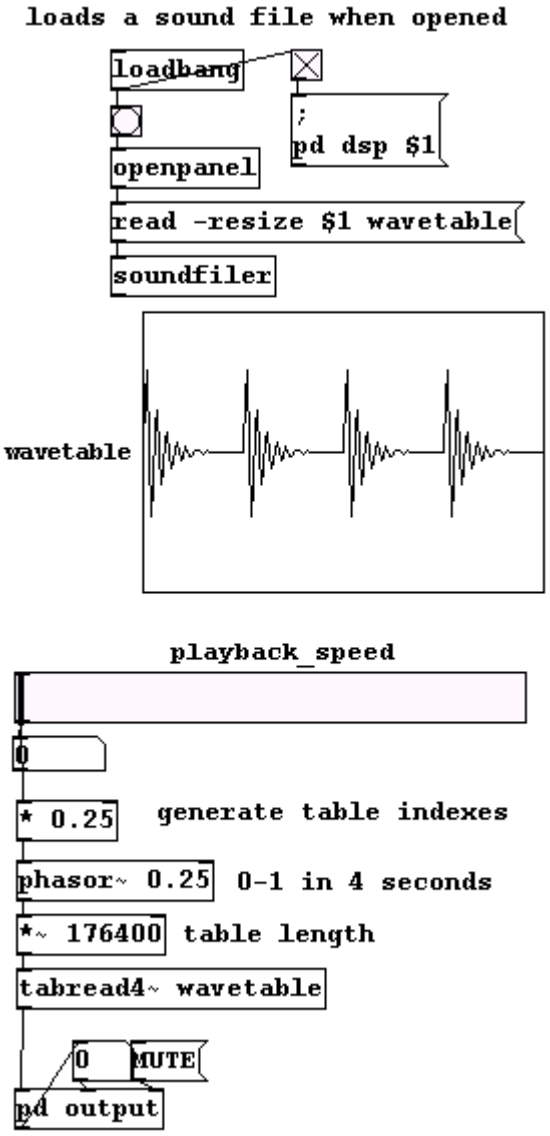
Tempo Matching

Composers of electronic music have often presented their work to a live concert audience in the form of a live performer playing along with a prerecorded electronic part. As was discussed, this arrangement places great restrictions on the performer, specifically in terms of temporal flexibility. An early approach to solving the problems of live performance with taped electronics involved manually varying the playback speed of the tape to better match the tempo of the performers. This method was used by Earle Brown in his 1963 piece "Times Five" in order to synchronize a tape with a small orchestra. [18] However, two major problems are quickly observed: first, from a purely technical standpoint, changing the speed of the tape playback effects not only tempo, but pitch as well, and this may not be a desired effect. Secondly, and more importantly for the discussion at hand, the lever that controls tape playback speed must be controlled by another human, another performer. The problem hasn't really been solved, merely sidestepped, for the electronics still aren't listening.

We shall put aside the deeper issue for a moment and look at the more technical problem of the relationship between changes in playback speed and changes in pitch. Such simple "time warping" effects can be mimicked in the digital domain (with the same pitch shifting side-effects) by altering the playback speed of a sample contained in a wavetable. In **Figure 5**, the sample player **tabread4~** receives a "playback rate index" from the horizontal fader; an index of 1 causes the playback to be at the original speed.

18 Chadabe's *Electric Sound*, 70.

Figure 5: Varispeed Sample Playback



A fractional index causes the playback to be slowed, an index greater than 1 causes the playback speed to be increased, and a negative index causes playback to be reversed. Depending on the material being replayed, the pitch-shifting side-effects can be innocuous, except for when the playback rate index is actually changing, during which time the warping effect is extremely noticeable. Causing the playback speed to shift more abruptly reduces the warping effect but can result in noticeable artifacts, especially pops.

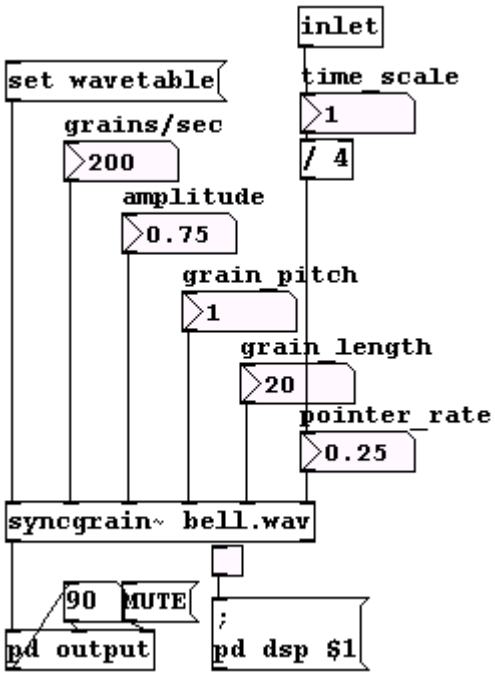
A novel solution to the side effect of frequency warping was concocted as early as 1946, when the British physicist and Nobel Laureate Dennis Gabor invented a device capable of changing the time scale of a sound without changing its pitch, which he called the "Kinematical Frequency Convertor." The basic physical mechanism was developed by music concrete gurus Pierre Schaeffer and Jacques Poullin and called the Phonogene. It was later marketed by the German company Springer as the Zeitregler and the Tempophon. The Tempophon was used in Herbert Eimert's 1963 piece "Epitaph fur Aikichi Kuboyama." [19]

19 Curtis Roads' *Microsound* (Cambridge, Mass.: MIT Press, 2001), 61 .

These machines work by time-granulating the recorded sound through the use of multiple, spinning playback heads. The heads spin across the tape (or, in Gabor's case, film) in the same direction that the tape is moving. The result is that the sound on tape is "sampled" at regular intervals, forming short "grains" of sound. To achieve time expansion, the spinning speed of the heads is increased, causing multiple copies of the original signal to be taken. When summed, these grains form a time-stretched version of the original signal. The pitch of the recorded material is contained within each grain; the rate at which the sequence of grains is played back effects the playback speed but does not effect the pitch . Conversely, to change pitch without changing duration, one needs only to change the playback rate of the original (and therefore alter the frequency content of each grain) and then use the time modification process to restore the sound to its original duration. [20]

This process is an early analog form of what is commonly known today as synchronous granular synthesis. The digital implementation of this technology has been widely explored and thoroughly documented.[21] A PD implementation of basic synchronous granular synthesis is executed by **syncgrain~** and pictured in **Figure 6**. The original waveform in **wavetable** is sliced into a series of segments, each 20 milliseconds long. To avoid

Figure 6: granular synthesis



20 Roads, *Microsound*,61-62.

21 See Curtis Roads' *Microsound*.

pops, each grain begins and ends with a smooth fade. At any one point in time, four grains are playing simultaneously (this measurement is called *grain overlaps*); each grain begins 5 milliseconds after the previous grain began. Playback speed (and therefore duration) is controlled via a playback rate index, which effects where each successive grain begins reading from **wavetable**, a measurement called the *grain increment*. Grain increment is calculated using the formula $inc_{gr} = \frac{grainLength * samplingRate * index}{grainOverlaps}$. When replaying without time expansion or contraction, the index is 1 and each grain begins reading $\frac{.02 * 44{,}100 * 1}{4} = 220$ samples after the previous grain began. As the index is varied, each grain starts earlier or later in **wavetable**. Regardless of the index, each grain is still 20 ms long and reads through **wavetable** at the sampling rate.

This method of time-domain granulation is not perfect, however. The act of quickly fading in and out—which is effectively what we are doing—is, by definition, a form of amplitude modulation, and, like other forms of amplitude modulation, the frequency content of the sound being modulated is effected when the modulation occurs at audio frequencies. If grains do not overlap with precise evenness, the rapid fading in and out that results has the same effect as ring modulation, introducing audible sidebands that result in buzziness. Sidebands will be created at the sum and difference of each harmonic component of the signal contained within the grains and the modulating frequency, F_m, where $F_m = \frac{1}{grain\_duration}$. This added frequency content cannot be removed.[22] Furthermore, the relationships between grain length, grain rate, waveform frequency, and perceived pitch are complex. A grain length that is incompatible with the

22 Dodge and Jerse's *Computer Music*, 92.

waveform frequency can lead to distortion ranging from octave doubling to broadband noise. Matching the grain length to the frequency of the grain's waveform can be exceedingly difficult, especially if the sound being granulated has complex and shifting spectral content. [23]

A far more effective technique involves rapidly taking a series of static snapshots of the signal's frequency content using a Fast-Fourier Transform (FFT) or similar technique. The snapshots (called "frames") can then be restored to the original time-domain waveform using an Inverse Fast-Fourier Transform (IFFT). If the rate in which the frames are played back is changed, the frequency content, which is contained within the individual frames, is not altered; only the length of the sound is effected. The most widely used version of this frequency-domain technique is referred to as *phase vocoding*. It is largely free from the artifacts that are common in time-domain granular synthesis, and is thus more widely used.

Phase vocoding is, however, also not perfect. Depending on the size of the frames used to transform the signal, either time smearing or frequency smearing, or both, occur. Longer frame sizes result in more accurate frequency analyses, but provide less information about the location of the frequency content in time. This tradeoff is inherent in all Fourier-based processing. [24] Furthermore, the windowing function that is used to isolate each frame for transform (essentially by fading in and out, as in granular synthesis) has an effect on the resulting snapshot of the window's frequency content: the resulting transform is of the convolution of the input signal and the windowing function. This has the potential to add audible sidebands, mathematically similar to the sidebands

23 Roads, *Microsound*, 93-96.

potentially created by granular synthesis. However, by using smooth windowing functions, these sidebands can be made to be much less problematic, or even unnoticeable. [25] Both the power and the failings of frequency domain processing, and especially phase vocoding, are well documented.

From an interface perspective, the phase vocoder is identical to the granulator; they both have an input for the "playback rate index." One needs only to hook either of them up to the "playback rate index" fader that was used earlier to achieve time-scaling without affecting frequency content. Provided one's hand is fast and accurate enough to follow the incidental and expressive tempo changes of an ongoing performance, one could match the playback of a prerecorded electronic part to the performer's tempo.

Beginning to listen: automated time synchronization

Our performance model so far consists of a performer playing along with a pre-recorded electronic part. In order to make the performance more musical, we have sought to find a way to have the computer part speed up and slow down to follow the tempo of the performer. While we have more or less solved the technical problem of changing the audio's playback speed without changing its pitch, we are still stuck with controlling playback speed with an onscreen fader, which is an only slightly updated version of the lever on Earle Brown's varispeed tape player. Its now time to tackle the larger problem, the task of figuring out a musical way for the performer to communicate his tempo to the computer.

---

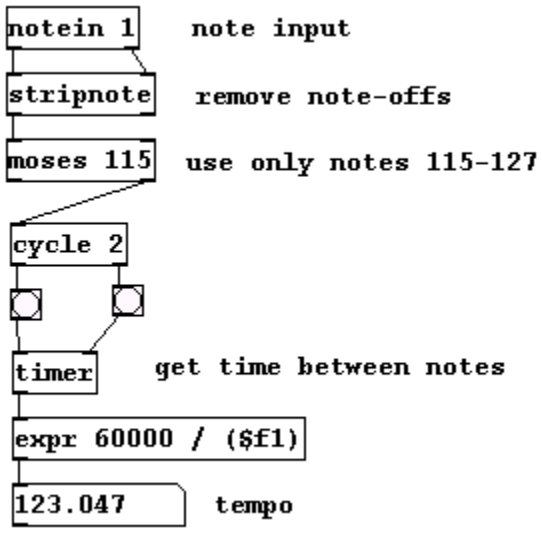
24 Roads, *Microsound*, 251.

25 Ibid., 246-247.

In the simplest case, the performer could communicate the current tempo to the computer by tapping the beat on a MIDI controller such as a foot pedal. Alternately, we could choose a specific key or range of keys on a MIDI keyboard and tap the beat there. Winkler describes a number of ways in which the performer can directly tap out the tempo for the computer. [26] At their core, all of his patches calculate the tempo using the formula $Tempo = \frac{60}{\Delta T}$ where ΔT is the time in seconds between taps on the key or pedal. The tempo is expressed in beats per minute.

A simple PD version of this approach is pictured in **Figure 7**. The user taps the tempo on any key in the top octave of the keyboard, and the patch calculates and displays the tempo. The tempo is recalculated once every two taps; this allows the user to update the tempo throughout the piece by two simple taps on the upper keyboard.

Figure 7: Tap Tempo Calculator



To calculate our "playback ratio index" from our tapped tempo, we need only to divide the calculated tempo by the expected tempo. If the two are exactly the same, we get an index of 1; otherwise, the index is scaled up or down appropriately.

This beat-tapping approach is an improvement over our onscreen fader, since the performer doesn't need to remove his hands from the keyboard to enter the tempo. The computer has begun to pay attention to what the performer is doing. The performer, though, must still think to stop, move to the top of the keyboard, and update the

26 Winkler's *Composing Interactive Music*, 137-155.

computer's tempo. It would be ideal for the computer to be able to extract the tempo from the human's performance itself, without the need for the performer to explicitly tap out the tempo. This process, known as beat extraction or beat tracking, is no small task, and extensive research has been devoted to implementing fast and reliable beat tracking systems. The many different approaches to beat tracking can be lumped into two general groups, called "rule-based" and "connectionist" systems. As we're about to delve into a discussion involving a more serious level of programming, it is important at this point to say a few words about computer program design philosophy, most notably the difference between "classical" and "connectionist" programming.

Background: Rules versus Neural Nets

Classical Programming

Programs using the "classical," "symbolic," or "rule-based" approach work by creating and manipulating symbols. As a simple example, suppose we'd like to write a program to determine whether a note played on a keyboard is loud, soft, or medium. We'll assume that the keyboard is outputting MIDI, and that the MIDI data is being buffered and parsed by some external program. This external program will be feeding our program MIDI velocity values for each note that is played.

We'll do this example in the C programming language, since C is fairly easy to read and understand even for readers who have never done any programming. First, we'll create some symbols. We'll need an integer to hold a newly input velocity value, which we will call *NewVelocity*. We will also need a way to store where the divisions between our velocity types will fall. We'll use integers for this, and call them

LowMediumBound and *MediumHighBound*. We'll give these boundaries some initial values, ones that will split the 128 possible velocity values more or less in three parts: LowMediumBound will equal 42, and MediumHighBound will equal 84. We'll declare them in C as follows:

```
int NewVelocity;
int LowMediumBound = 42;
int MediumHighBound = 84;
```

Now we need to come up with some rules that govern the relationship between these symbols. Stated simply, if the new velocity value is less than LowMediumBound, the velocity type is low; if the new velocity value is greater than MediumHighBound, the velocity type is high; if it falls between LowMediumBound and MediumHighBound, the velocity type is medium. If the velocity equals zero, we should ignore it, since it's acting as a note-off and not a note-on. Finally, we want to wrap this rule in a method, called determineType that is called whenever a new velocity value is received. This method should accept the new velocity value and should print the velocity type. The entire program looks like this:

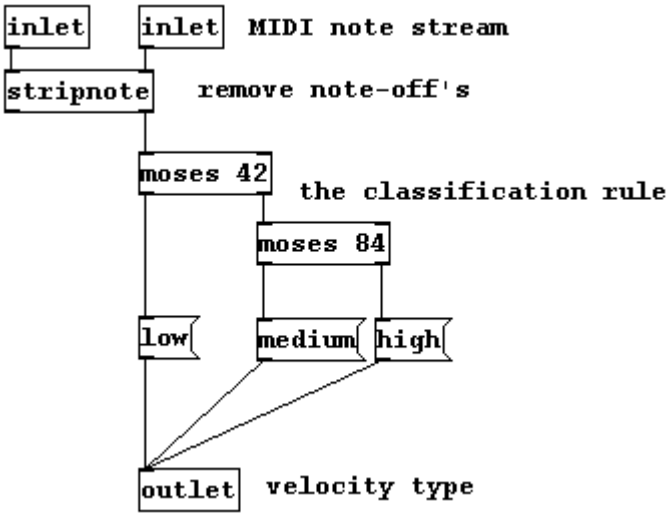
```
int LowMediumBound = 42;
int MediumHighBound = 84;

int determineType (int NewVelocity){

        if (NewVelocity < LowMediumBound)
                printf("Velocity is low \n");
        else if (NewVelocity > MediumHighBound)
                printf("Velocity is high \n");
        else if (NewVelocity == 0)
                printf("Velocity is zero \n");
        else
                printf("Velocity is medium \n");
        return 0;
}
```

The PD equivalent of this program is pictured in **Figure 8**. Note that this is not at all a good way to characterize velocity, since our perception of whether notes are loud or soft is based largely on the musical context in which they are found. This does, however, serve as a good example of rule-based programming.

Figure 8: Rule-based Velocity Classifier



Connectionist Programming

The other approach to programming is called "connectionism," "distributed," or "sub-symbolic" programming. Rather than using symbols and rules, connectionist systems use neural networks, often called neural nets. A neural net consist of a number of individual units joined together in a pattern of connections. The individual units, often called "nodes," are differentiated into different types, each of which has a specific role; the most common are input nodes, output nodes, and "hidden" nodes.

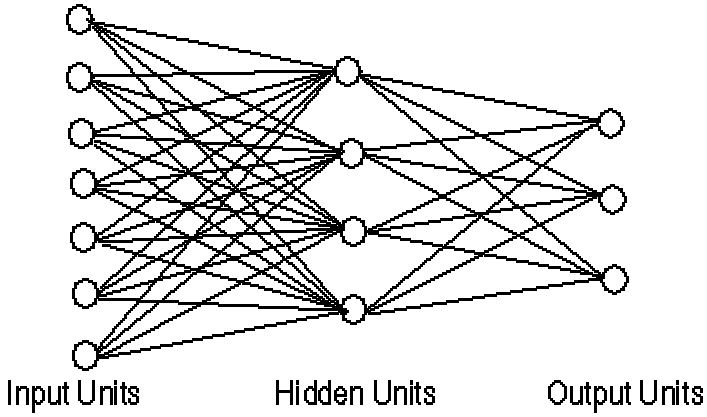
Each input node has an "activation" value that represents something external to the net. For example, if the input nodes are keeping track of what notes are currently being played by a performer, we could have one node for each pitch-class (one for C, one for Db, one for D, etc.): the "activation" value of each node will be 1 if a note of that pitch-class is currently being played, and otherwise will be 0. Each input node sends its activation value to each of the nodes to which it is connected. Each of these receiving nodes calculates its own activation value depending on the activation values it receives

from the input units, and then passes its activation value on through the net. In this way, activity at the input nodes will eventually show up as a change in the activation values at some or all of the output nodes.

The behavior of a neural net is determined entirely by its topography, by the unique layout of the connections between the various layers of nodes. Many different types of neural nets have been created; the most common has every input node connected to every output node, sometimes with a layer of hidden nodes inserted in between them (see **Figure 9**). This is called a "feed forward" net, since information is only passed from input to output—there is no backward communication built into the structure of the net. A net's topography generally does not change when it's in use.

Neural nets are capable of correlating patterns of input with patterns of output by translating inputs into patterns of activation that propagate through the net. The pattern of activation set up in a net is determined by its weights, by the strength of the various connections between the nodes. An input node will not be equally connected to every output node, but rather a separate weight on each connection will scale the input node's activation level before it is sent on to an output node. Weight values may be positive or negative; a negative weight represents an inhibition of the receiving node by the activity of a sending node. The activation value at each output node is calculated by summing together the contributions of all input/hidden units: the

Figure 9: Neural Net



"contribution" of a node is calculated by multiplying the weight of the connection between the sending and receiving node by the sending node's activation value. "This sum is usually modified further, for example, by adjusting the activation sum to a value between 0 and 1 or by setting the activation to zero unless a threshold level for the sum is reached".[27] The pattern of activity at the output nodes is then mapped to an overall output value. The node with the highest activation level may, for example, be chosen, and its number or label will be used as the net's output.

Having the correct connection weights is the most essential part of any connectionist system; if the weights are not set correctly, a net will not function properly. The number of connection weights can be large; a net with 12 input nodes and 12 output nodes will have at least 144 connections, and even more if net contains hidden nodes. Luckily, the ability of neural nets to set their own weight values is one of their most useful and intriguing qualities. Given repeated exposure to a set of expected patterns and the ability to correlate those patterns with expected output, neural nets can essentially "learn their own rules" by setting their own connection weights through the process of backpropagation.

Backpropagation involves the use of a training set, which consists of many examples of expected input patterns along with the output that is to be associated with each pattern. The connection weights of the net to be trained are initially set to random values or to zero. The net's input nodes are then repeatedly exposed item-by-item to the exercises in the training set. For each training item, the net's output is compared to the correct output listed in the training set. If they are the same, then the net got the answer

27 James Garson's "Connectionism", The Stanford Encyclopedia of Philosophy (Winter 2002 Edition), Edward N. Zalta (ed.).

correct and nothing more need be done. If, however, the training set's answer and the neural net's answer are different, the connection weights are adjusted slightly in the direction that would bring the net's output values closer to the values for the desired output. Specifically, the connections between the input nodes that were activated during the training exercise and the output node that erroneously received the hightest sum are adjusted in the negative direction, and the connections between the activated input nodes and the output node that should have won are adjusted in the positive direction. After the adjustments are complete, the net continues on with the next training exercise. When the net makes it through the entire training set without getting any answers incorrect, training is complete. Aside from being able to associate items from the training set with correct output, the net many also have learned to generalize to the desired behavior for inputs and outputs, allowing it to recognize patterns and produce correct output for items that were not in the training set.

Classical "symbolic" programs attempt to emulate, on some level, the way humans think about what they think about—start with a group of symbols, do things to the symbols to change them in some way, follow directions step-by-step. Connectionist programming, on the other hand, attempts to simulate human brains on a physical level: individual nodes and connections can be view as simulations of neurons and synapses in the brain (hence the term "neural network"). While the ability of models of this type to produce complex behaviors, solve intricate problems, and even simulate human perception is hotly debated, the many incredible successes in this field of programming have been well documented.

...continued Graphic used with permission of the editor.

Rules can be too exclusive, symbols too brittle; faced with the unexpected, with input that is beyond the anticipated bounds, rule-based systems fail catastrophically. Connectionist systems, on the other hand, degrade gracefully, and have the inherent ability to deal with input that is similar to, but not identical to, expected input. They can, in a sense, venture a guess when they do not "know" the right answer. Classical systems are inherently explicit, with symbols and rules that are spelled-out and obvious; connectionist systems can understand and operate on patterns that would be very difficult to express in terms of rules and symbols. Connectionism provides a model of human intelligence that is more in line with how we feel about the operation of our own nervous systems, a theory of artificial intelligence that places the majority of the processing literally "below" the formation of rules and symbols. Consciousness and perception at least feel more immediate than classical systems are capable of emulating. After all, do neurons in the optic nerve, for example, really follow an explicit set of rules when processing information from the eye, or is something more "low-level" going on?

Connectionist models of programming are not, however, without their problems. From a neurological point of view, neural nets are a vastly oversimplified model of the brain and its neurons. Differentiation and specialization among neurons is completely ignored, as is the fact that the brain must contain a large number of "reverse" connections that function as it as operating, rather than in a separate "training" stage, as during backpropagation. Furthermore, it is obvious that, at some level in human consciousness, symbols and rules are indeed constructed and used, and neural network models seems ill-equipped to do this. Practically, many problems require the structure provided by logic

and higher reasoning, and neural nets have shown themselves to be especially bad at this kind of processing.

To be sure, this is not a purely academic debate, even in the context of this paper. Music theory is concerned with understanding how we understand music. Interactive systems must be taught to understand music, in at least a very limited sense, if they are to be of much use in a performance. We are attempting, then, to model the behavior of human musicianship, if not its mechanisms. The way in which we choose to do that will have an immense effect on the eventual shape of the outcome.

Beat extraction: existing research

Lerdahl and Jackendoff's "A Generative Theory of Tonal Music" provided a starting point for much research in human and computer rhythmic perception, primarily because it postulated what was perhaps the first cohesive perceptual definition of rhythm, and clarified the elements of musical rhythmic structure. The authors establish sets of rules, which lie out formal conditions for establishing hierarchical grouping structures and describe conditions that determine which of the large number of possible hierarchical groupings of any passage of music are actually likely to be perceived. [28] Lerdahl and Jackendoff describe the metrical structure of a piece of as being like a grid, and advance the idea that music seems to consist of two interacting time scales: the discrete time intervals of a metrical structure and the continuous time scales of tempo changes, expressive timing, and temporal "noise" from limitations in the human motor system

28 Eric Clarke's, "Rhythm and Timing in Music." in *The Psychology of Music*, 2d ed., ed. Diana Deutsch,. (San Diego, California: Academic Press, 1999), 478-479.

during performance.[29] The process of assigning a single rhythmic value to a range of different note durations, by filtering out the continuous temporal "noise" and focusing on the metric grid, is called quantization.

Rule-based Beat Extraction

Rule-based quantization systems use knowledge about human rhythmic perception and preferred metric groupings to quantize the input data, and even to construct higher-order metric groupings. One of the most successful of the rule-based systems is Longuet-Higgins and Lee's rhythmic parser. After hearing two notes, the parser estimates where the next notes will fall. This estimate is revised in light its verification or rejection.[30] Once beat estimates are verified, the system attempts to move up the metrical hierarchy by combining the established metrical units into a single metrical unit. Meter, therefore, is created from the bottom upward over time, but quickly becomes a top-down structure that guides the model's perception of new input.[31] The parser continuously applies Lerdahl and Jackendoff-style well-formedness rules to update and improve the parser's concept of meter.

Rule-based systems have some inherent flaws that need to be overcome for them to be of use in an interactive music system. These systems generally have problems with irregular metrical structures such as syncopations and upbeats, as these are exceptions to the basic metric rules and are difficult to express algorithmically. When rule-based systems are faced with input that doesn't conform to expectations, they break down

---

29 Peter Desain's "A connectionist and a traditional AI quantizer, symbolic versus sub-symbolic models of rhythm perception." *Contemporary Music Review* 9, Parts 1 & 2 (1993): 56.

30 Longuet-Higgins and Lee's *The Perception of Musical Rhythms* (1982): 118-119.

rapidly. When faced with rhythmic ambiguity, rule-based systems will generally not produce a "best guess" answer—they will, instead, produce no answer at all. Finally, and perhaps most importantly, many rule-based systems, including Longuet-Higgins and Lee's rhythmic parser, operate best when they have access to a piece of music in its entirety—in other words, not in real time. This limits their ability to participate in a live musical performance.

In light of these problems, and because of the fact that many researchers feel that the processes involved with human rhythmic perception are more immediate and automatic than models such as the Longuet-Higgins and Lee parser suggest, many beat tracking systems take a different approach, one that is grounded in connectionist design philosophy.

Connectionist Beat Tracking

Perhaps the most successful, and certainly the best documented, connectionist beat tracking system is Desain and Honing's Connectionist Quantizer. This system operates on inter-onset intervals (IOI), which is the time lapse between the start of a note and the start of the previous note. A number of cells store recent IOI values; the cells' values are gradually increased or decreased to bring them closer to small integer multiples of each other. The quantizer's output is a set of "expectancy curves" that predict when new notes will occur; these curves are representations of the metric structure of the music.[32]

---

31 P. Desain and H. Honing's "Computational models of beat induction: the rule-based approach." *Journal of New Music Research* 28, no. 1 (1999): 29.

32 Desain and Honing's *The Quantization of Musical Time: A Connectionist Approach* (1989): 57-60.

A different sub-symbolic approach is taken by Large and Kolen. They view repeated metric pulses as a form of sinusoidal oscillation, and see beat tracking as the process of tracking an oscillator's period and phase. Their system consists of a number of adaptive oscillators that "fire" when they are sufficiently activated by an input signal, resulting in a series of pulses that approximates the input's metric structure. Each oscillator's phase is effected by the input signal only during parts of the oscillator's cycle; in this way, each oscillator's cycle is reinforced by new input patterns that are similar to previous input patterns. Petri Toivianinen's *Interactive MIDI Accompanist* is based on Large and Kolen's system of oscillators, but contains several modifications that makes it more useful for live performance applications. First, the output of the system is smoothed to remove abrupt changes in output period that would be disruptive to an ongoing performance. More importantly, it attempts to combine and coordinate individual input notes into larger groups before passing them onto the oscillator network; the reduces the overall effect of very short notes and allows the system to track a larger metric unit.[33]

The sub-symbolic nature of connectionist quantization systems is quite powerful; the input of a few notes will, regardless of tempo or expressive timing deviations, create a projection of the metrical scheme that will be refined with each new input event. The system is very context-dependent, so syncopations are easily handled. Ambiguous or unfamiliar input causes the system to attempt to guess, and even, in some systems, learn the correct output for newly discovered patterns of input. The cost of this increased power and flexibility is paid in computational load: connectionist systems are generally

33 Rowe's *Machine Musicianship*, 130-135.

more computationally intensive than rule-based systems. Nonetheless, many of them are still fast enough to operate accurately in realtime.[34]

Summary: The challenges of beat tracking

Successfully extracting a tempo from an ongoing human performance is indeed no small feat. To accomplish this goal, a beat tracking system must:

1. be able to understand and incorporate a variety of temporal multiples, divisions, and subdivisions of the beat (unlike the beat-tapping approaches described above, which require the user to explicitly and exclusively tap the pulse);
2. understand and smooth over the temporal "wow and flutter" that is inherent in all human performances of music;
3. be able to distinguish small, unintentional tempo deviations from larger, more directed expressive tempo changes, and should be able to accurately track the latter;
4. operate without an internal representation or score of its expected input;
5. work in realtime.

Criteria 1-3 are necessary for any system that will be dealing with real human performance input. Criteria 4-5 are necessary for the system to be able to participate in an ongoing performance of improvised or partially improvised music.

34 Desain and Honing's *The Quantization of Musical Time*, 60-65.

Beat tracking in PD

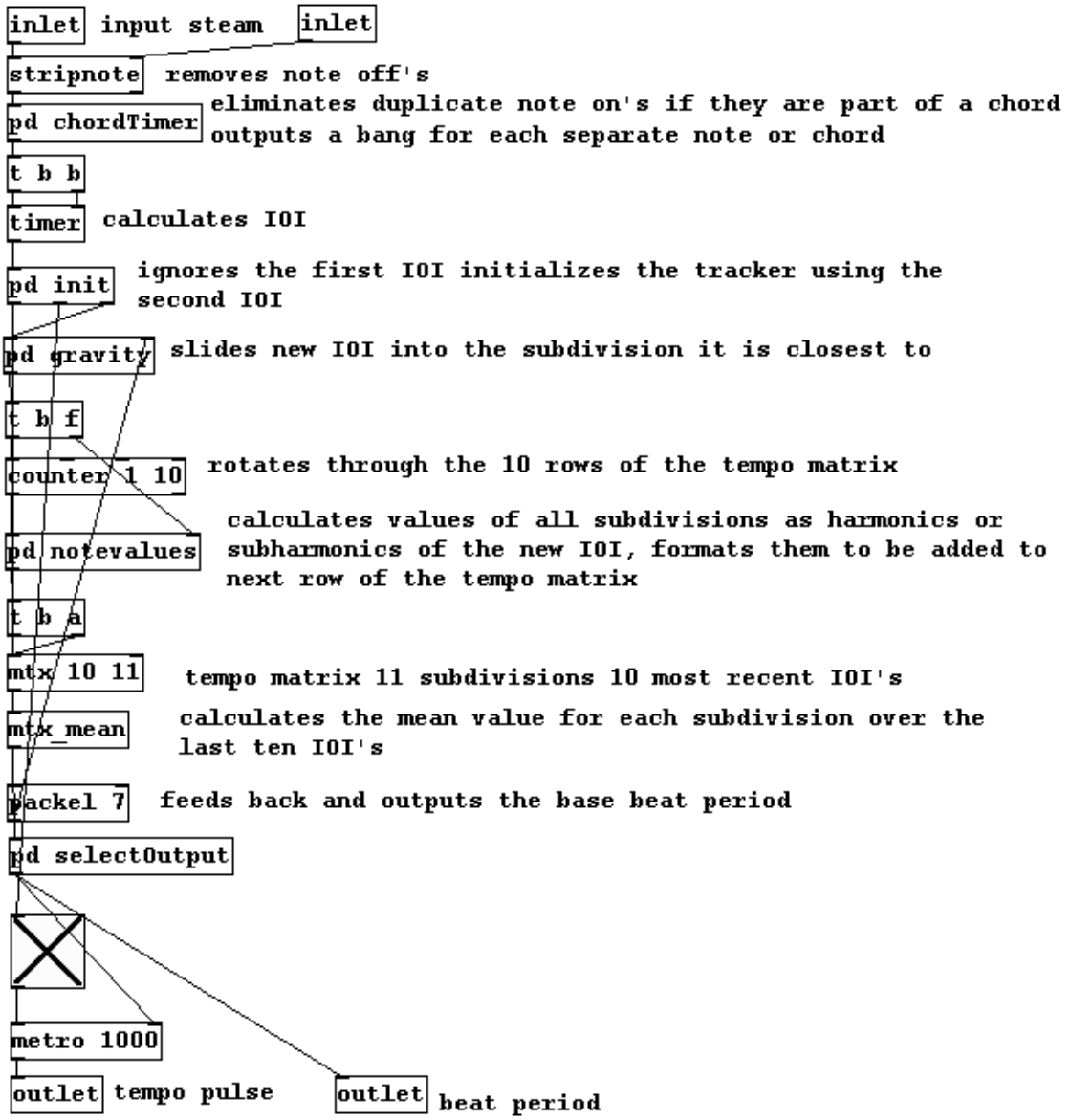
I have constructed a quasi-connectionist beat extractor that is loosely based on the research of Desain and Honing, Robert Rowe, and others. It is native to PD, utilizing ZEXY's powerful matrix objects to simplify calculations for realtime operation. It accepts a stream of MIDI notes as input, and outputs a sequence of BANG messages on the beat, as well as the current beat period in milliseconds.

The beat tracker executes the following steps, described in detail below, each time new input is received:

1. MIDI input data is parsed, and everything but Note On messages is discarded;
2. Note-On Messages that come extremely close to a previous Note-On Message are discarded;
3. The Inter-onset Interval (IOI), the time between the new Note-On and the previous Note-On, is calculated;
4. The new IOI is compared to the current beat theory and the metric subdivision of the theory to which the new IOI most closely matches is determined;
5. The new IOI is scaled to its quarter-note equivalent (e.g. an IOI thought to be an eighth-note is multiplied by 2);
6. The quarter-note version of the new IOI is added to the tempo matrix, and a new beat theory is calculated;
7. The new beat theory is output, and the metronome generating the BANG messages on every beat is updated with the new tempo.

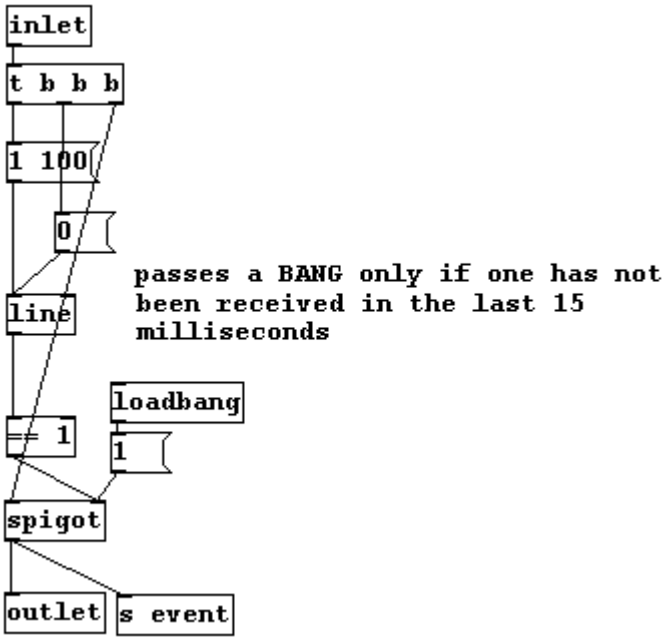
Figure 10 is of the parent patch of my PD beat extractor, called TempoTracker. MIDI data enters at the top of the patch; **stripnote** removes note off messages and passes note on messages. Notice that, as Note-Off messages are discarded, this tracker does not consider actual note durations, but rather works with Inter-onset Intervals.

Figure 10: TempoTracker Parent Patch



The **chordtimer** subpatch, pictured in **Figure 11** handles note on messages that come extremely close to each other. Such notes may be members of a chord, part of a trill, or accidentally "slipped" notes, but it is unlikely that they will be intentionally played rhythmically significant notes. **Chordtimer** outputs a BANG when a new note on is received only if another note on has not been received in the last 100 milliseconds. A long trill, therefore, will cause a BANG to be generated only at the very beginning of the trill. In the simplest case, a series of single notes is input into **chordtimer**, each causing a BANG to be sent.

Figure 11: chordTimer



At the core of the tempo tracker is a tempo matrix, called **mtx 10 11**. It is a 10x11 matrix that PD manages as a single, multidimensional data structure. Each row of the matrix holds a different IOI; since there are ten rows, the matrix holds the most recent ten inter-onset intervals. Guided by a **counter** object, the matrix cycles through its ten rows sequentially, writing the new IOI to the next row; when row ten is reached, the counter resets to 1 and the next IOI is written to the first row of the matrix, replacing its previous contents.

Each of the eleven columns of the matrix holds the value for a different metric multiple, expressed in milliseconds; the metric multiples include 1/6, 1/4 1/3, 1/2, 2/3, 3/4, 1, 1 1/2, 2, 3, and 4. These metric units are casually referred to by the names of the corresponding rhythmic notational unit, such as 1 = quarter-note, 2 = half-note, and so on. While this nomenclature could possibly be accurate, the actual musical value of each of the metric multiples depends on several factors, most notably the musical value of the notes played when the tracker was initialized. The initialization process is described in detail shortly.

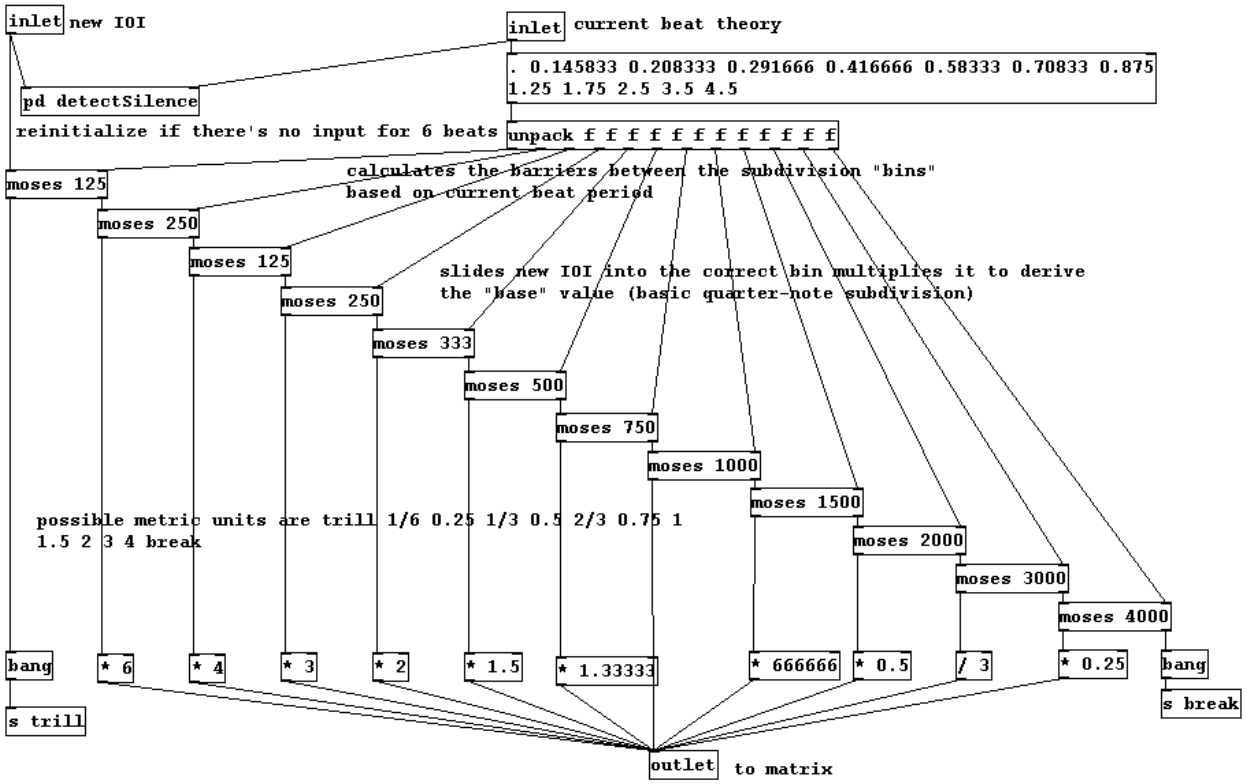
With each new input note, an entire row of the matrix is filled, using multiples of the calculated IOI. The matrix therefore contains a great deal of redundancy, as, given the value of one column, one could easily use multiplication to derive the value of the other columns. However, having the all the metric values calculated is necessary, as we shall wish to compare them to choose which one we wish to use to drive the output, and it is more efficient to calculate them inside the matrix.

Every time a new input note is received, the contents of the matrix are sent to the **mtx_mean** object, which calculates the arithmatic mean of each column, thereby calculating the average value of each metric unit over the last ten input notes.

When a new note is received, its IOI is calculated, and is sent to the subpatch **gravity**, which is shown in **Figure 12**. This subpatch compares the new IOI to each of the average metric values that were calculated by the **mtx_mean** object. The new IOI is "slid" into the metric "bin" to which it is closest. Then, depending on which bin it was slid into, the new IOI is scaled so that it is the quarter note equivalent of the new IOI. If,

for example, the new IOI is closest to the average value for the half-note unit, it is multiplied by .5.

Figure 12: Gravity Subpatch



The **gravity** subpatch also handles two special cases: very long notes, and very short notes. Notes that are shorter than one-eighth of the current beat theory, equal to a thirty-second-note or less, are assumed to be members of a trill that somehow made it through **chordtimer**. The IOI is discarded. In the case of notes that are farther apart than a dotted whole-note, the assumption is made that the performer has finished a section and has begun playing another; a BANG is sent to INIT, which causes the tempo tracker to reset itself and wait for new input. Similarly, the subpatch **detectSilence** waits for six

beats, using the most recently calculated tempo. If no new input has been received in six beats, it is assumed that the performer has fallen silent, and the tracker is reset.

Once the quarter note-equivalent of the new IOI is calculated, it is passed to the subpatch **notevalues**, which calculates the values of all the metric units based on the new IOI's quarter note equivalent. **Notevalues** also formats these values by attaching matrix row and column numbers. The formatted values are then added to the next available row of the matrix.

After the new data has been added to the matrix, the matrix is sent to the **mtx_mean** object, and a new tempo is calculated. The new values of each metric multiple are sent to the **selectOutput** subpatch, which chooses one of the metric units to drive the output. Initially, the "quarter-note" (x1) value is used; if this moves outside the acceptable range of tempos, the subpatch can "shift" to using a different metric unit to drive the output. If, for example, the beat period is less than 400 ms, the current metric unit is too fast to be the basic pulse; **selectOutput** "upshifts" and begin using the "half-note" (x2) metric value to drive the output. There are two possible "shifting patterns:" the "simple" pattern can shift to the x1, x2, x4, x ¼, and x ½ metric units, while the "compound" pattern can shift to x1, x ⅔, x3, x ⅓, and x ⅔ units. As their names imply, the simple pattern is designed for use with simple meters and the compound pattern works better with compound meters. The user must currently specify which shifting pattern to use by sending a 1 (simple) or 0 (compound) message to **s metertype** (the default type is simple). Whatever the chosen metric value it is used to drive the output by setting the rate of a **metro** object, which outputs a BANG once every beat. The beat period of the selected metric unit is also output from the tracker.

The subpatch **init** handles the special operations involved with turning the tracker on and off, which includes turning the output metronome on and off, as well as initializing and resetting the tempo matrix. The tracker takes three input notes to be fully initialized, but it begins outputting a beat after two notes. Most importantly, when **init** is sent a BANG from any patch in PD, it causes the tracker to cease output and reset.

The first IOI is ignored, since it will be the time since the tracker was last used or loaded. The second IOI (the time between the first and second notes) is the first valid IOI; it is used to fill all ten rows of the matrix, allowing an "average" to be taken, causing all subdivisions to be calculated, and allowing output to begin. The third IOI is placed in the first row of the matrix, and normal operation begins. The second IOI is therefore very important, for two reasons. First, the entire matrix is filled with data based on this IOI; this allows the **mtx_mean**, which normally averages the last ten IOI's, to start outputting without having to wait for ten notes to pass. The result, however, is that remnants from this second IOI will be inside the matrix (and will effect calculations) until ten more notes have been played. More importantly, this first IOI is assumed to be the x1 metric subdivision. All future input will be categorized around this IOI. While the **selectOutput** reduces this effect by choosing other metric units to base output on, the fact remains that the tracker's "perception" of future input is greatly shaped by this first valid IOI.

Performance comparison of beat tracking systems

A detailed analysis of the performance of **tempoTracker** was performed, using two MIDI test files containing J. S. Bach's *Jesu Meine Freude* and Scriabin's *Prelude in C* as test material. The test MIDI files were generated by a human pianist playing in time with a metronome, so they contain the usual amount of slight human rhythmic inaccuracies. The Scriabin example also contains a small amount of *rubato*, and a slight *ritardando* near the end of the excerpt.

Overall, the tracker performs very well. In both examples, the tracker begins tracking the eighth-note pulse, and, after several beats, "upshifts" and tracks the quarter-note pulse. In the Bach example, the tracker continues tracking the quarter-note pulse, catching every downbeat until the end of the example. Similarly, in the Scriabin example, the tracker follows the quarter-note pulse and smoothly tracks the *ritardando* at the end of the example. In both examples, the tracker's beat is occasionally ahead of or behind the human's beat, especially during the *rubato* in the Scriabin example, but this inaccuracy is slight and never lasts longer than two or three beats.

In the Bach example, the tracker manages to find and follow a rhythmic unit, and to "upshift" to find the basic metric pulse. In the Scriabin example, on the other hand, the "upshift" did not result in the tracker finding the basic metric pulse, but rather a higher-order, somewhat spurious, subdivision. Most musicians would "feel" the Scriabin piece in groups of five eighth-notes, but the tracker follows it in groups of two eighth-notes. As a result, the tracker's beat is on the "and" of every other perceptual downbeat. The tracker, having identified and followed rhythm, is unable to understand it in terms of

meter, largely because it does not have access to the other elements of the music, such as chord type and dynamics, that help human's extrapolate a meter from a number of possible rhythmic groupings. However, the **tempoTracker** does manage to find and track a basic rhythmic unit, and does so rather well, even in a complex rhythmic environment.

The output of tempoTracker during these tests in both raw and analyzed form is included in the appendix of this paper. The test MIDI files are included on the CD-ROM appendix.

4. Machine Decision Making

Pseudorandom Number Generators

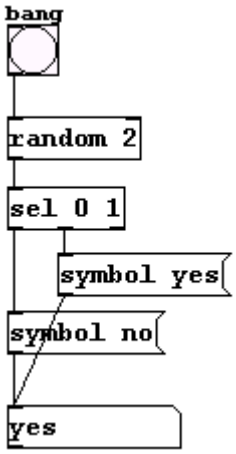
Our system to this point has consisted of a computer replaying a wavetable, the playback speed of which is controlled by a realtime tempo tracker. While this does represent a more "intelligent" behavior than the tape player we started with, it is not a particularly innovative use of a computer. It is, in fact, only a slight improvement on the Tempophon, a technology that is over fifty years old.

While we have managed to increase human→computer Influence, the amount of computer Participation is still minimal. While the computer has begun to listen, it is not playing, in the sense that it is not making decisions about when and how sound will be produced. Computers can be programmed to make these decisions, and the outcome can be based, to a greater or lesser degree (depending on the amount of Influence at the time), on the activity of the human performer.

As an illustration of a basic case of computer decision making, the computer is presented with a simple philosophical question: yes or no. The outcome of this choice is not to be based on any outside factors (there is no Influence) and the computer has no reason to pick yes over no (there is no weighting). In each such decision, the computer picks one outcome or the other in a seemingly random fashion. The computer is, essentially, flipping a coin. Activities such as these, which involve chance or randomness, are theoretically impossible on a computer, which is, by definition, entirely algorithmic and non-random. We can, however, simulate randomness using an algorithm known as a pseudorandom number generator.

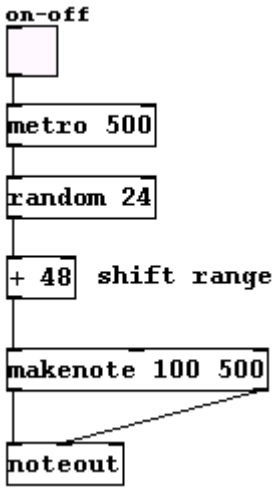
Random is PD's native pseudorandom number generator. It accepts an initialization argument of an integer, *n*, and whenever it receives a BANG message it outputs a randomly chosen integer between 0 and *n*-1. To answer our basic yes-no question, we could use a **random 2** object, say that 1=yes and 0=no, and BANG the object repeatedly. Yes and no will appear more or less the same number of times. This is demonstrated in **Figure 13**.

Figure 13: Yes-No



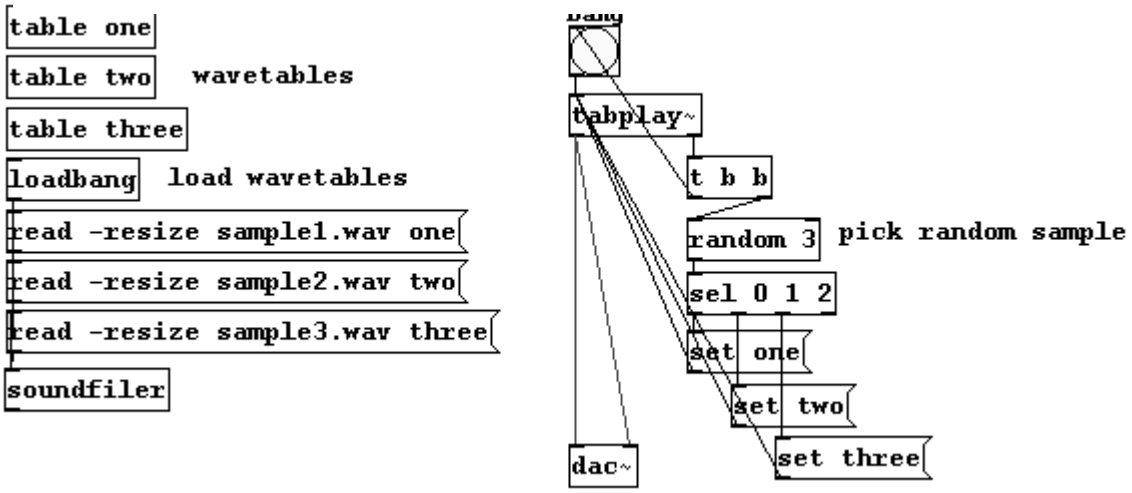
A more musical example, depicted in **Figure 14**, consists of a **Random** picking MIDI note numbers. The **metro 500** outputs a BANG every half second, causing the **random 24** to pick a number between 0 and 23; this range is then shifted up to 48-72 by adding 48 to every randomly picked number. This patch now randomly plays a note from the two middle octaves of the piano (note numbers 48-72) every half of a second.

Figure 14: Random Notes



A more sophisticated use of **random** is shown in **Figure 15**. Here, three wavetables have been loaded with short samples of sound. The **random 3** is used to randomly pick and play one of the wavetables. When the wavetable being read is finished, the **sfplay~** outputs a BANG, which is routed to the **random**, causing it to start playing another wavetable. In this way, the patch plays through the samples in random order.

Figure 15: Playing Random Samples



As a final example, we could use a **random** object to select rhythmic patterns and output the corresponding durational values at the correct times. Whenever a pulse is received, **patternGen**, pictured in **Figure 16**, randomly selects from ten rhythmic patterns, such as eighth-note eighth-note or half-note. At the heart of **patternGen** is a number of "pattern" subpatches, one of which is pictured in **Figure 17**. This subpatch serves two functions. First, it calculates the duration of one or more metric subdivisions based on the beat period, which is received in the patch's right inlet. Secondly, when it receives a BANG in its left inlet, it outputs the calculated durations with the appropriate metric timing. For example, the pictured subpatch outputs an eighth-note eighth-note pattern. First, it calculates the duration of each eighth-note as being half of the current beat period: if the beat period is 500 ms, each eighth-note should be 250 ms long. Then, when a BANG is received, the duration for the first eighth-note is output immediately. The patch then waits for the length of the first eighth note, and then outputs the duration for the next note. So, when a BANG is received, the example patch will output 250, will then wait 250 ms, and finally output 250 again. The "valve" section of the subpatch

Figure 16: patternGen

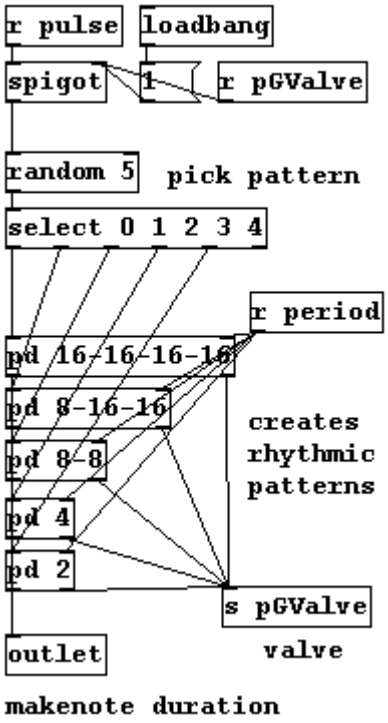
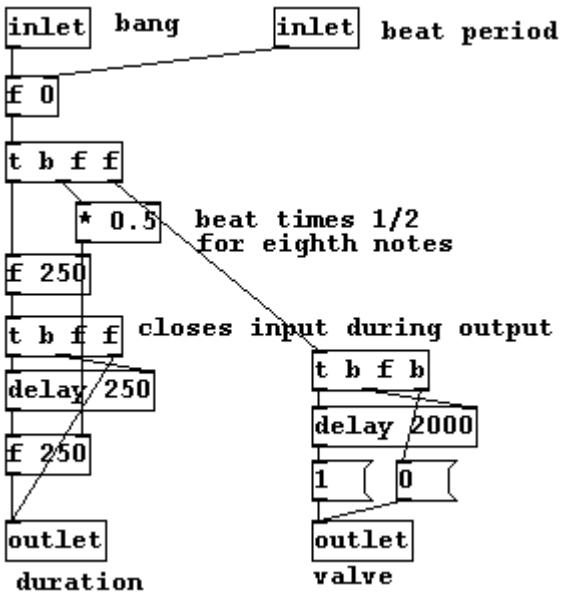


Figure 17: pattern subpatch

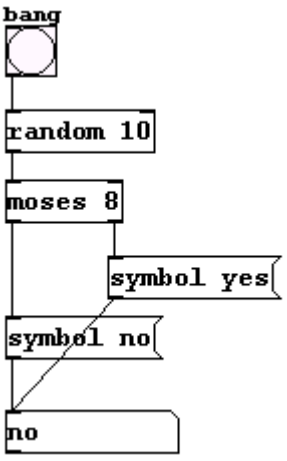


closes a **spigot** that prevents new patterns from being started before the current pattern is finished.

Weighted random number selection

Returning to our yes-no example, suppose that we want the computer to answer "yes" more often than "no." More specifically, say that we want one "no" for every four "yes" answers. This is called weighting, and is easily accomplished in PD. In **Figure 18**, the output from the **random 10** is sent to a **moses 8** object, which sends any input less than 8 out its left outlet, and any input greater

Figure 18: Weighted Yes-No



than or equal to 8 out its right outlet. The outlets are then connected to produce a 1 or a 0 respectively. If we were to BANG the **random** one hundred times, we would expect that the answer would be "yes" roughly 80 times. The weighting can be changed dynamically by resetting **moses** split point, and more complex weightings could easily be achieved using multiple **moses** objects in series.

Aleatoric Music

Music involving elements of randomness, indeterminacy, and chance, often called Aleatoric music, has a long and full history. A few brief examples will illustrate the variety and power of musical indeterminacy. John Cage, for example, used a process derived from the I Ching, in which he repeatedly flipped coins, to pick the contents and arrange the structure of his Williams Mix (1952) from pieces of taped sounds.[35] For "HPSCHD" (1969) Cage and Jerry Hiller wrote computer programs called ICHING, DICEGAME, and HPSCHD, which assembled musical scores and taped sounds by, in part, randomly swapping out measures of Mozart's "Musical Dice Game" and substituting measures from works by Cage, Beethoven, Chopin, Schumann, Schoenberg, and Bach. The resulting tapes and scores are played back, with each performer having his own self-directed starting time and tempo, and with multiple tape decks playing simultaneously. [36] Alternately, Curtis Roads used a computer to randomly pick the frequencies and other

35 Chadabe's *Electric Sound*, 55-56.

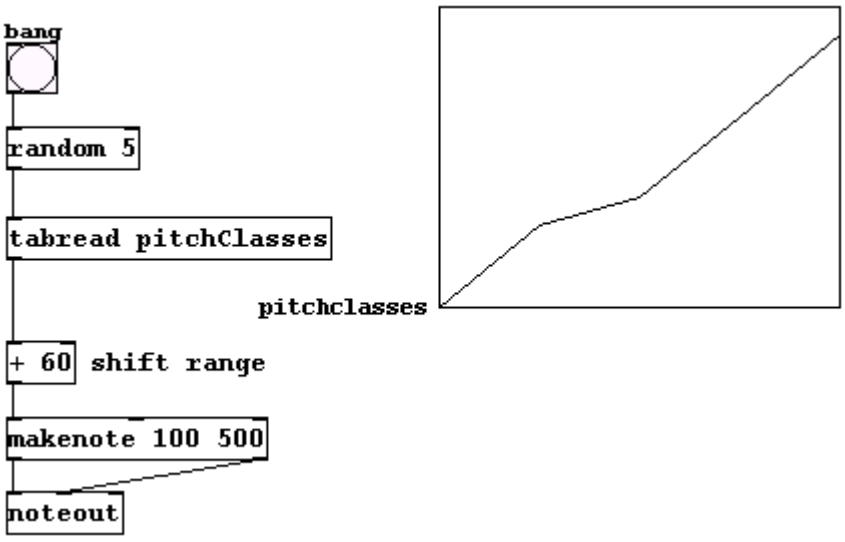
36 Ibid., 274-277.

characteristics of the hundreds of sonic grains per second that make up the sound events in "Half-life" (1999). [37]

These examples clearly illustrate the musical potential of randomness. However, while simply picking numbers at random can be useful, it is sometimes inappropriately uncontrolled and overly broad. After all, most composers and improvisers do not work by randomly picking notes from the set of all possible notes (although someone may at some point have attempted this). Instead, composers often start with a smaller set of pitches, a key, chord, or scale, and work from there. In a situation where more control is needed, when the output needs to be more constrained, approaches other than random number picking are needed.

Several helpful random-order object have been ported from Max to PD via the *Cyclone* library, namely the objects called **drunk** and **urn**. **Drunk** allows the user to set an upper limit and a maximum step size; when banged repeatedly, the output "walks" randomly around between the upper limit and zero, with no leaps bigger than the set maximum step size. As a tool for generating sequential note numbers for melodies, **drunk** generates much more traditionally coherent output than **random**.

Figure 19: Random Array Reading



37 Roads' *Microsound*, 309-310.

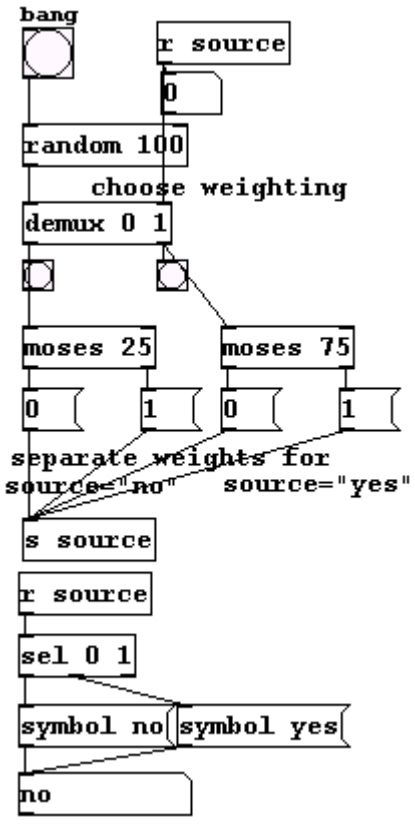
Urn, on the other hand, holds the numbers 0 - *n* and outputs them in random order, one after the other, without repeating any until all of the possible numbers have been output. This type of serial sequence of values could easily be used to control any (or all) parameters of an unfolding stream of sound.

Suppose one wants the constrained random output to consist of a more specific set of numbers than 0 - (*n*-1). One might, for example, want the computer to randomly pick notes from a specific chord or scale. To accomplish this, one could couple **random**, **urn**, or **drunk** with an **array** of numbers. The **array** will contain the values we wish to choose from (the note numbers of a scale, for example) and the output of the random-order object will act as the index to the array. This is depicted in **Figure 19**. What's more, since PD's **array** objects are both a data structure and a graphical interface, the user can change the contents of the **array**, and therefore the values present in the output, in realtime, simply by clicking and dragging.

Sequential constraint: Markov chains

Returning to our philosophical yes-no question, suppose we want each answer to depend partly on the answer that came before. Suppose, for example, we want the answer to change 75% of the time; if the computer's last answer was "yes," its next answer should be "no" 75% of the time. Clearly, the static weighting described above will not work for this situation, since the weights need to change depending

Figure 20: Markov Chain



on the last output. **Figure 20** pictures a patch designed to handle this situation; the weights are reset according to the patch's last output. This is known as a *Markov* chain. A Markov chain consists of a series of states, and of transitions between these states:

> The state at the beginning of a transition is the *source*; the state at the end is the transition's *destination*. In a Markov chain, each successive destination of one transition becomes the source of the next. The behavior of the chain is captured by a table of transition probabilities, which give the likelihood of any particular destination being reached from some source. . . The number of pervious states used to determine the net destination state is called the *order* of the chain. Chains that use only the source state to determine a transition are called *first order*; chains that use the two most recent states to find the transition to the next are *second order*, and so on.[38]

As the number of possible output values grows, maintaining and loading the appropriate weighting for all of the different possible output values becomes unmanageable. Luckily, PD has an object called **prob**, again ported from Max to the *Cyclone* library, which is designed to automatically perform this exact task: it maintains probability tables for the construction of first-order Markov chains. To construct the probability tables, it accepts three-item lists in the form [State 1, State 2, Probability], where State 1 and State 2 are possible values, and Probability is the percent of the time that State 2 should follow State 1. **Prob** accepts integers that tell it what the last output (State 1) was; when a BANG is received, **prob** makes a constrained random choice between the possible State 2 values that have been specified by consulting its probability tables. The new State 2 value is output. In the simplest case, this output is fed back into the input, thereby becoming the next State 1 value. If **prob** encounters a State 1 value

38 Rowe's *Interactive Music Systems*, 188.

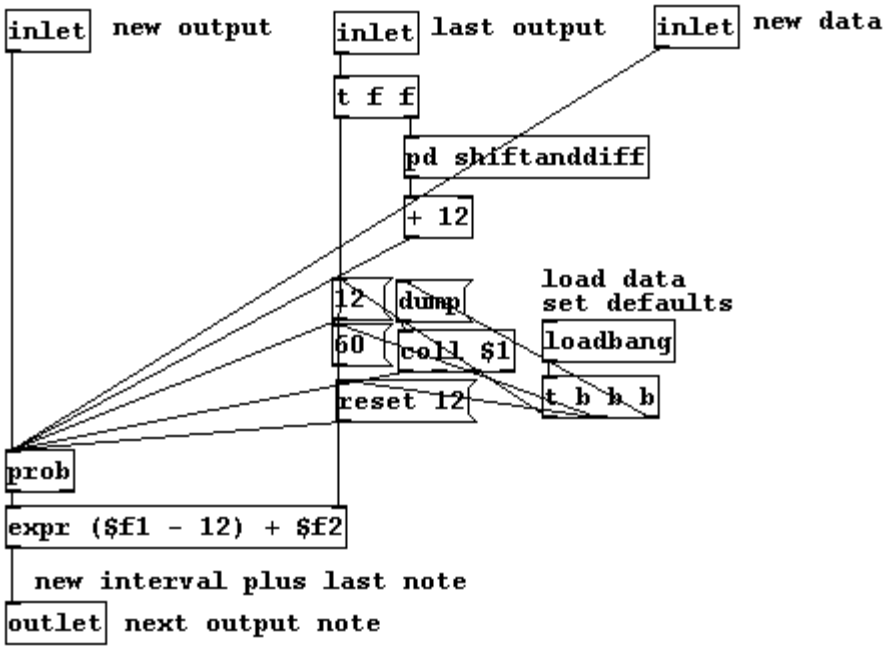
that it has no probability information for, it resets to a default value previously specified by the user.

PD also contains another object, called **anal**, which automates the construction of probability tables for use by **prob**. It accepts a series of integers and outputs the three-item lists described above that correspond to the input it has received. These lists can then be sent to **prob** for use in the construction of Markov chains.

I have created a group of object abstractions, called OSCAR2, which facilitates the use of first-order Markov chains for the generation of melodic material. OSCAR2 contains objects for analysis of melodic material both before a performance and during it, and objects for the creation in realtime of Markov chain output based on those analyses. OSCAR2 is the much-streamlined cousin of OSCAR,[39] a package of objects that I built for the creation of first- and second-order melodic and harmonic Markov chains. OSCAR and its extensive HTML documentation is contained in the CD-ROM Appendix to this paper.

OSCAR2's basic Markov-based melody generating patch, **blankHands**, is pictured in **Figure 21**. The object can accept one initialization

Figure 21: blankHands

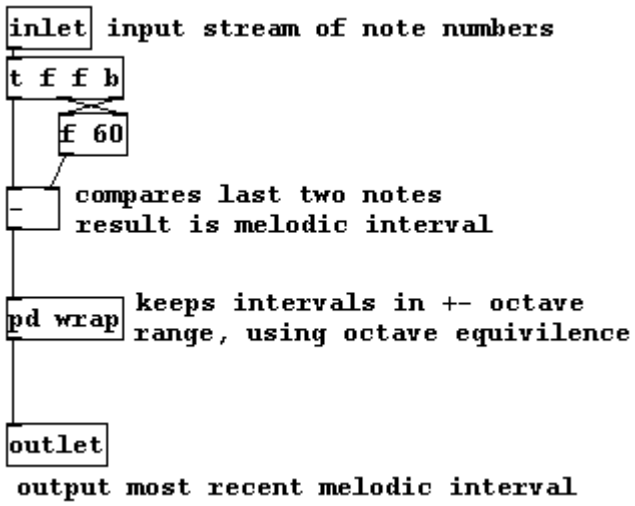


39 My OSCAR bears no relation to Peter Beyls' system of the same name.

argument, the name of a .coll (collection) file that contains previously compiled analysis information, which is loaded into the **prob** at loadtime. If no initialization argument is given, no analysis data is loaded. **BlankHands** has three inlets. The right inlet accepts new analysis info in the form of three-item lists, as described above. These lists are passed directly to the patch's **prob** object. The patch's middle inlet accepts the note number of the last note that was output by the system (referred to a *State 1* above). As mentioned, the most basic configuration would have the **blankHands**‘ output being fed directly back into the middle inlet as the last output. However, having this feedback occur outside of **blankHands** allows additional processing to being inserted between **blankHands**‘ output and the system's output; this processing could possibly result in changes in the output. Since the last note output is fed into **blankHands** from the outside, the patch can be kept "in the loop" of any output changes that occur.

Once a note is entered into the middle inlet, it goes to the subpatch **shiftanddiff**, shown in **Figure 22**, which calculates the melodic interval between the most recent note and the one that came before it. The interval is measured in half-steps, and is folded into a two-octave range, from descending octave (-12) through ascending octave (12). Zero (0) is a unison: the same note played twice. Since **prob** only accept state values that are greater than or equal to zero, 12 is added to the interval, making the range of state values sent to **prob** 0-24.

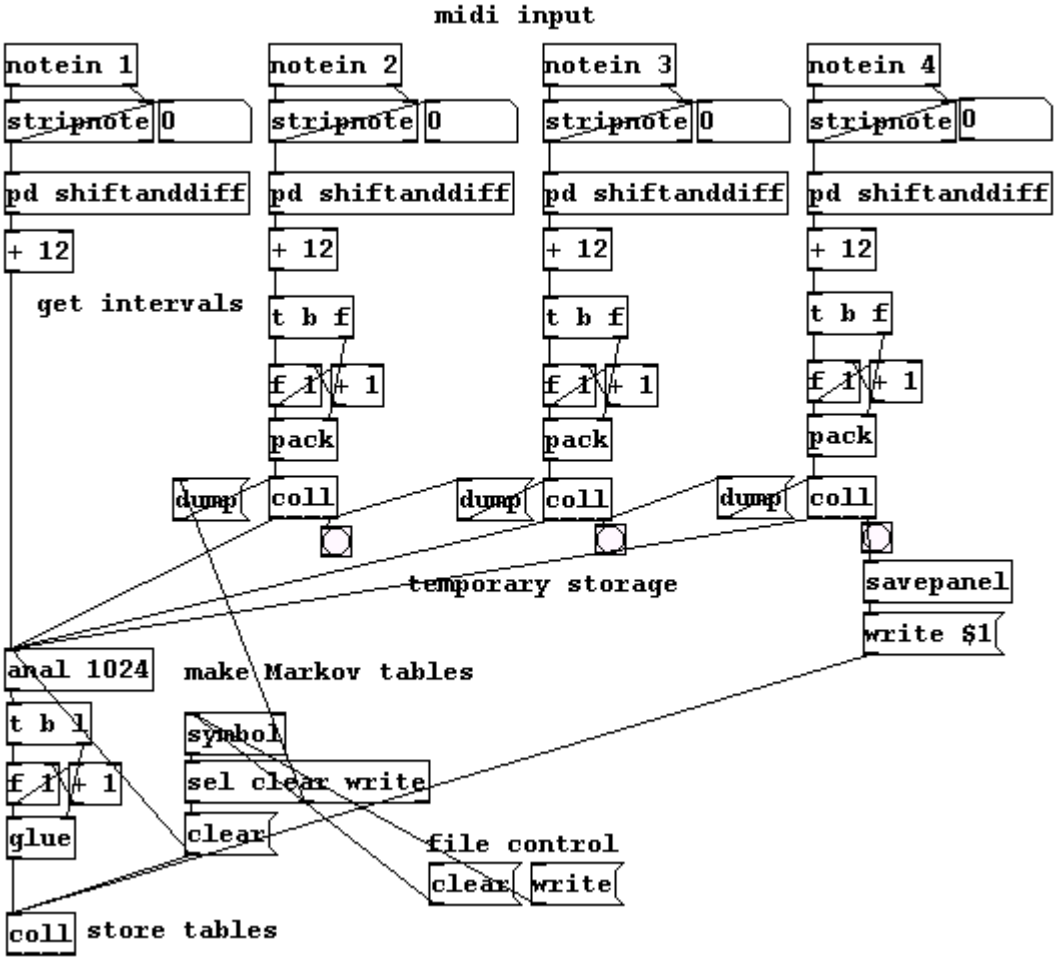
Figure 22: shiftanddiff



Lastly, **blankHands** accepts BANG messages in its left inlet, which cause the patch to output a new note. The BANG is sent to **prob**, which generates a new interval by consulting the probability tables specified by the most recent note input into **blankHands**' middle inlet. Twelve is subtracted from this interval (to compensate for adding twelve to all of **prob**'s input), and the result is added to the number of the last note output (the last note received in the middle inlet). The resulting note number is output from the patch.

The remainder of **blankHands** handles housekeeping tasks associated with initialization. The .coll file specified in the initialization argument is loaded, the last output note value is set to note number 60 (an arbitrary choice for a necessary decision—one must start somewhere), and the default reset state for the **prob** object is set to 12; thus, if **prob** encounters an interval that it has no data for, it outputs a 12, which is a

Figure 23: MIDIEars



unison. If no initialization argument was given, no .coll file is loaded, and **blankHands** starts with an empty **prob** object.

The OSCAR2 object called **midiEars**, pictured in **Figure 23**, automates the creation of analysis data before a performance. The patch accepts melodies on up to four MIDI channels simultaneously; is assumed that the input is polyphonic or poly-melodic, with each voice on a different channel. The incoming data is transformed into a string of melodic intervals by **shiftanddiff**, in the range described above, and stored in **coll** objects. After all of the MIDI data has been fed into **midiEars**, the user hits "write." This causes the **coll** objects to sequentially dump their strings of melodic intervals into the **anal** object, which outputs three-item probability lists, as described above. These lists are captured in another **coll** object, and are saved to a user-specified .coll file for future use in a **blankHands** object.

A similar object, **runtimeEars**, is designed to automate the analysis of a melody *during* a performance. It accepts a single stream of MIDI note numbers in its left inlet. It performs the same analysis as **midiEars**, except that is does it entirely in realtime. The probability lists created by **prob** are sent directly out of **runtimeEars**‘ outlet, where it can then be routed to a **blankHands** object for use in creating melodies. Simultaneously, the lists are sent to a **coll** object, so that they can be saved for future use. CLEAR, WRITE, and SAVE messages are accepted in **runtimeEars**‘ right inlet; these cause the **coll** to clear its contents, save its contents to a user-specified file, and save its contents to a default file, respectively.

Using the patches described above, we can construct a rudimentary melody generating system. **PatternGen** can supply the rhythm, with the tempo being controlled

by **tempoTracker**, and **blankHands** can provide the melodic material. Two versions of this idea are presented in **Figure 24** and **Figure 25**; the first version loads **blankHands** with a melodic analysis of several dozen Bach inventions and sinfonia, while the second version gets its analysis information by listening to the user's melodic input. Version 1 contains no more human→computer Influence than the varispeed wavetable player-follower that we started with, but it does allot the computer much more Participation, since the computer is actively deciding when and what notes to play. Version 2, on the other hand, involves, in addition to the increased Participation, much more human→computer Influence, since the computer's melodic output is based on analysis of the human's playing. From the standpoint of interactivity, this is our most developed system yet.

Figure 24: Example 1

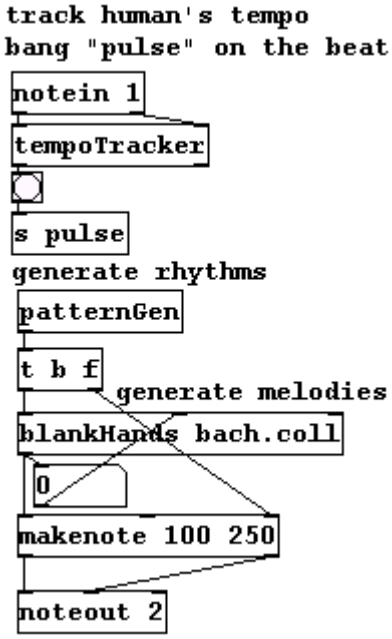
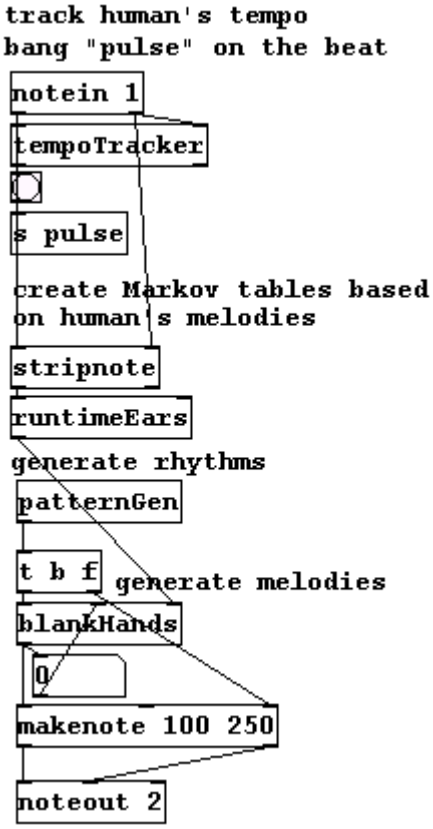


Figure 25: Example 2



5. Extended Influence: Chord Identification

The Need for More Context

Thanks to our realtime tempo tracker, our computer system is somewhat aware of the human's sense of musical time. The computer is not, however, at all aware of what the human is playing, in that it has no general musical sense of what is going on. It can look at what's happening on a note-by-note basis (and does so, to gather material for Markov chain melodies) but it lacks a more general sense of context that is needed for making music. Composers and improvisers do not generally think in individual notes, but rather in broader strokes, starting with basic chords, scales, and keys to begin their musical output and to keep it centered and coherent. It would be helpful for our computer to have a similar sense of context.

Giving the computer the ability to identify the chord that is being played by the musician would be a good first step toward achieving a sense of musical context. Any deeper sense of context (such as scale or key) could be built upon the reliable identification of chordal structures.

Like tempo tracking, chord identification proves to be no small task. Much research has been done in this area, and, as in tempo tracking, most systems that have been developed are either rule-based (symbolic) or connectionist (sub-symbolic). Both approaches once again have strengths and weaknesses that effect their ability to work in a live musical performance.

Rule-based systems

Robert Rowe discusses and implements several rule-based root and chord identifiers in *Machine Musicianship*.[40] The most simple and general identifiers are context-free: that is, they look only at the pitch information immediately available while a chord is being played, without reference or acknowledgement to the chord's surrounding rhythmic or tonal context.

Chord members are first reduced to pitch-class using a *modulo 12* operation (divide by twelve, throw out the dividend and keep the remainder). All C's, for example, become 0, since MIDI notes 48, 60, 72, 84, and all other C's equal 0 after a modulo 12 operation. Multiple occurrences of the same pitch-class are therefore reduced to a single, repeated note, and chord voicing is effectively eliminated from consideration of a chord's root and type.

A twelve-address pitch-class array is maintained to keep track of the pitch-classes that are present in a chord. A 1 is placed at a pitch-class's location in the array if a note of that pitch-class is present; otherwise, the pitch-class's location in the array is reset to 0. The number of chord members can be easily determined by summing the contents of the array.

The most basic chord identifier described by Rowe relies primarily on table lookup for determining chord root and type; this requires a complete listing of all possible chord types. The number of table entries is greatly reduced when the table lists chord intervals rather than actual pitch classes. There are, for example, two hundred and twenty distinct three-note chords, but these can be expressed using only fifty-five possible

40 Rowe's *Machine Musicianship*, 17-60.

combinations of two intervals.[41] Counting chords with between two and eleven members, there are 2,048 possible types of chords when they are expressed intervallically.

Once the chord members are converted from pitch to pitch-class, a method is called that calculates the interval of each chord member above the first active pitch-class in the array. This algorithm then compares the measured intervals to each successive entry in the lookup table. When a match is found, the chord type corresponding to that entry in the table is output, and the root is calculated. For example, if pitch classes 2 (D), 5 (F), and 11 (B) are turned on (set to 1 in the array), the measured intervals are [3 9], which is a minor chord in first inversion. A minor chord in first inversion has a root of 9 (A) when pitch class 0(C) is the lowest chord member (this information would be stored in the lookup table). Since the lowest member in this chord is pitch class 2 (D), 2 is added to the default root of 9, resulting in 11, so the chord is B minor in first inversion.

The lookup table in Rowe's chord identifier contains only the most common chord types (mostly three- and four-note chords). When a chord that is not listed in the lookup table is encountered, an algorithm is called that removes the most dissonant chord members one by one until the chord matches an entry in the lookup table. A member's "dissonance" is calculated by summing the member's intervallic relationship (expressed in half-steps) with all other chord members; the member with the smallest total intervallic measurement is considered the most dissonant, and is removed.[42]

Rowe implements a separate algorithm that independently identifies a chord's type once its root has been determined. It works by looking for minor and major thirds

41 Rowe's *Machine Musicianship*, 21.

stacked above the root. The algorithm looks for a third, then a fifth, then a seventh, and so on, until all chord members have been identified. Chord type is then expressed as a function of its members, e.g. D with a minor third, a perfect fifth, and a minor seventh. This algorithm can be coupled with a separate root identifier (such as the one described above) to complete successful chord identification.[43]

More complex rule-based chord identifiers begin to model human grouping and separation mechanisms, and therefore begin to depend partially on the context in which a chord is found. David Temperley's Serioso Music Analyzer uses preference rules (similar to those proposed by Lerdahl and Jackendoff, as discussed earlier) to identify and correctly spell chords and larger harmonic areas. These preference rules describe preferred root relationships, as well as the influence of metric relationships on chord identification. One rule states, for example, that new chords tend to be started on strong beats. Another says that roots of nearby chords tend to be close to each other on the "line of fifths" (in other words, they tend to be "closely related" in the strong theoretical sense).[44] Richard Parncutt extends this psycho-musical model further by referring to more contextual information such as chord voicing and underlying tonality to help determine chord roots. Parncutt's model is deeply rooted in research in Music Cognition, and includes such psychoacoustical concepts as virtual pitch, pitch salience, and root stability. Parncutt and Temperly's systems work primarily to identify a chord's root; they

---

42 Rowe's *Machine Musicianship*, 38-89.

43 Ibid., 43-44.

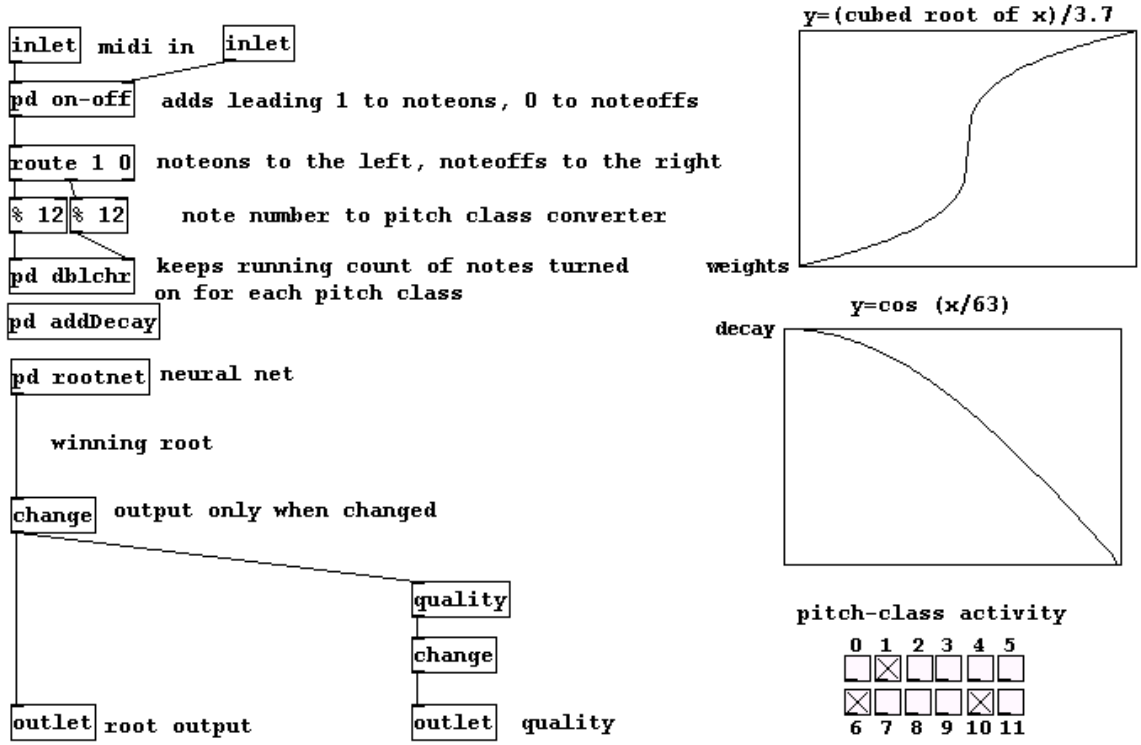
44 Ibid., 45-47.

then calculate additional information about the chord, such as its mode or type, using the thirds-stacking method described above.[45]

A connectionist chord root identifier

I have developed a sub-symbolic chord route identifier called **chords**. It uses a neural net to determine the root of arbitrary chords in realtime. The chords can be blocked or arpeggiated. Once the root is identified, the basic chord type (major, minor, diminished, or augmented; diminished, minor, major, or augmented seventh) is identified using a rule-based stacked-thirds approach similar to the one described above.

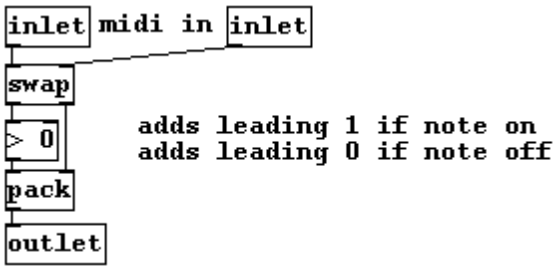
Figure 26: chords parent patch



45 Ibid., 49-60.

The parent patch of the chord identifier is pictured in **Figure 26**. MIDI note data enter at the top of the patch. The first step in root identification is to reduce chord members to pitch-class, and to construct a way to keep track of which pitch-classes are currently being played; this is achieved in the top portion of the parent patch. The **on-off** object, pictured in **Figure 27**, attaches a leading 1 to all note-on's, and a leading 0 to all note-off's. The following **route** object directs all note-on's to the left, and all note-off's to the right. In either case, the messages move into a **% 12** (modulo 12) object; this converts all pitches from MIDI note number to pitch-class, where all C's = 0, Db = 1, etc. As mentioned above, this allows all notes to be treated with octave equivalence, which removes multiple occurrences of the same pitch class and chord voicing from consideration by the chord identifier.

Figure 27: on-off



The subpatch **dblchr**, short for double-checker, is pictured in **Figure 28**. This patch ensures that a pitch-class is considered "active" when one or more notes of that class are playing, and is turned to "inactive" only when all notes of that class have stopped playing. Note-on's enter the left inlet, note-off's come in the right inlet (from their respective modulo objects). The dual **select** objects send a BANG to the correct **pccount** (pitch-class count) object when a note message is received; note-on's cause a BANG to be sent to the **pccount**'s left inlet, note-off's bang the **pccount**'s right inlet. **Pccount**, pictured in **Figure 29**, keeps a running count of the number of activated notes of a single pitch-class. There is one **pccount** object for each pitch-class.

Figure 28: dblchr (double-checker) subpatch

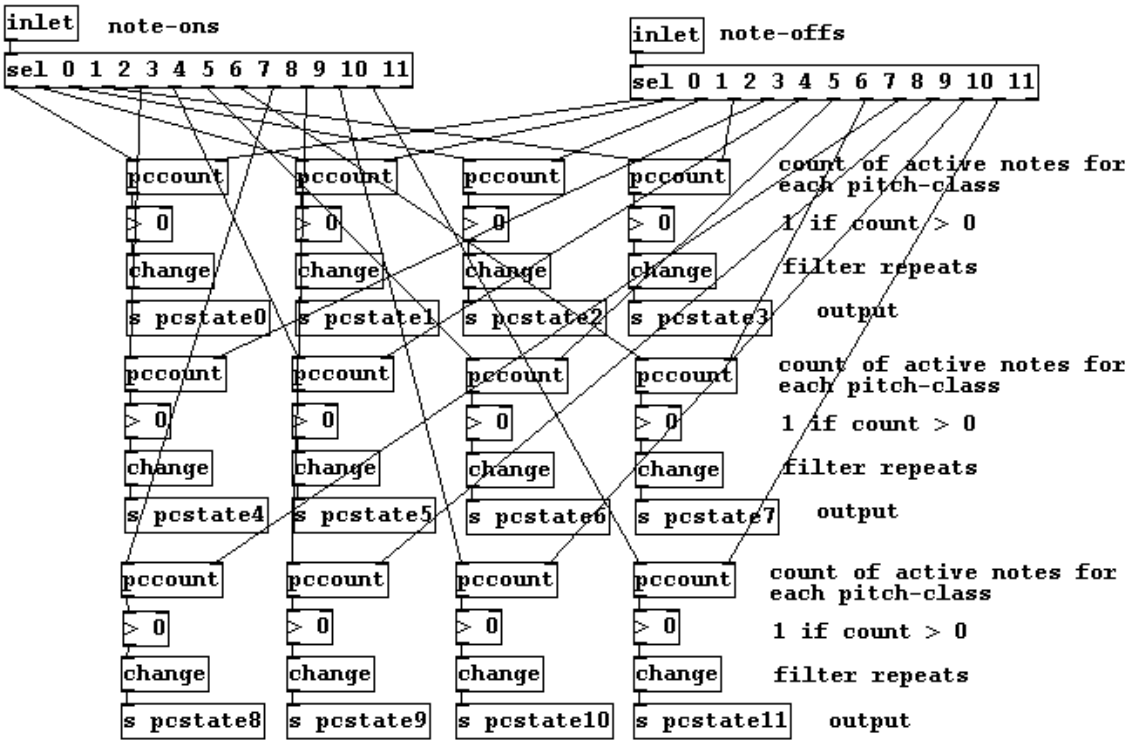
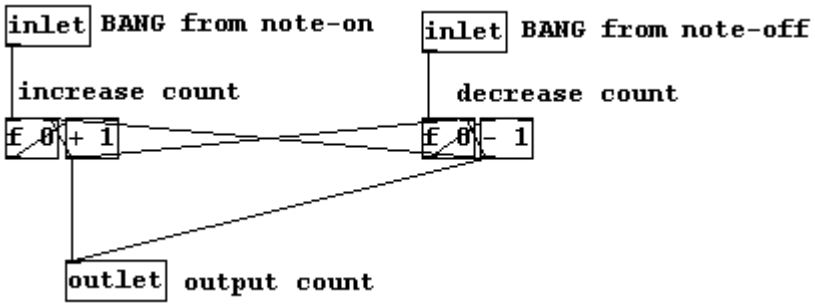


Figure 29: pccount subpatch



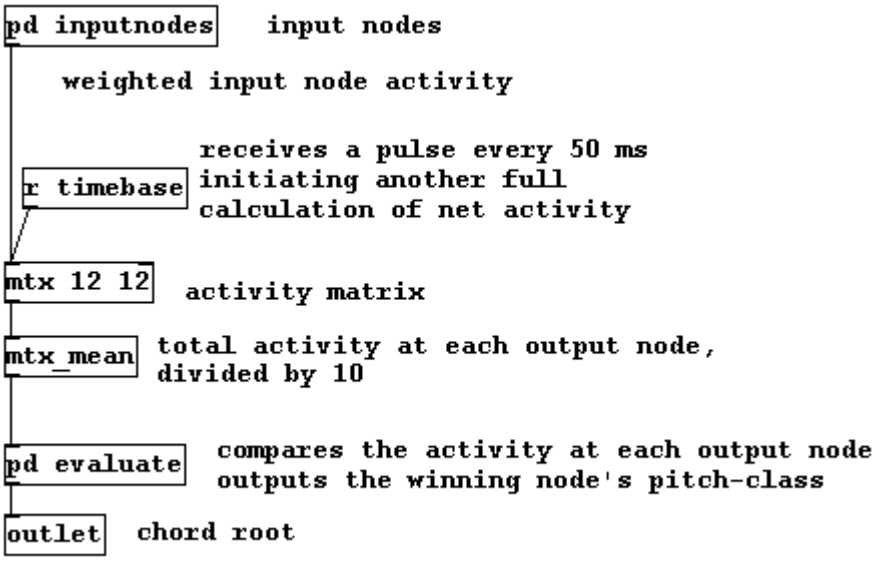
For each pitch-class, the active note count is output from **pccount** into a **>0** object. If the note count for that pitch-class is greater than 0, a 1 is output; otherwise, a 0 is output. The result is sent through a **change** object, which passes the input only if it is different from the previous input. Therefore, although pitch-class count can change rapidly as chords are played, output will only be sent from **change** when the pitch-class changes to inactive or active; repeated 1's and 0's are filtered out. Finally, the output

from each **change** is sent to a **s pccount** (short for send pitch-class state) object, which broadcasts the state of that pitch class to anywhere that has a **r pccount** (short for receive pitch-class state) object. There is one **s pccount** object for each pitch-class; they are differentiated by their trailing integer, so the state of pitch-class 0 is sent out via **s pccount0**, pitch-class 1 is sent out via **s pccount1**, and so on. A complete set of **r pccount** objects exists inside the GUI toggles on the **chords** parent patch, providing the user with a graphic representation of pitch-class activity (a box is checked when its pitch-class is active). The status of each pitch-class is also sent to the neural net, which has one input node for each pitch-class.

Output nodes

The bulk of the chord identifier's processing, including the neural net, is contained within the **rootnet** subpatch, pictured in **Figure 30**. In the most basic terms, **rootnet** consists of input nodes (one for each pitch-class); output nodes (again, one for each pitch-class); a 12x12 matrix which contains the weights for the connections between each input node and

Figure 30: rootnet



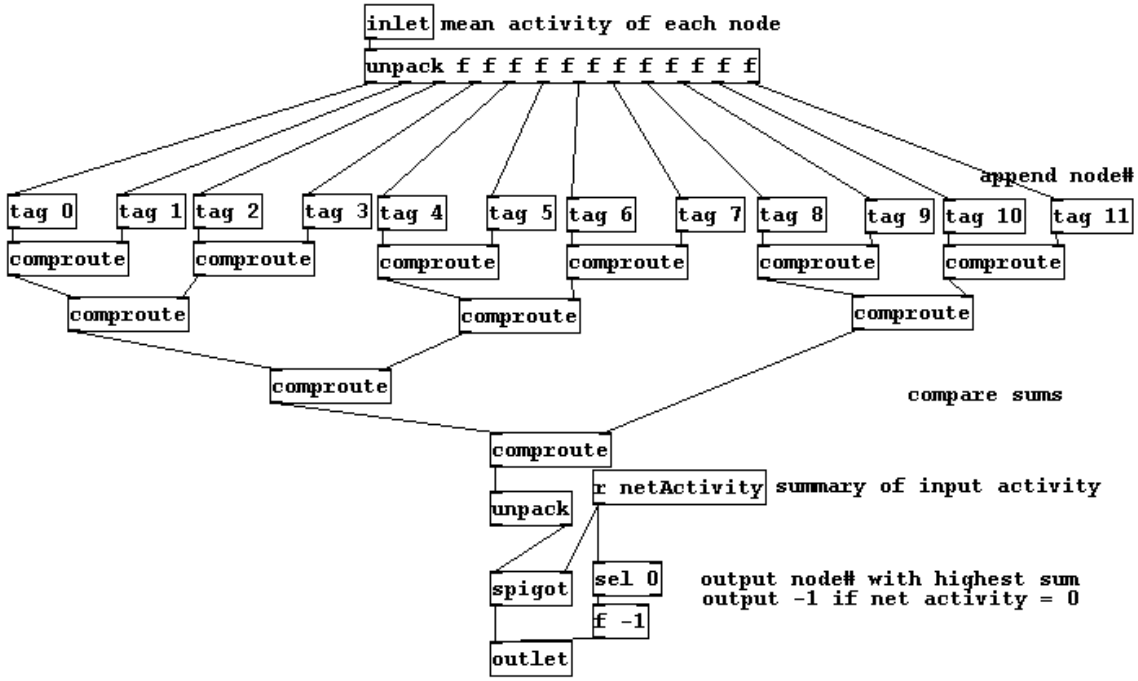
each output node; another 12x12 matrix that contains the ongoing activity between each input node and each output node; a method for summing the activity at each output node; and a method for comparing the activity at each output node and selecting a winner. The node with the highest sum of activity is declared the winner; the pitch-class that it corresponds to is output as the chord root.

The heart of the neural net is the activity matrix, a 12x12 table that summarizes the connections between the input and output nodes. Rows 1-12 contain activity coming from input nodes 0-11 (corresponding to pitch-classes 0-11), and columns 1-12 list the activity at output nodes 0-11 (corresponding to chord roots with pitch class 0-11). At each location in the matrix is a number that is equal to the activity at the row's input node (which will be between 0 and 1) times the weight on the connection between the row's input node and the column's output node (which will be between -1 and 1). To calculate the value at [3, 3], for example, one looks at the activity at input node 2 (let's say it is 1) and at the weight between input node 2 and output node 2 (which we'll say is .5), so the value at this location will be 0.5. The values in the activity matrix are continually updated as notes are turned on and off.

Every 50 milliseconds, a pulse is received through **r timebase**; this remote "metronome" synchronizes ongoing analysis processes. The pulse (actually a BANG message) is sent to the matrix, causing it output its contents into the **mtx_mean** object, which calculates the arithmetic mean of each column. The mean is then multiplied by 12; the result is a quick and efficient calculation of the sum of each column, and a total picture of the activity at each output node.

Once the total value of each output node is calculated, the **evaluate** subpatch (pictured in **Figure 31**) compares them, selects the highest sum, and outputs the node number of the winner. First, the **tag** subpatch appends the node number prepended to the node's activity. The resulting [*sum*, *node#*] packages are sent through a series of **comproute** objects. Each **comproute** (**Figure 32**) receives one of these packages in each of its two inlets. The two packages are unwrapped and their *sums* are compared; the higher *sum* is repacked with its *node#* and sent out the outlet. At the bottom of the cascade of **comproute** objects, the package with the highest *sum* is output. The *node#* of the winner is output from the patch.

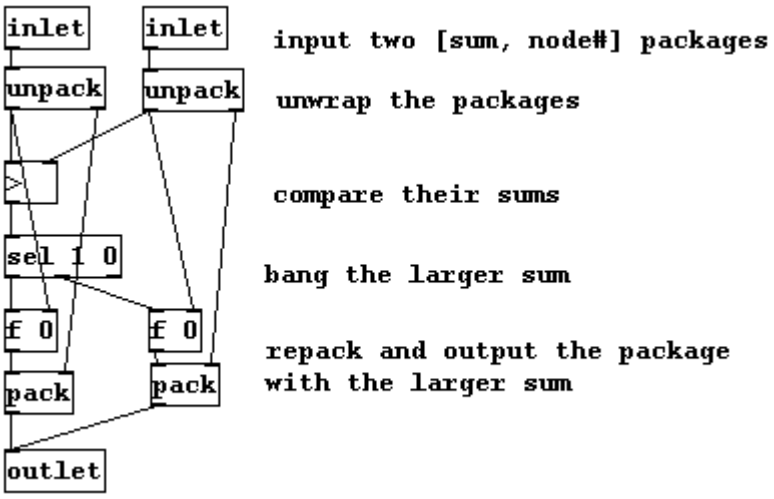
Figure 31: evaluate subpatch



There is one special case, handled at the bottom of the **evaluate** patch: if there is no input to the net, a node will still be picked arbitrarily as the winner. In this case the value coming from **r netActivity** will be 0, so output from the net is blocked and -1 is

output as the root, to indicate that no input is being applied to the net (no notes are being played).

Figure 32: comproute



Input nodes

Imagine that note number 60 (middle C) is played, causing pitch-class 0 and its associated input node to be activated. The node outputs a 1, which is sent to every output node. Each output node has an activity level of 1.

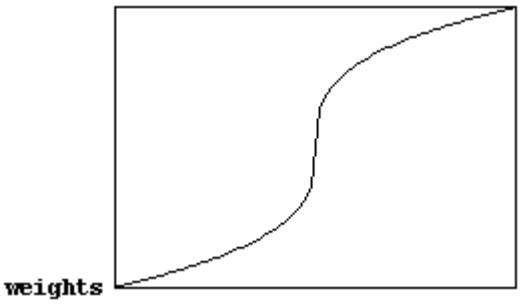
Clearly, this example could not produce intelligent behavior, as any input would cause all output nodes to be equally activated. What we want is for an active input node to activate certain output nodes more than others. In other words, we want the connections between input and output nodes to be weighted, with strongly correlated connections having higher numerical weights than uncorrelated connections. To use a musical example, we want the connections between the input node corresponding to C and the output nodes corresponding to C, F, and Ab to receive strong weights (because the note C is a likely member of C, F, and Ab chords), and the connections between input-C and output-B and output-Db to receive weak or negative weights (since C is not a natural member of B and Db triads).

Each connection between an input node and an output node will have its own individual weight; 12 input nodes connected to each of 12 output nodes equals 144

separate connections with 144 separate weights. The management of these weights is handled by another 12x12 matrix, called **address**; the contents of the matrix is stored in a file called **address.mtx** (see **Table 1**).

Each location in the matrix holds an integer between 0 and 100. These integers are not the connection weights themselves, but rather are indexes to a function stored in an array called **weights**, which is pictured in **Figure 33**. This graph is of the function $y = \sqrt[3]{x}/3.7$, which is the basic equation $y = \sqrt[3]{x}$ scaled so that the y values -1 to 1, inclusive, fit within a 101-address array (50 negative value, 50 positive values, and 0). The combination of the matrix address and the **weights** array cause the connection weights to be a function of $y_{[-1,1]} = \sqrt[3]{x-50}/3.7$. The calculated weights are listed in **Table 2**. The connection weights themselves could very easily be stored within the matrix; the reason they are not has to do with the backpropagation algorithm that allows the neural net to automatically configure its own weights. This algorithm will be discussed in detail shortly.

Figure 33: weights array



The **loadweights** subpatch pictured in **Figure 34** is responsible for loading the weights for a single input node at loadtime (when the patch is first loaded). The input node number being loaded is entered in the inlet; the row of the **address** matrix corresponding to that node number is output; these numbers are indexes to the **weights** array. The weights are read from the array, packed, and output from the patch.

Table 1: weight addresses, as stored in address.mtx

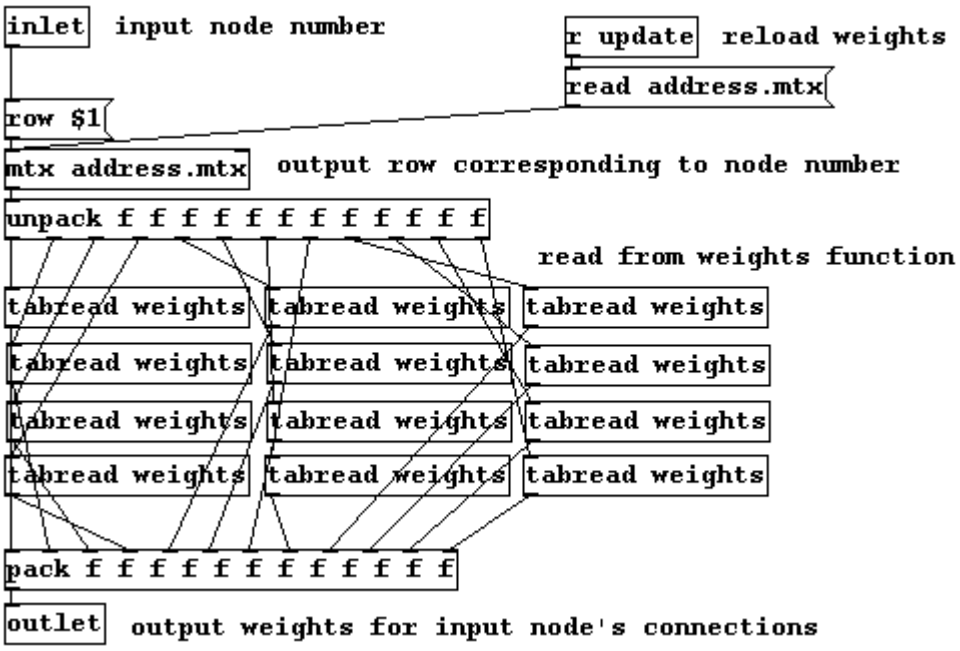
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	100	19	54	1	1	93	64	42	88	75	1	1
1	1	100	1	56	16	11	70	60	1	94	57	61
2	44	1	100	54	51	27	1	89	64	1	63	80
3	87	15	1	98	70	42	1	1	84	61	11	67
4	95	91	46	1	97	49	35	1	1	95	83	1
5	12	95	94	28	1	100	29	46	27	18	97	81
6	53	1	86	60	44	1	90	50	44	1	1	83
7	50	51	1	57	51	49	1	74	50	48	1	1
8	49	89	82	1	89	86	18	1	100	14	84	1
9	34	52	77	91	1	74	90	2	42	99	1	72
10	57	28	51	91	83	44	73	67	12	1	100	8
11	58	48	23	62	76	82	1	54	75	12	1	100

Table 2: calculated connection weights

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0.99568	-0.849022	0.429027	-0.989002	-0.989002	0.946864	0.65139	-0.540541	0.908642	0.790275	-0.989002	-0.989002
1	-0.989002	0.99568	-0.989002	0.491114	-0.875571	-0.916544	0.733626	0.58228	-0.989002	0.954148	0.517008	0.601076
2	-0.491114	-0.989002	0.99568	0.429027	0.27027	-0.768613	-0.989002	0.916544	0.65139	-0.989002	0.635496	0.839793
3	0.900601	-0.884072	-0.989002	0.982227	0.733626	-0.540541	-0.989002	-0.989002	0.875571	0.601076	-0.916544	0.694941
4	0.961322	0.931951	-0.429027	-0.989002	0.975358	-0.27027	-0.666544	-0.989002	-0.989002	0.961322	0.866901	-0.989002
5	-0.908642	0.961322	0.954148	-0.757308	-0.989002	0.99568	-0.745655	-0.429027	-0.768613	-0.858055	0.975358	0.849022
6	0.389797	-0.989002	0.892413	0.58228	-0.491114	-0.989002	0.924311	0	-0.491114	-0.989002	-0.989002	0.8669010
7	0	0.27027	-0.989002	0.517008	0.27027	-0.27027	-0.989002	0.779594	0	-0.340519	-0.989002	-0.989002
8	-0.27027	0.916544	0.858055	-0.989002	0.916544	0.892413	-0.858055	-0.989002	0.99568	-0.892413	0.875571	-0.989002
9	-0.681038	0.340519	0.810811	0.931951	-0.989002	0.779594	0.924311	-0.982227	-0.540541	0.989002	-0.989002	0.757308
10	0.517008	-0.757308	0.27027	0.931951	0.866901	-0.491114	0.768613	0.694941	-0.908642	-0.989002	0.99568	-0.939467
11	0.540541	-0.340519	-0.810811	0.618764	0.800675	0.858055	-0.989002	0.429027	0.790275	-0.908642	-0.989002	0.99568

When a pitch-class is activated, it should send these weights to the activity matrix. This is a simple matter: one needs only to prepend the correct row and column numbers to each weight and send the resulting package to the matrix. When the pitch-class is deactivated, the row corresponding to the turned-off pitch-class should be filled with zeros. Since an active pitch-class is indicated by a 1, and an inactive pitch-class is indicated by a 0, one needs only to multiply each weight by the proper pitch-class activity and then send the result to the matrix.

Figure 34: loadweights subpatch



Suppose, though, that one wants to play arpeggios and have them correctly identified as chords. Our current setup will not work in this case, since the neural net has no "memory"; once a note is released, it is essentially forgotten by the net. What's needed is a way to extend the "presence" of a pitch class over a short period of time, so that a note can interact with notes that come later.

This functionality is provided by the **decay** subpatch, pictured in **Figure 35**. This patch applies a "decay" function (**Figure 36**) when a pitch-class is deactivated, causing

Figure 35: decay subpatch

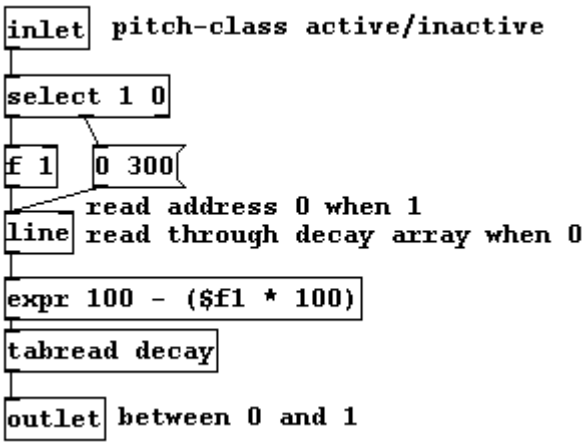
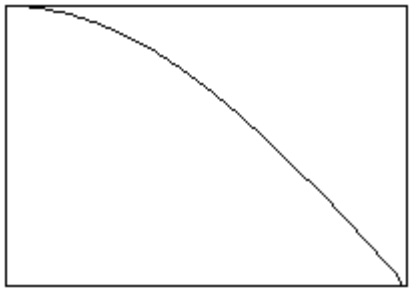


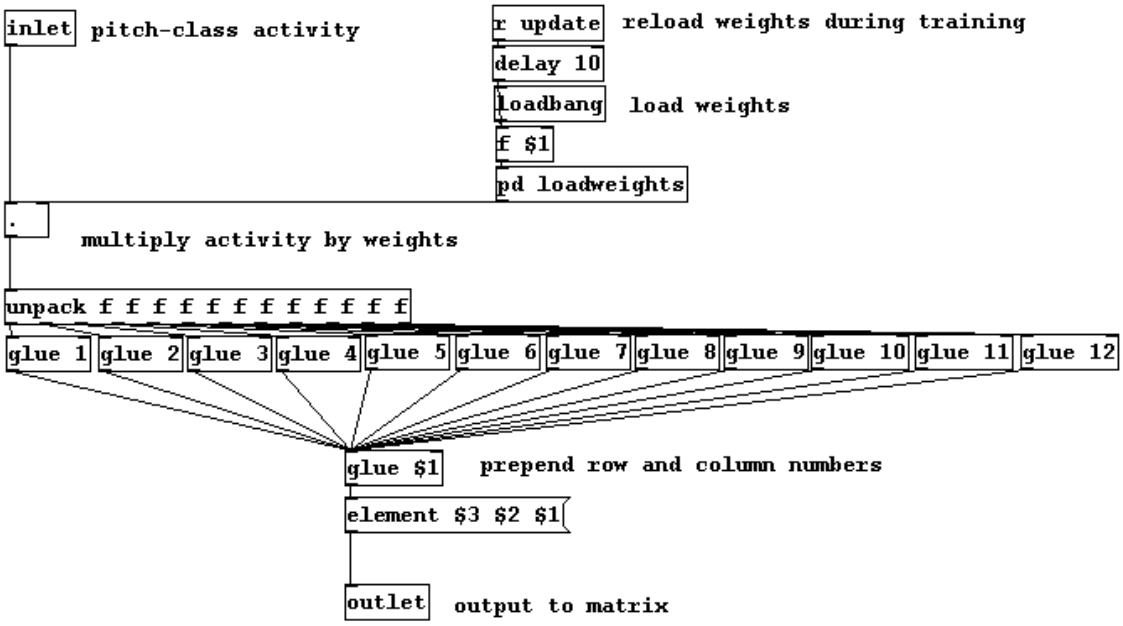
Figure 36: decay function



the pitch-class's activity to be extended for a short period of time. The decay function moves from 1 to 0 over the course of 300 milliseconds; notes will have significant interaction with notes that come immediately afterward, but will then have an increasingly smaller effect on subsequent notes. The decay is not linear, but rather is sloped, so that activation decreases slowly at first, and then rapidly. The function is derived from $y = \cos(x)$, scaled so that $y_{[0,1]}$ will fit in a 100-point array, resulting in $y_{[0,1]} = \cos(x/63)$. The **decay** subpatch outputs values from the **decay** array; it reads from address 0 (x =1) whenever the pitch-class is active, and then reads through the remaining 99 addresses of the array when the pitch-class is deactivated.

With the decay operation in place, the structure of our input node algorithm is complete, and is pictured in **Figure 37**. The connection weights are calculated at loadtime, and remain static as the net is being operated. The weights are multiplied by the pitch-class activity, after it is processed through the decay function, which extends the influence of pitch-classes beyond the point that they are deactivated. The result is the fluctuating activity that is sent to the activity matrix.

Figure 37: inputnode



As each input node's activity is being continuously updated, the sum of all activity at each output node is calculated by summing each column in the activity matrix. The sum for each output node is compared, and the node with the highest sum is selected. The node number of this winner is output as the chord root. This output is sent through a **change** object, which again filters successive repeats of the same number, and then out of the patch. The root is also sent to the **quality** subpatch, which determines the chord's basic type. The **quality** subpatch is discussed in detail shortly.

Backpropagation

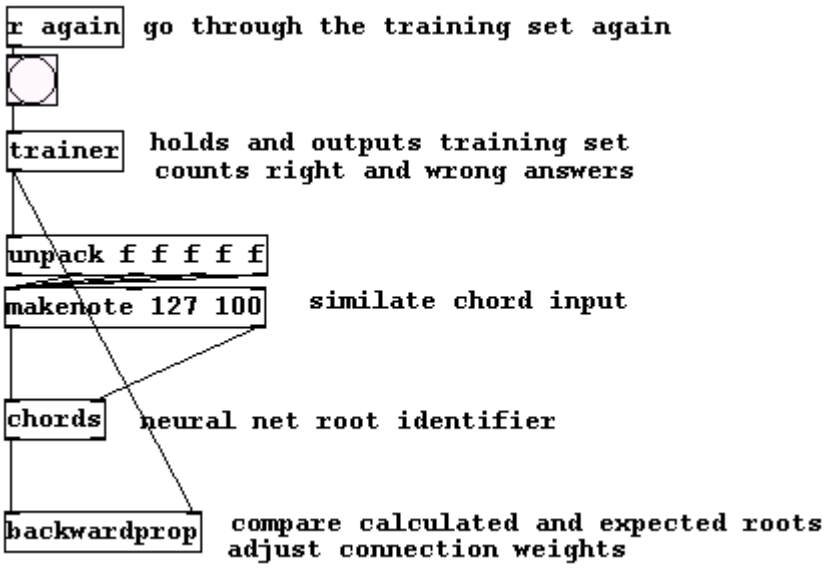
It is clear that the connection weights are the most essential part of our connectionist root identifier; if they are not set properly, the net will not function correctly. With 144 connection weights to set, each with 100 possible values, one is faced with a large number of possible combinations of weight values. Luckily, neural

nets can essentially "learn their own rules" by setting their own connection weights through the process of backpropagation.

To begin backpropagation, one constructs a training set, which consists of examples of the expected patterns along with the output that is to be associated with that pattern. In the case of our root identifier, our training set will consist of a list of spelled-out chords along with the correct root of each chord. The training set is presented item-by-item (chord-by-chord) to the neural net, whose connection weights are set to random values or are zeroed out. For each training item, the net's output is compared to the correct output listed in the training set. If they are the same, then the net got the answer correct and nothing more need be done. If, however, the training set's answer and the neural net's answer are different, the connection weights are adjusted. Specifically, the connections between the input nodes that were activated during the training exercise and the output node that erroneously received the highest sum are adjusted in the negative direction, and the connections between the activated input nodes and the output node that should have won are adjusted in the positive direction. A musical example will help to clarify the process. If the training exercise is a C major chord, spelled out C-E-G, and the net identifies this incorrectly as having a root of Bb, then the following weight adjustments are made: the connections between input nodes C, E, and G, and output node Bb are adjusted in the negative direction, and the connections between input nodes C, E, and G, and output node C are adjusted in the positive direction. After the adjustments are complete, the net continues on with the next training exercise. When the net makes it through the entire training set without getting any answers incorrect, training is complete.

The backpropagation algorithm for our chord root identifier is called **trainingsession**, and is pictured in **Figure 38**. The **trainer** subpatch holds the training set, which consists of all major, minor, diminished, and augmented chords, with all common varieties of 7th chords.

Figure 38: trainingsession



The set is stored in the form of a text file; it is contained in the Appendix of this paper. One by one, **trainer** outputs exercises from the training set. The chord members are sent to the **makenote**, which plays the chord in MIDI notes and sends them to the chord identifier. The **backwardprop** subpatch compares the identifier's output with the expected output listed in the training set, and makes any necessary weight adjustments. **Trainer** then updates its running count of correct and incorrect answers and then proceeds with the next training exercise.

Figure 39: Backwardprop

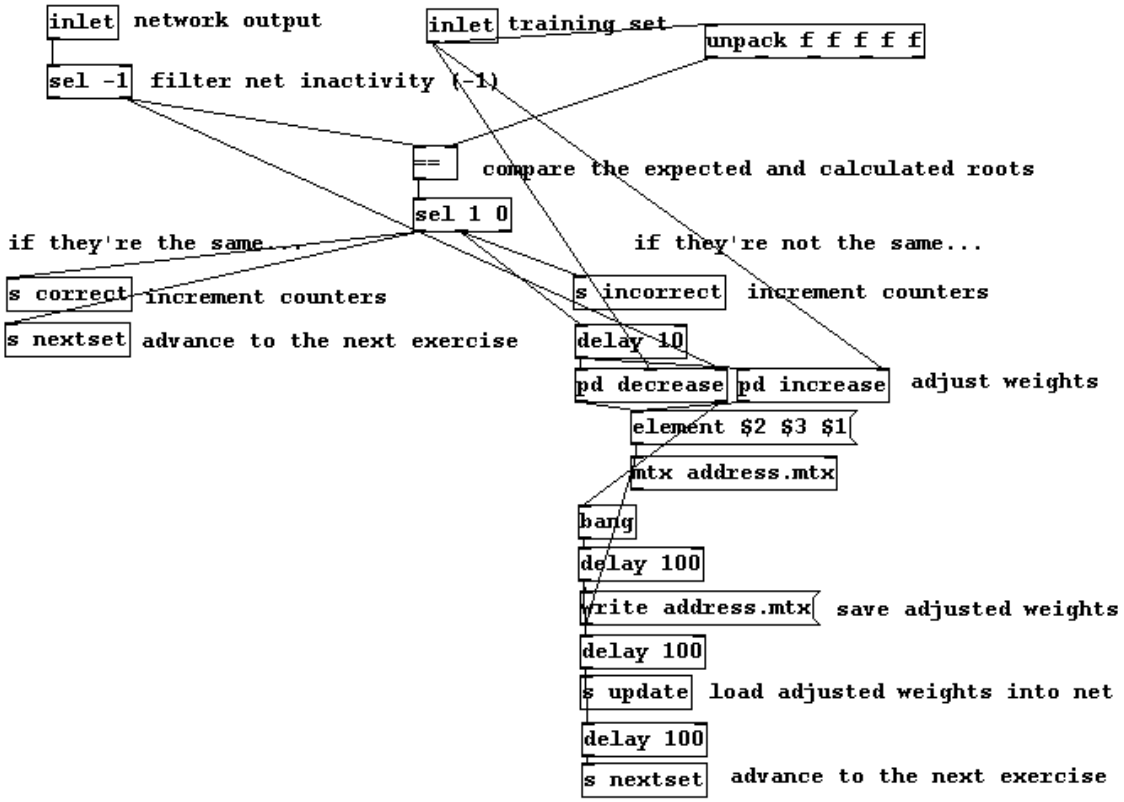
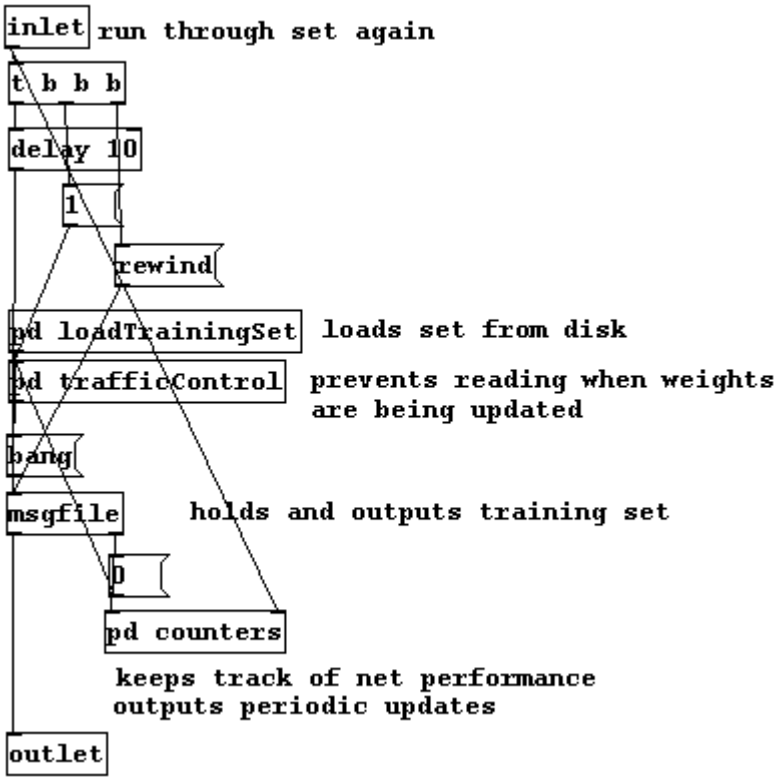
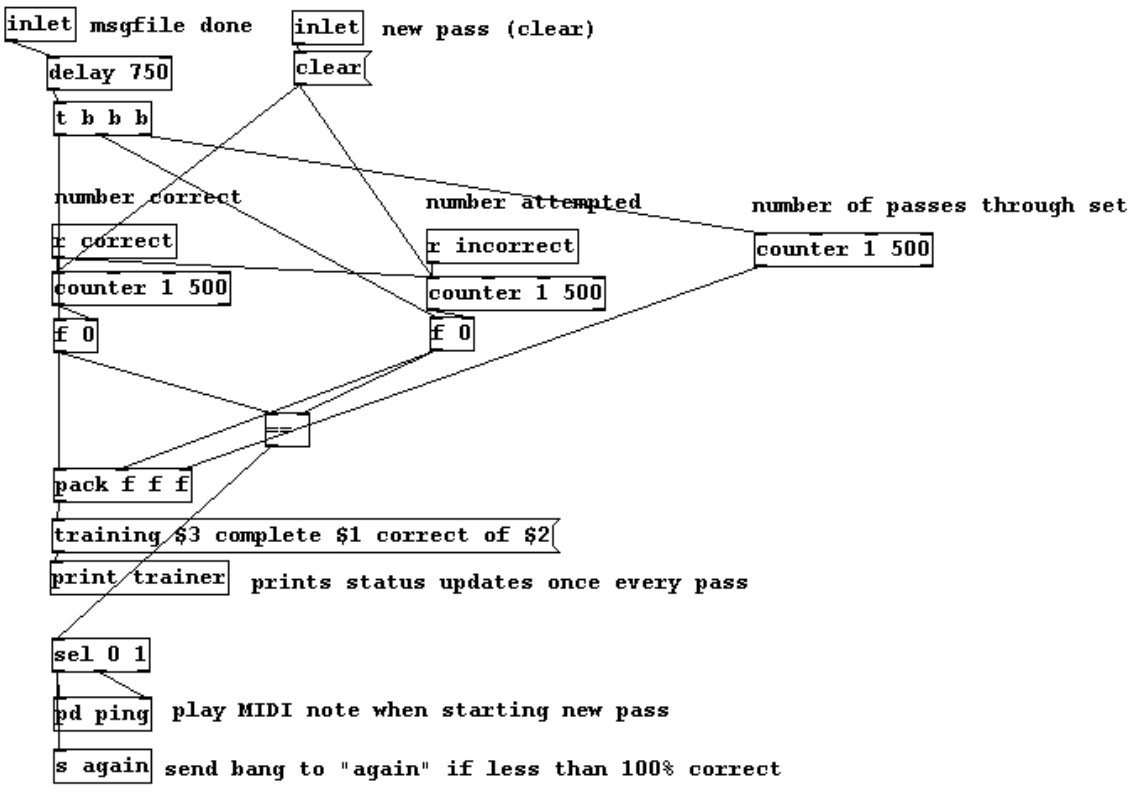


Figure 40: trainer



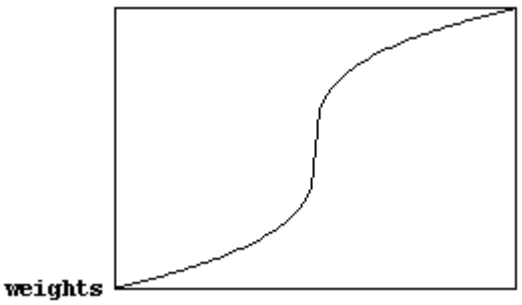
Backwardprop and **trainer** are pictured in **Figures 39 and 40**. The bulk of **backwardprop** deals with updating the weight addresses in the **address** matrix, saving the updated matrix to disk, and loading the new weight addresses into the root identifier. **Trainer** deals primary with outputting the training set and with keeping counts of right and wrong answers. The **counters** subpatch, pictured in **Figure 41**, maintains counts of the number of training exercises attempted, the number answered correctly, and the number of passes that have been made through the training set. Importantly, these counts are printed for the user (and a MIDI note is sounded) at the start of every training pass. This keeps the user informed of the trainer's progress, an important consideration since the algorithm is designed to repeat indefinitely until the net gets 100% of the training exercises in the set correct.

Figure 41: counters



The unique shape of the **weights** array (see **Figure 42**), which is the "learning curve" of this neural net, has a great effect of the backpropagation process. Training started with weights set to 0, at the center of the graph. As the weights begin to be adjusted, they change drastically, since the slope near x = 0 is quite steep. As training continues and the weights approach -1 and 1, the adjustments become more subtle and refined, since the slope of the learning curve is less extreme as you get further from 0. The rational for storing addresses to the weight array, and not the weights themselves, becomes clear: it is far simpler to adjust an integer array address by 1 or -1 than it is to adjust a floating-point weight from one arbitrary value to another (from 0.989002 to 0.99568, for example).

Figure 42: weights array



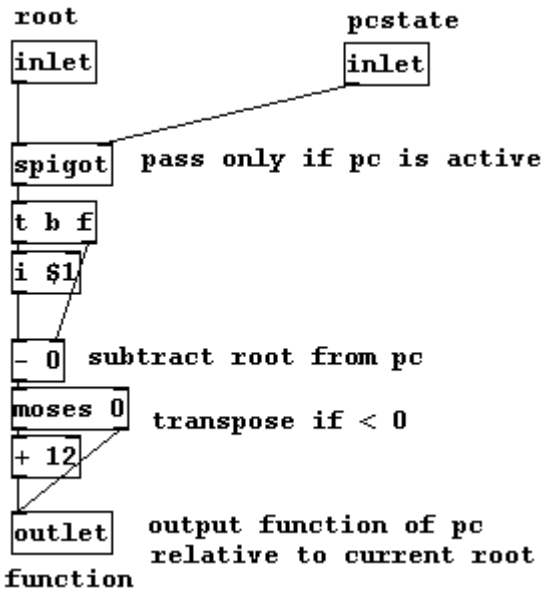
Chord type identification

Once a chord's root is identified, this information can be used to facilitate the identification of the chord's type or quality. Several methods of doing this have already been outlined; the one that I have implemented in PD bears some resemblance to Robert Rowe's thirds-stacking algorithm. My chord identifier is a bit simpler, in that it looks for the presence of notes that function in specific ways relative to the identified root. It makes no attempt to identify the role of each and every note in the chord.

First, chord members are classified with regards to their height above the root; if the root is G, the pitch-class D is classified as 7, since it is 7 half-steps above the root. This process is akin to the "moveable do" solfège system that is used in sight-singing

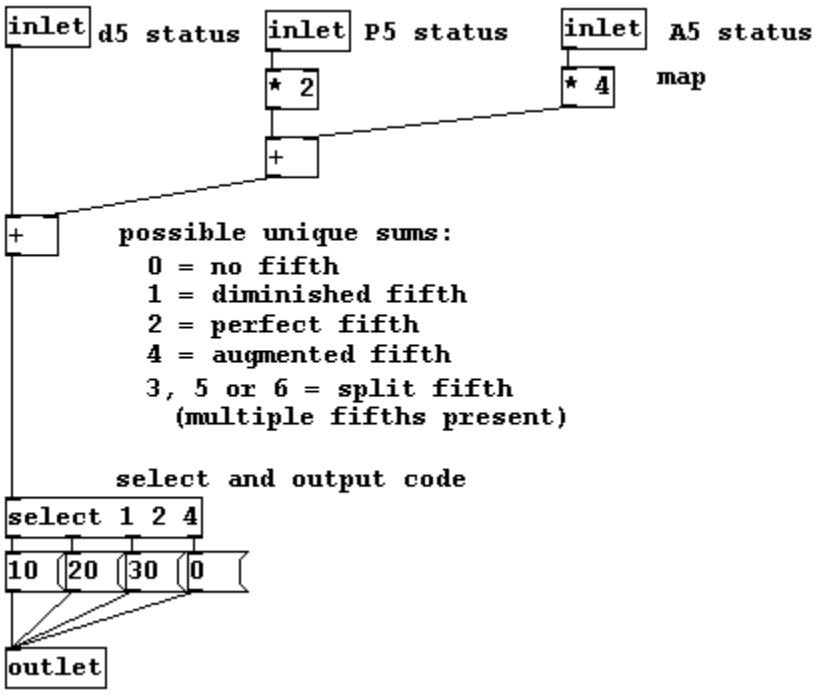
exercises, in which the solfège syllable (do, re, mi, etc.) associated with a specific note changes according to the key or tonal center of the exercise. The **functioncalc** subpatch (**Figure 43**) performs this operation by subtracting the root number from all active pitch-class numbers, and then adding 12 if the result is less than 0. The results are summarized in a 1x12 matrix: the 12 columns in the matrix represent the 12 possible chord member functions (root, m2, M2, etc.). A 1 is placed in a column if that chord member is present, and a 0 is placed in a column if that chord member is not present.

Figure 43: functioncalc



The matrix is then examined for the presence of specific chord members. The subpatch in **Figure 44**, for example, tries to determine what kind of fifth is present in the chord. This patch looks for the exclusive presence of one type of fifth or another. In its inlet it receives the status flags (0 or 1) for chord members 6 (diminished fifth or

Figure 44: look for 5ths

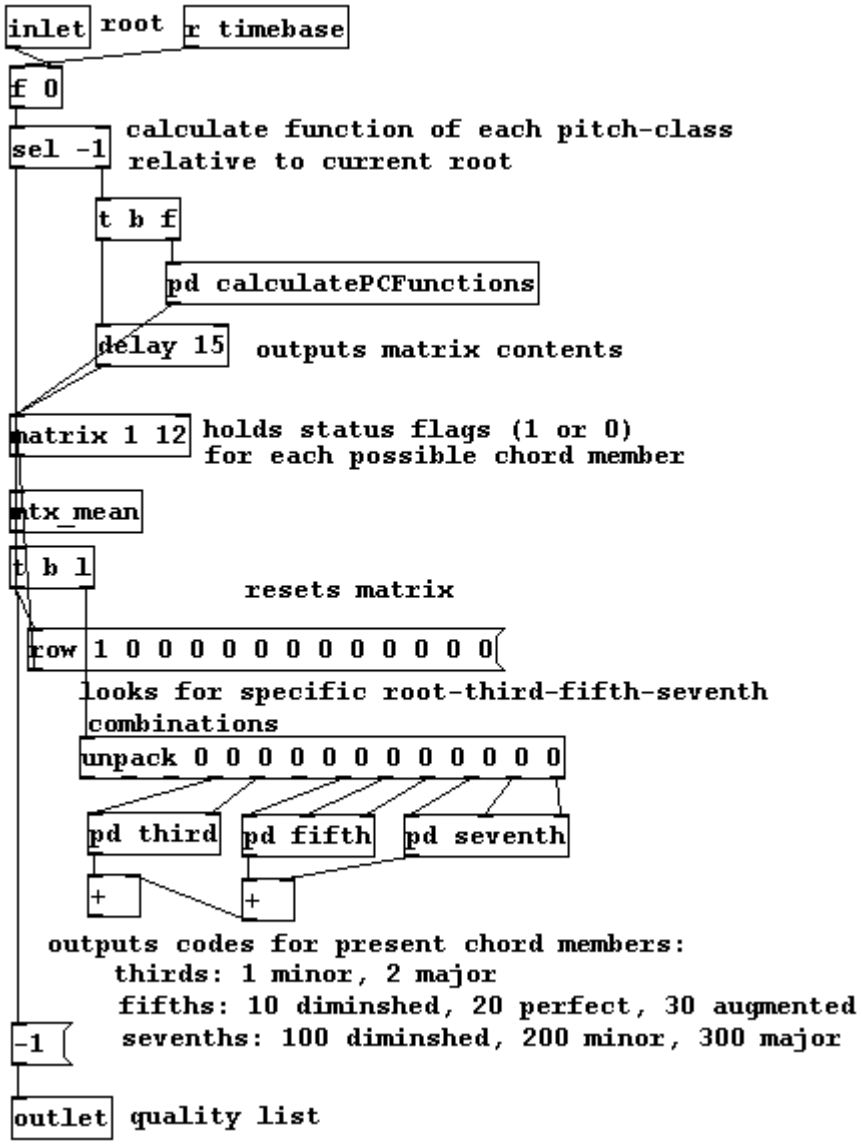


tritone) 7 (perfect fifth) and 8 (augmented fifth). The patch uses an arithmetic trick where each type of 5th is given a "map" value, which is multiplied by the 5th's status flag. The results are then summed together, resulting in the *formula Sum = (d5 status * d5 map) + (P5 status * P5 map) + (A5 status * A5 map)*. The map values are carefully chosen so that each unique sum represents the exclusive presence of a specific fifth or combination of fifths. Diminished fifth is given a map value of 1; perfect fifth is mapped to 2, and augmented fifth is mapped to 4. So, for example, if only a perfect fifth is present, the result will be (0 * 1) + (1 * 2) + (0 * 4) = 2. A sum of 2 means that a perfect fifth, and only a perfect fifth, is present. The possible sums and what they each represent are listed in **Figure 44**. A similar process is carried out when looking for minor and major thirds and diminished, minor, and major sevenths.

Once the types of thirds, fifths, and sevenths present in a chord have been identified, the results are output in the form of a "quality code." The codes are listed on the **quality** parent patch, pictured in **Figure 45**. For example, the code 222 represents a dominant seventh chord, since 200 = minor seventh, 20 = perfect fifth, and 2 = major third. This code system is admittedly ad hoc and arbitrary, but it serves to communicate the basic harmonic quality of a chord in a very compact way.

Our chord identifier is now fully complete. It accepts a stream of MIDI notes as its input, and outputs the root expressed as a pitch-class (0-11) and the quality expressed as a quality code. This information can be distributed to any patch that needs to know the immediate harmonic context. The computer is now able to operate on a higher, more musical level. The human→computer Influence now has the potential to be not only more extensive, but more musically meaningful as well.

Figure 45: quality parent patch



Performance comparison of chord identification systems

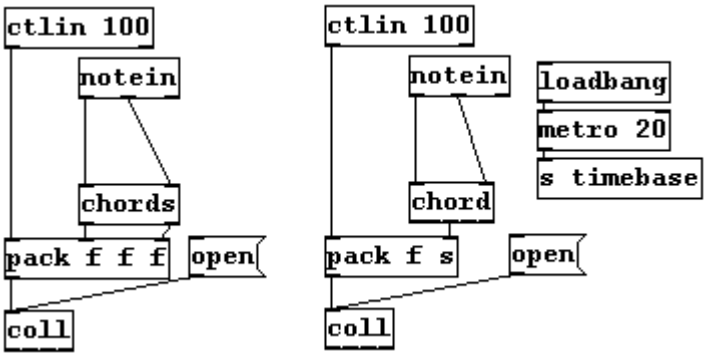
Factors influencing performance

How well do the various approaches to chord identifying work? How does the output of each of them compare to an analysis performed by a musician?

Olaf Matthes has ported Robert Rowe's chord identification code to PD and included it in has *Maxlib* library of externals as the object **chord**. My connectionist chord identifier and Rowe's rule-based system can therefore operate and be tested side-by-side in the same environment. Two testing exercises, one involving block chords and the other arpeggios, will serve to illustrate the strengths and weaknesses of each system. For these tests, the neural net in **chords** was trained using a 312-item training set (included in the appendix) containing triads and various seventh chords.

The patch pictured in **Figure 46** is designed to test and capture the output of **chord** and **chords**. It samples the output of each identifier whenever it receives a *controller 100* message on channel 16; these messages are embedded in the test MIDI files (the reasoning for this is explained shortly). The test MIDI files are recordings of a musician playing along with a metronome. The sampled output of each identifier is sent to **coll** objects, where it can be viewed and saved to a file. Annotated scores of the test files, along with the resulting

Figure 46: chord tester



contents of the .coll files, are in the appendix of this paper. The sample MIDI files are included on the CD-ROM appendix.

The speed in which **chord** and **chords** can provide an assessment of the chord currently being played has a large effect on their efficacy and usefulness. Rowe's derived **chord** extern was written and compiled in C, and is therefore quite efficient. It is designed to complete an analysis within one DSP cycle of the presentation of input, resulting in no measurable time delay between the input and output. My **chords** abstraction was, by comparison, built within PD, and is therefore somewhat slower. The speed of this object is determined largely by the setting on the metronome that supplies BANG messages to **s timebase**, the synchronizing clock for all ongoing analyses. Smaller timebase values result in more rapid analysis with a heavier CPU load; larger values produce sluggish analyses. An optimal setting, arrived at through trial and error, has the metronome sending a BANG to **timebase** every 20 milliseconds. At this setting, most input causes an analyzed root to be output in less than 10 milliseconds, with root analysis never taking longer than 22 milliseconds. The chord type identifier, which must wait for the root to be identified before even starting its analysis, is somewhat slower, most often needing between 10 and 30 milliseconds, and sometimes as much as 70 milliseconds, to produce output. This analysis, while sluggish, is still useable in a live performance situation.

Both **chord** and **chords** output a new assessment of chord root and type each time a new note is played or released. The resulting stream of data is dense and redundant, and must be "thinned" for it to be useable. One way of doing this involves listening to or "sampling" the algorithm's output once every beat or two. Our beat tracker could be

used to ensure that this is done at metrically appropriate times (the test MIDI files contain controller messages to trigger a round of output sampling, so that inaccuracies in the beat tracker do not show up as inaccuracies in the chord identifier output stream). In a live performance, however, chords may not be played "on the beat." Some styles of music in fact dictate that chords and notes should be played slightly "behind the beat." Sampling the output of a chord identifier directly on the beat may therefore miss significant important information, leading to erroneous chord identification.

An obvious but effective solution to this problem involves sampling the output of a chord identifier slightly after the beat, a feat which can be accomplished by simply inserting a delay between the beat output of the beat tracker and the inlet that triggers a new sampling of the chord identifier. Choosing the length of this delay is a small but important matter. Rowe's **chord** object seems to function best when its output is sampled about 25 milliseconds after the beat; longer delays do not improve performance noticeably. Because of the slower processing speed described above, my **chords** object requires a delay of 75 milliseconds to achieve top performance.

Finally, both **chord** and **chords** have mechanisms that allow them to identify arpeggiated chords. In my **chords** object, this is achieved through the use of a decay function, as described above. The length of the decay is variable, and has a great effect on the object's performance. For block chords, a decay of 0 is ideal, since the resulting analysis will be based only on notes that are on at the time when the chord identifier is sampled. For arpeggios, the ideal decay would be long enough so that the first note of each arpeggio will still be somewhat active when the last note is played, allowing the arpeggio as a whole can be identified together as a single chord. A decay of 300

milliseconds seems to be optimal for purely arpeggiated passages. In practice, the length of the decay must be set at loadtime, and cannot change throughout the course of a piece, so a compromise much be reached. While a decay of 150 milliseconds works moderately well for passages of mixed block chords and arpeggios, it is obvious that this parameter should be calibrated to achieve optimal performance on individual pieces. Interestingly, while underestimating the decay length causes each individual note to successively be identified as the chord root, overestimating it causes a rough harmonic analysis of larger metric groups (such as groups of beats or measures) to be output, an analysis which could potentially be useful.

Block chords example: J. S. Bach's *Jesu Meine Freude*

Both **chord** and **chords** provide an adequate analysis of this piece. Rowe's **chord** has two problems that cause several chords to be identified incorrectly. First, In trying to determine the function of each and every note in a chord, **chord** often classifies nonharmonic tones, such as passing tones and upper neighbors, as exotic functioning chord members. In measure 19, for example, the chord in the first beat is identified as F major, flat five, major seventh. This chord is actually an E-minor with a double suspension: factoring the A and the F into the chord, rather than ignoring them as nonharmonic tones, causes the erroneous identification. Secondly, the fact that **chord** is designed to identify arpeggios as well as block chords sometimes causes successive chords to be blurred together. In measure 17, for example, the G-major chord in the fourth beat is misidentified as an E-minor minor-seventh, because it is blended together

with the C-major chord in the first three beats. These two problems account for most or all of the chords misidentified by **chord** in this example.

My **chords** is also affected by this blending of successive chords to a greater or lesser extent, depending on the setting of the decay length. When decay = 0 ms, no blending occurs, but significant blending is apparent if decay is set as high as 300 ms. For example, when decay = 300 ms, the chord in measure 17 misidentified by **chord** above is misidentified here as a C-major major-seventh. This effect can be largely controlled by resetting the decay length to a more appropriate value; **chords** achieved 90+% accuracy on this example when decay was set to 0 ms, but was reduced to below 50% accuracy when decay was changed to 300 ms. However, as was mentioned above, even a blurred chord identification provides some insight into the harmonic structure of a larger structural grouping. The last beat of measure 18 and the whole of measure 19 are collectively identified as having a root of F, which is not an unfeasible analysis, considering that it's actually an F chord followed by several A-minor chords.

Chords does not share **chord**'s tendency to force nonharmonic tones into an exotic functional role within a chord. Since it looks to identify only a root, third, fifth, and seventh, any notes that cannot be considered one of these members is simply ignored. If one wishes to correctly identify the upper members of chords, one would need to add them to the training set and to modify the chord type algorithm.

The few remaining inaccuracies in **chords**' analysis are difficult to explain. The last beat of measure 20, for example, is identified as having a root of C#, despite the fact that there has not been a C# in the entire example to this point. These errors are most likely biproducts of the neural net's functioning, and perhaps suggest that the net needs to

be "overtrained" to eliminate any spurious biases that may be lingering in an otherwise well-functioning neural net.

Arpeggio example: Scriabin's Opus 11 *Prelude* in C

Neither **chord** nor **chords** performs remarkably on this difficult piece, but, given the challenges of arpeggiation, missing chord members, and ambiguous roots, neither algorithm fails miserably.

All of the chord identifications output but Rowe's **chord** are feasible, and most of them are correct. All of the questionable identifications can be attributed to the two problems discussed above, namely chord blurring and the algorithm's already discussed tendency to classify all active notes as functional chord members. In measure 9, for example, the first two beats are blurred together and identified as F major-ninth.

My **chords** supplies roughly the same identification as **chord** about 75% of the time. The erroneous identifications, such as the repeated identification in measures 8-10 of E-minor and G-Major chords as C#-diminished, are once again difficult to explain. These are, as mentioned, most likely byproducts of the net's operation in the form of biases in the connection weights, and could possibly be eliminated by training the net past the point where it achieves 100% accuracy on the training set.

While the performance of my connectionist chord identifier is certainly not perfect, it is, in my experience, accurate and fast enough to be useful in a live performance situation. This tests have demonstrated that **chords**, while not producing a 100% accurate representation of the music it is analyzing, does manage to capture a

reliable harmonic context in which compositional algorithms can operate. A performer can then Influence the computer's output in a harmonically meaningful way.

6. Conclusion

Deriving musical meaning from an ongoing performance is the first step in making computers act and react in musical ways. The tools described in the previous chapters provide a computer with only the most basic musical understanding. Research into the task of analyzing a human's musical performance is ongoing, and will be for some time to come. Advances in this field are continuously being made, and many researchers make their findings available to others almost immediately via the internet in the form of patches or external programs.

Yet the tools outlined above, the tempo tracker and chord identifier, are enough to allow real musical interaction to occur when used in a larger musical environment. A compositional example will serve to illustrate the potential power of these tools.

Compositional Example: *Mass1*

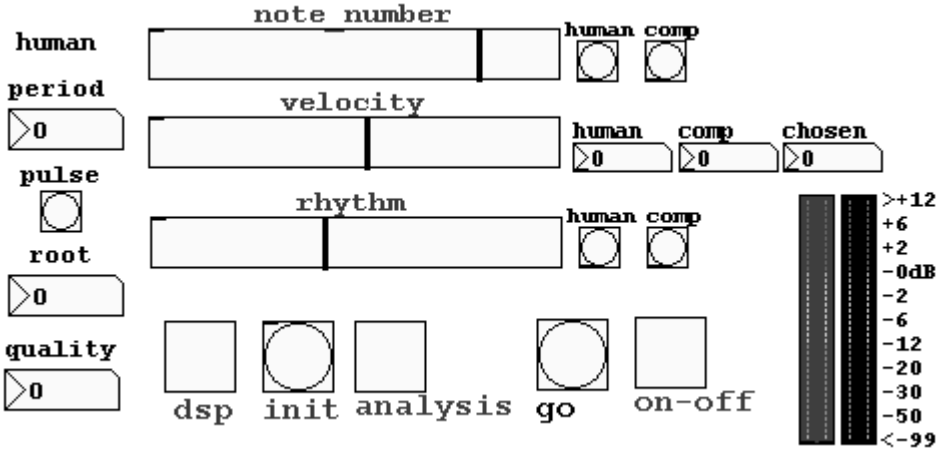
The *Mass* series is a set of templates for interactive composition and performance. They are not pieces of music *per se*, but rather environments for the production of pieces. The two current versions, *Mass1* and *Mass3* are very similar in that they consist of two individuals producing musical material, and a compositional algorithm that generates an audio output stream based on interpolation of various features of the two inputs.

Mass1 involves a human musician whose performance is being analyzed, a "computer composer" suggesting notes for future output, and a "shared space" that ultimately decides what notes should be played. The interface for *Mass1* is shown in **Figure 47**. Most of the objects are informational: *period*, *pulse*, *root*, and *quality* are all

outputs of the objects analyzing the human's input. *Dsp* turns PD's sound engine on and off; *analysis* turns the analysis objects on and off; *init* resets the analyses. *On-off* turns the computer composer on and off, and *go* tells it to start producing output.

The "computer composer" generates suggestions for what the note number,

Figure 47: ***Mass1***



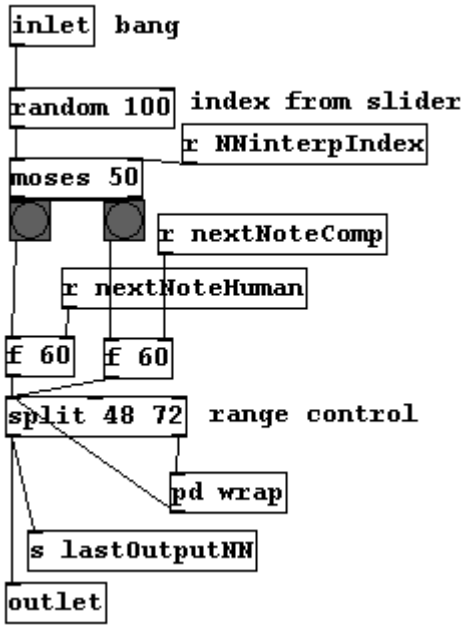
velocity, and rhythmic value of the next output notes should be; the computer composer does not itself make any sound. Note number and rhythm suggestions are chosen using Markov chains created by **blankHands**, as previously discussed, loaded, for example, with the Bach analysis files we created earlier. Velocity is chosen using a set of musical rules, which state, for example, that an ascending melodic line should be accompanied by a crescendo, and that the note following a tritone should be soft. On the human side, note number suggestions are made via Markov chains generated from the realtime analysis of the human's input. Velocity explicitly follows the human's playing: the suggestion for the next velocity is the last velocity played by the human. Similarly, the last metric unit played by the human becomes the suggestion for the next output rhythm.

Any number of rules and analysis-genesis algorithms could be put in place of those described above: the point is that both the human and the computer make

suggestions for each of the parameters of future output notes. These suggestions are sent to a "shared space" that ultimately decides what the parameters of the next output should be.

Returning to the interface, the three faders, *note number*, *velocity*, and *rhythm,* control who has control over each parameter. If, for example, the note number fader is entirely to the left, the human's suggestion for note number will be used every time; if the fader is to the right, the computer's suggestion will be used every time. If the fader is the middle, the human's suggestion will be used about half the time, and the computer's suggestion will be used about half the time. The further to the left the fader gets, the more often the human's suggestion will be used. This behavior is accomplished by the patch in **Figure 48**. Velocity is handled a bit differently: a weighted average is taken between the computer and human's suggestions, so that when the fader is somewhere in the middle the velocity value is somewhere in the middle as well. As the fader approaches hard left, the output velocity approaches the human's suggestion.

Figure 48: parameter interpolation



Mass1 therefore has three parameters, three control surfaces that effect the unfolding composition, and all three of them are, in a very direct way, manifestations of the Influence parameter introduced above. *Mass3*, a similar template for composition, expands this concept to include interaction between two humans: the "computer composer" is replaced

by a second human, and the "shared space" interpolates between the analysis of their input to direct a third, computer-generated musical voice.

Pieces created with these *Mass1* and *Mass3* are demonstrations of the Influence and Participation parameters that are the subject of this paper. Influence and Participation together describe the roles of the players in interactive music systems. The drama of humans and computers interacting in the ongoing creation of music is the method and result of much of today's computer music. While it is obvious that not all interactive composers think in terms of information flow, it is clear that some do. What is apparent is that all interactive music systems can, to a greater or lesser extent, be understood in these terms. As the ability to communicate musical intention to computers grows, they more and more become manifestations of our musical thought, allowing them to be creative partners in improvisation, composition, and performance.

Bibliography

Chadabe, Joel. *Electric Sound : the past and promise of electronic music*. Upper Saddle River, NJ : Prentice Hall, 1997.

Clarke, Eric R.. "Rhythm and Timing in Music." In *The Psychology of Music*, 2d ed., ed. Diana Deutsch, pp. 473-500. San Diego, California: Academic Press, 1999.

Desain, Peter. "A connectionist and a traditional AI quantizer, symbolic versus sub-symbolic models of rhythm perception." *Contemporary Music Review* 9, Parts 1 & 2 (1993): 239-254.

Desain, P., and H. Honing. "The Quantization of Musical Time: A Connectionist Approach." *Computer Music Journal* 13, No. 3 (1989): 56-66.

Desain, P., and H. Honing. "Advanced issues in beat induction modeling: syncopation, tempo, and timing." *Proceedings of the 1994 International Computer Music Conference*, pp.92-94. San Francisco: International Computer Music Association, 1994.

Desain, P., and H. Honing. "Computational models of beat induction: the rule-based approach." *Journal of New Music Research* 28, no. 1 (1999): 29-42.

Di Scipio, Agostino. "Sound is the Interface: Sketches of a Constructivistic Ecosystem View of Interactive Signal Processing." *Proceedings of the Colloquium on Musical Informatics*, Firenze 8-10 May 2003.

Dodge, Charles, and Thomas A. Jerse. *Computer Music: synthesis, composition, and performance*, Second Edition. United States: Wadsworth Publishing Company, 1997.

Garson, James, "Connectionism", *The Stanford Encyclopedia of Philosophy* (Winter 2002 Edition), Edward N. Zalta (ed.), URL = http://plato.stanford.edu/archives/win2002/entries/connectionism/.

Large, Edward W.. "The Resonant Dynamics of Beat Tracking and Meter Perception." *Proceedings of the 1994 International Computer Music Conference*, pp.90-91. San Francisco: International Computer Music Association, 1994.

Lerdahl, F., and R. Jackendoff. *A Generative Theory of Tonal Music*. Cambridge, Mass.: The MIT Press, 1983.

Lewis, George E. "Too Many Notes: Computers, Complexity, and Culture in *Voyager*." *Leonardo Music Journal* 10 (2000): 33-39.

Longuet-Higgins, H. C., and C. S. Lee. "The Perception of Musical Rhythms." *Perception* 11 (1982): 115-128.

Longuet-Higgins, H. C., and C. S. Lee. "The Rhythmic Interpretation of Monophonic Music." *Music Perception* 1, no. 4 (1984): 424-441.

Moore, F. Richard. "The Dysfunctions of MIDI." *Computer Music Journal* 12, No. 1 (Spring 1998), 19-28.

Parncutt, Richard. "A model of beat induction accounting for perceptual ambiguity by continuously variable parameters." *Proceedings of the 1994 International Computer Music Conference*, pp.83-84. San Francisco: International Computer Music Association, 1994.

Roads, Curtis. *Microsound*. Cambridge, Mass.: MIT Press, 2001.

Rosenthal, David. "A Model of the Process of Listening to Simple Rhythms." *Music Perception* 6, no. 3 (1989): 315-328.

Rosenthal, D., M. Goto, and Y. Muraoka. "Rhythm Tracking Using Multiple Hypotheses." *Proceedings of the 1994 International Computer Music Conference*, pp.85-87. San Francisco: International Computer Music Association, 1994.

Rowe, Robert. *Interactive Music Systems: machine listening and composing*. Cambridge, Mass.: MIT Press, 1993.

Rowe, Robert. *Machine Musicianship*. Cambridge, Mass.: MIT Press, 2001.

Tenny, J., and L. Polansky. "Temporal Gestalt Perception in Music." *Journal of Music Theory* 24 (1980): 205-241.

Winkler, Todd. *Composing Interactive Music: Techniques and Ideas Using Max*. Cambridge, Mass.: MIT Press, 2001

Appendix

CHORDS Training Set

Correct chord root followed by active pitch-classes. Pitch-class 0 is C, 1 is Db, and so on, and –1 is "off."

```
0  0  -1  -1  1;
1  1;
2  2;
3  3;
4  4;
5  5;
6  6;
7  7;
8  8;
9  9;
10 10;
11 11;
0  0  7;
1  1  8;
2  2  9;
3  3  10;
4  4  11;
5  5  0;
6  6  1;
7  7  2;
8  8  3;
9  9  4;
10 10 5;
11 11 6;
0  0  4;
1  1  5;
2  2  6;
3  3  7;
4  4  8;
5  5  9;
6  6  10;
7  7  11;
8  8  0;
9  9  1;
10 10 2;
11 11 3;
0  0  3;
1  1  4;
2  2  5;
3  3  6;
4  4  7;
5  5  8;
6  6  9;
7  7  10;
8  8  11;
9  9  0;
10 10 1;
11 11 2;
0  0  4  7 -1;
1  1  5  8;
2  2  6  9;
3  3  7  10;
4  4  8  11;
5  5  9  0;
```

```
6  6  10 1;
7  7  11 2;
8  8  0  3;
9  9  1  4;
10 10 2  5;
11 11 3  6;
0  4  7  0;
1  5  8  1;
2  6  9  2;
3  7  10 3;
4  8  11 4;
5  9  0  5;
6  10 1  6;
7  11 2  7;
8  0  3  8;
9  1  4  9;
10 2  5  10;
11 3  6  11;
0  7  0  4;
1  8  1  5;
2  9  2  6;
3  10 3  7;
4  11 4  8;
5  0  5  9;
6  1  6  10;
7  2  7  11;
8  3  8  0;
9  4  9  1;
10 5  10 2;
11 6  11 3;
0  0  3  7;
1  1  4  8;
2  2  5  9;
3  3  6  10;
4  4  7  11;
5  5  8  0;
6  6  9  1;
7  7  10 2;
8  8  11 3;
9  9  0  4;
10 10 1  5;
11 11 2  6;
0  3  7  0;
1  4  8  1;
2  5  9  2;
3  6  10 3;
4  7  11 4;
5  8  0  5;
6  9  1  6;
7  10 2  7;
8  11 3  8;
9  0  4  9;
10 1  5  10;
11 2  6  11;
```

```
0  7  0  3;
1  8  1  4;
2  9  2  5;
3  10 3  6;
4  11 4  7;
5  0  5  8;
6  1  6  9;
7  2  7  10;
8  3  8  11;
9  4  9  0;
10 5  10 1;
11 6  11 2;
0  0  3  6;
1  1  4  7;
2  2  5  8;
3  3  6  9;
4  4  7  10;
5  5  8  11;
6  6  9  0;
7  7  10 1;
8  8  11 2;
9  9  0  3;
10 10 1  4;
11 11 2  5;
0  3  6  0;
1  4  7  1;
2  5  8  2;
3  6  9  3;
4  7  10 4;
5  8  11 5;
6  9  0  6;
7  10 1  7;
8  11 2  8;
9  0  3  9;
10 1  4  10;
11 2  5  11;
0  6  0  3;
1  7  1  4;
2  8  2  5;
3  9  3  6;
4  10 4  7;
5  11 5  8;
6  0  6  9;
7  1  7  10;
8  2  8  11;
9  3  9  0;
10 4  10 1;
11 5  11 2;
0  0  4  8;
1  1  5  9;
2  2  6  10;
3  3  7  11;
0  4  8  0;
1  5  9  1;
```

```
2  6  10 2;
3  7  11 3;
0  8  0  4;
1  9  1  5;
2  10 2  6;
3  11 3  7;
0  4  7  10 0;
1  5  8  11 1;
2  6  9  0  2;
3  7  10 1  3;
4  8  11 2  4;
5  9  0  3  5;
6  10 1  4  6;
7  11 2  5  7;
8  0  3  6  8;
9  1  4  7  9;
10 2  5  8  10;
11 3  6  9  11;
0  7  10 0  4;
1  8  11 1  5;
2  9  0  2  6;
3  10 1  3  7;
4  11 2  4  8;
5  0  3  5  9;
6  1  4  6  10;
7  2  5  7  11;
8  3  6  8  0;
9  4  7  9  1;
10 5  8  10 2;
11 6  9  11 3;
0  10 0  4  7;
1  11 1  5  8;
2  0  2  6  9;
3  1  3  7  10;
4  2  4  8  11;
5  3  5  9  0;
6  4  6  10 1;
7  5  7  11 2;
8  6  8  0  3;
9  7  9  1  4;
10 8  10 2  5;
11 9  11 3  6;
0  0  3  7  10;
1  1  4  8  11;
2  2  5  9  0;
3  3  6  10 1;
4  4  7  11 2;
5  5  8  0  3;
6  6  9  1  4;
7  7  10 2  5;
8  8  11 3  6;
9  9  0  4  7;
10 10 1  5  8;
11 11 2  6  9;
```

```
0  3  7  10 0;     4  10 2  4  7;
1  4  8  11 1;     5  11 3  5  8;
2  5  9  0  2;     6  0  4  6  9;
3  6  10 1  3;     7  1  5  7  10;
4  7  11 2  4;     8  2  6  8  11;
5  8  0  3  5;     9  3  7  9  0;
6  9  1  4  6;     10 4  8  10 1;
7  10 2  5  7;     11 5  9  11 2;
8  11 3  6  8;     0  10 0  3  6;
9  0  4  7  9;     1  11 1  4  7;
10 1  5  8  10;    2  0  2  5  8;
11 2  6  9  11;    3  1  3  6  9;
0  7  10 0  3;     4  2  4  7  10;
1  8  11 1  4;     5  3  5  8  11;
2  9  0  2  5;     6  4  6  9  0;
3  10 1  3  6;     7  5  7  10 1;
4  11 2  4  7;     8  6  8  11 2;
5  0  3  5  8;     9  7  9  0  3;
6  1  4  6  9;     10 8  10 1  4;
7  2  5  7  10;    11 9  11 2  5;
8  3  6  8  11;    0  0  3  6  9;
9  4  7  9  0;     1  1  4  7  10;
10 5  8  10 1;     2  2  5  8  11;
11 6  9  11 2;     0  3  6  9  0;
0  10 0  3  7;     1  4  7  10  1;
1  11 1  4  8;     2  5  8  11  2;
2  0  2  5  9;     0  6  9  0  3;
3  1  3  6  10;    1  7  10 1  4;
4  2  4  7  11;    2  8  11 2  5;
5  3  5  8  0;     0  9  0  3  6;
6  4  6  9  1;     1  10 1  4  7;
7  5  7  10 2;     2  11 2  5  8;
8  6  8  11 3;
9  7  9  0  4;
10 8  10 1  5;
11 9  11 2  6;
0  0  3  6  10;
1  1  4  7  11;
2  2  5  8  0;
3  3  6  9  1;
4  4  7  10 2;
5  5  8  11 3;
6  6  9  0  4;
7  7  10 1  5;
8  8  11 2  6;
9  9  0  3  7;
10 10 1  4  8;
11 11 2  5  9;
0  3  6  10 0;
1  4  7  11 1;
2  5  8  0  2;
3  6  9  1  3;
4  7  10 2  4;
5  8  11 3  5;
6  9  0  4  6;
7  10 1  5  7;
8  11 2  6  8;
9  0  3  7  9;
10 1  4  8  10;
11 2  5  9  11;
0  6  10 0  3;
1  7  11 1  4;
2  8  0  2  5;
3  9  1  3  6;
```

Tempo Tracker Test Data

Bach Test

Data is beat number (embedded in test MIDI file) followed by beat period theory in milliseconds. MIDI file generated by a pianist playing along with a metronome set to 120 BPM so the basic quarter-note pulse is around 500 milliseconds. ** indicates that the tracker has switched from tracking eighth-notes to tracking quarter-notes. Musical score follows raw test data.

```
1  0;          20 503.293;    41 501.261;    65 499.229;
1  229.297;    20 509.968;    41 499.664;    66 499.229;
1  232.925;    20 507.066;    42 499.664;    67 499.229;
2  232.925;    21 507.066;    42 500.1;      67 501.116;
2  236.408;    21 500.825;    43 500.1;      68 501.116;
3  236.408;    21 503.728;    44 500.1;      68 499.229;
3  238.295;    22 503.728;    44 498.721;    69 499.229;
3  476.88;**   23 503.728;    45 498.721;    69 498.503;
3  480.943;    23 501.624;    45 503.22;     70 498.503;
4  480.943;    24 501.624;    46 503.22;     70 503.147;
4  482.685;    24 502.204;    46 503.22;     71 503.147;
4  489.941;    25 502.204;    47 503.22;     71 499.229;
5  489.941;    25 502.639;    47 504.381;    72 499.229;
5  494.295;    26 502.639;    48 504.381;    72 504.308;
5  498.068;    26 501.188;    48 504.961;    73 504.308;
6  498.068;    26 500.898;    49 504.961;    73 498.794;
6  500.68;     27 500.898;    49 503.8;      74 498.794;
7  500.68;     27 494.803;    50 503.8;      74 497.923;
7  497.197;    27 485.515;    50 501.478;    75 497.923;
7  487.909;    28 485.515;    51 501.478;    75 503.438;
8  487.909;    28 488.127;    51 491.465;    76 503.438;
8  490.086;    28 493.932;    52 491.465;    76 502.277;
9  490.086;    29 493.932;    53 491.465;    77 502.277;
9  499.664;    29 493.642;    53 484.741;
9  503.147;    29 499.519;    53 480.822;
10 503.147;    30 499.519;    54 480.822;
10 500.825;    30 496.036;    54 484.668;
10 497.633;    31 496.036;    55 484.668;
11 497.633;    31 500.535;    56 484.668;
11 493.569;    31 501.406;    56 485.321;
11 494.15;     32 501.406;    57 485.321;
12 494.15;     32 501.841;    57 483.435;
12 499.229;    33 501.841;    58 483.435;
13 499.229;    33 505.76;     58 483.289;
13 499.519;    34 505.76;     59 483.289;
13 507.356;    34 506.776;    59 483.725;
14 507.356;    35 506.776;    60 483.725;
15 507.356;    35 507.937;    60 480.097;
16 507.356;    36 507.937;    61 480.097;
17 507.356;    36 503.728;    61 481.113;
17 507.066;    37 503.728;    62 481.113;
18 507.066;    37 499.084;    62 482.999;
18 499.229;    38 499.084;    63 482.999;
18 496.036;    38 494.005;    63 498.286;
19 496.036;    39 494.005;    64 498.286;
19 500.68;     39 497.488;    65 498.286;
19 503.293;    40 497.488;    65 503.365;
               40 501.261;    65 496.98;
```

Scriabin Test

Data is beat number (embedded in test MIDI file) followed by beat period theory in milliseconds. MIDI file generated by a pianist playing along with a metronome set to 120 BPM so the basic quarter-note pulse is around 500 milliseconds, although musicians generally perceive this in groups of five eighth-notes. * indicates that the tracker has switched from tracking eighth-notes to tracking quarter-notes. Note the larger fluctuations in beat period due to *rubato*, and the increasing beat periods due to the *ritardando* at the end. Musical score follows raw test data.

```
1  0;          12 520.417;  23 500.1;     34 602.17;    45 681.359;
1  245.261;    12 510.839;  23 493.134;   34 596.752;   45 689.293;
2  245.261;    12 508.807;  23 500.1;     34 607.104;   45 690.164;
2  245.551;    13 508.807;  24 500.1;     35 607.104;   46 690.164;
2  245.841;    13 510.839;  24 490.231;   35 609.426;   46 693.26;
3  245.841;    13 513.451;  24 493.714;   35 616.682;   46 691.083;
3  247.293;    14 513.451;  25 493.714;   36 616.682;   46 688.084;
3  499.713;*   14 503.873;  25 489.07;    36 620.939;   47 688.084;
3  506.679;    14 511.42;   25 495.456;   36 614.554;   47 686.149;
4  506.679;    15 511.42;   26 495.456;   37 614.554;   47 676.909;
4  502.035;    15 499.519;  26 500.39;    37 612.619;   48 676.909;
4  501.164;    15 490.522;  26 500.971;   37 621.81;    48 687.552;
5  501.164;    16 490.522;  27 500.971;   38 621.81;    48 683.198;
5  504.066;    16 495.166;  27 508.807;   38 627.615;   49 683.198;
5  502.035;    16 495.456;  27 505.905;   38 624.519;   49 675.845;
6  502.035;    17 495.456;  28 505.905;   39 624.519;   49 673.281;
6  501.164;    17 493.424;  28 511.42;    39 635.742;   50 673.281;
6  499.422;    17 493.424;  28 514.322;   39 632.259;   50 681.021;
7  499.422;    18 493.424;  29 514.322;   40 632.259;   51 681.021;
7  507.839;    18 492.553;  29 528.447;   40 627.421;   52 681.021;
7  508.71;     18 489.361;  29 531.64;    40 629.163;   53 681.021;
8  508.71;     19 489.361;  30 531.64;    40 633.42;    54 681.021;
8  507.936;    19 497.488;  30 545.475;   41 633.42;    55 681.021;
8  497.778;    19 493.714;  31 545.475;   41 644.014;   56 681.021;
9  497.778;    20 493.714;  31 545.185;   41 649.819;   57 681.021;
9  509.388;    20 498.649;  31 553.118;   42 649.819;   58 681.021;
9  506.195;    20 501.551;  31 563.567;   42 659.881;   59 681.021;
10 506.195;    21 501.551;  32 563.567;   42 665.686;   60 681.021;
10 513.161;    21 499.519;  32 574.306;   43 665.686;   61 681.021;
10 522.449;    21 502.132;  32 578.079;   43 660.462;
11 522.449;    22 502.132;  33 578.079;   44 660.462;
11 520.707;    22 499.81;   33 589.206;   44 668.976;
11 520.417;    22 500.1;    33 602.17;    44 681.359;
```

Chords Test Data

Bach Test: Rippin's **Chords**

Data is beat number (embedded in test MIDI file) followed by root and quality code. Root is in pitch-class, where 0 is C, 1 is Db, and so on, and –1 is "off." Quality code is deciphered as follows: M7=300, m7=200, d7=100; A5=30, P5=20, d5=10; M3=20, m3=10. Data from multiple tests with varying decay and delay settings. Musical score follows raw test data.

```
decay 0   0       300     0
delay 50  75      75      100

4  7 22;  7  22;  7  22;   7  22;
5  7 22;  7  22;  3  332;  7  22;
6  7 22;  7  22;  3  332;  7  22;
7  7 22;  7  22;  3  332;  7  22;
8  0 22;  0  22;  0  22;   0  22;
9  5 20;  5  20;  0  220;  5  20;
10 5 22;  5  22;  5  22    5  22;
11 5 22;  5  22;  5  2     5  22;
12 0 22;  0  22;  5  320;  0  22;
13 7 22;  7  22;  5  110;  7  22;
14 9 21;  9  21;  5  322   9  21;
15 2 221; 2  221; 2  221;  2  221;
16 3 332; 7  22;  5  110;  7  22;
17 0 22;  0  22   5  320;  0  22;
18 0 22;  0  22;  0  22;   0  22;
19 0 22;  0  22;  0  22;   0  22;
20 7 22;  7  22;  0  320;  7  22;
21 7 22;  7  22;  0  320;  7  22;
22 7 22;  11 220; 3  1;    11 220;
23 7 22;  4  21;  3  332;  4  21;
24 7 22;  5  301; 5  112;  5  301;
25 7 22;  9  21;  5  112;  9  21;
26 7 22   9  21;  5  11    9  21;
27 7 22;  9  21;  5  112   9  21;
28 7 22;  11 211; 5  112;  11 211;
29 7 22;  5  312; 11 210;  5  312;
30 7 22;  4  21;  5  320;  4  21;
31 7 22;  7  22;  5  320;  7  22;
32 7 22;  1  210; 5  320;  1  210;
33 7 22;  5  20;  5  320;  5  20;
34 7 22;  0  22;  5  320;  0  22;
35 7 22;  0  22;  0  22;   0  22;
36 7 22;  0  22;  0  22;   0  22;
37 7 22;  0  22;  0  320;  0  22;
38 7 22;  7  22;  0  320;  7  22;
39 9 221; 9  221; 0  122;  9  221;
40 7 22;  7  22;  0  320;  7  22;
41 9 1;   2  20;  3  310;  2  20;
42 9 1;   4  21;  4  21;   4  21;
43 9  1;   5  22;  5  22;  5  22;
44 9  1;   5  2;   5  2;   5  2;
45 9  1;   11 211; 5  112; 11 211;
46 9  1;   0  22;  5  320; 0  22;
47 9  1;   0  22;  0  22;  0  22;
48 9  1;   0  22;  0  22;  0  22;
49 9  1;   7  22   0  320; 7  22;
50 9  1;   2  21;  11 211; 2  21;
51 9  1;   4  211; 10 112; 4  211;
52 9  1;   1  32;  10 321; 1  32;
53 9  1;   1  111; 1  111; 1  111;
54 9  1;   9  222; 1  1;   9  222;
55 9  1;   10 22;  10 22;  10 22;
56 9  1;   2  1;   2  21;  2  21;
57 9  1;   4  211; 2  30;  4  211;
58 9  1;   2  1;   2  21;  2  21;
59 9  1;   2  21   2  21;  2  21;
60 9  1;   9  21;  2  220; 9  21;
61 4  22;  4  22;  5  311; 4  22;
62 4  222; 4  222; 4  222; 4  222;
63 9  21;  9  21;  5  322; 9  21;
64 2  21;  2  21;  2  21;  2  21;
65 11 211; 11 211; 11 211; 11 211;
66 9  221; 0  221; 5  320; 0  22;
67 2  221; 2  221; 2  221; 2  221;
68 11 211; 11 211; 5  112; 11 211;
69 5  10;  5  10;  11 0;   5  10;
70 5  0;   5  320; 5  320; 0  22;
71 0  22;  0  22;  0  22;  0  22;
72 9  222; 9  222; 9  222; 9  222;
73 2  21;  2  21;  1  32;  2  21;
74 3  312; 3  312; 1  1;   3  312;
75 2  21;  2  21;  2  21;  2  21;
76 1  11;  1  11;  1  11;  1  11;
77 1  30;  1  30   1  30;  1  30;
78 1  111; 1  111; 1  111; 1  111;
79 2  2;   -1 -1;  3  301; -1 -1;
80 3  302; 7  21;  3  322; 7  21;
81 7  21;  7  21;  7  21;  7  21;
82 2  22;  2  22;  3  311; 2  22;
83 2  22;  2  22;  2  22;  2  22;
84 -1 -1;  -1 -1;  2  2;   -1 -1;
```

Bach Test: Rowe's **Chord**

Data is beat number (embedded in test MIDI file) followed by active pitch-classes and chord output. Data from multiple tests with varying delay settings. Musical score follows raw test data.

delay 25 75

```
4  G B D : G major;                    G B D : G major;
5  G B D : G major;                    G B D : G major;
6  G B D : G major;                    G B D : G major;
7  G B D A : G dominant 9th;           G B D A : G dominant 9th;
8  C E G : C major;                    C E G : C major;
9  C G Bb F : C dominant 11th;         C G Bb F : C dominant 11th;
10 F A C : F major;                    F A C : F major;
11 F A C : F major;                    F A C : F major;
12 F C E G : F major 9th;              F C E G : F major 9th;
13 G B D : G major;                    G B D : G major;
14 D A C E : D dominant 9th;           D A C E : D dominant 9th;
15 D F A C : D minor 7th;              D F A C : D minor 7th;
16 G B D A C : G dominant 11th;        G B D A C : G dominant 11th;
17 C E G D : C dominant 9th;           C E G : C major;
18 C E G : C major;                    C E G : C major;
19 C E G : C major;                    C E G : C major;
20 E G B D : E minor 7th;              E G B D : E minor 7th;
21 G B D : G major;                    G B D : G major;
22 B F# A E : B dominant 11th;         B F# A E : B dominant 11th;
23 A E G B : A dominant 9th;           A E G B : A dominant 9th;
24 C E G# B : C major 7th #5;          C E G# B : C major 7th #5;
25 A C E B : A minor 9th;              A C E B : A minor 9th;
26 A C E B : A minor 9th;              A C E B : A minor 9th;
27 A C E : A minor;                    A C E : A minor;
28 B D F A C E : B half dim 11th b9;   B D F A C E : B half dim 11th b9;
29 F A Cb E : F major 7th b5;          F A Cb E : F major 7th b5;
30 E G B : E minor;                    E G B : E minor;
31 G B D : G major;                    G B D : G major;
32 E G B D F : E minor 7th b9;         E G B D F : E minor 7th b9;
33 G B D F C : G dominant 11th;        G B D F C : G dominant 11th;
34 F C E G : F major 9th;              F C E G : F major 9th;
35 C E : C major;                      C E G : C major;
36 C E G : C major;                    C E G : C major;
37 C E G : C major;                    C E G : C major;
38 E G B D : E minor 7th;              E G B D : E minor 7th;
39 A C E G D : A minor 11th;           A C E G D : A minor 11th;
40 E G B D A : E minor 11th;           E G B D A : E minor 11th;
41 B D A : B minor 7th;                B D A : B minor 7th;
42 E G B D : E minor 7th;              E G B D : E minor 7th;
43 B F A C : B half diminished b9;     B F A C : B half diminished b9;
44 B F A C : B half diminished b9;     B F A C : B half diminished b9;
45 B D F A : B half diminished 7th;    B D F A : B half diminished 7th;
46 E G B D A : E minor 11th;           C E G D : C dominant 9th;
47 C E G D : C dominant 9th;           C E G D : C dominant 9th;
48 C E G : C major;                    C E G : C major;
49 C G B D : C major 9th;              C G B D : C major 9th;
50 G B D F A : G dominant 9th;         G B D F A : G dominant 9th;
51 Bb D Fb A : Bb major 7th b5;        Bb D Fb A : Bb major 7th b5;
52 G Bb Db F A : G half diminished 9th;   G Bb Db F A : G half
                                                    diminished 9th;
53 A C# E G Bb : A dominant 7th b9;    A C# E G Bb : A dominant 7th
                                                    b9;
```

54 A C# E G : A dominant 7th;	A C# E G : A dominant 7th;
55 G Bb D F C# : G minor 7th #11;	G Bb D F C# : G minor 7th #11;
56 Bb D F A : Bb major 7th;	Bb D F A : Bb major 7th;
57 E G Bb D F A : E half dim 11th b9;	E G Bb D F A : E half dim 11th b9;
58 Bb D F A : Bb major 7th;	Bb D F A : Bb major 7th;
59 Bb D F A : Bb major 7th;	Bb D F A : Bb major 7th;
60 D A C E : D dominant 9th;	D A C E : D dominant 9th;
61 E G# B : E major;	E G# B : E major;
62 E G# B D : E dominant 7th;	E G# B D : E dominant 7th;
63 A C E B : A minor 9th;	A C E B : A minor 9th;
64 D F A C : D minor 7th;	D F A C : D minor 7th;
65 B D F A : B half diminished 7th;	B D F A : B half diminished 7th;
66 A C E G D : A minor 11th;	A C E G D : A minor 11th;
67 D F A C G : D minor 11th;	D F A C G : D minor 11th;
68 B D F A C : B half diminished b9;	B D F A C : B half diminished b9;
69 G B D F A : G dominant 9th;	G B D F A : G dominant 9th;
70 G B F : G dominant 7th;	F C E G : F major 9th;
71 F C E G : F major 9th;	F C E G : F major 9th;
72 A C# E G B# : A dominant #9;	A C# E G B# : A dominant #9;
73 D F A C# : D minor/major 7th;	D F A C# : D minor/major 7th;
74 A E G D : A dominant 11th;	A E G D : A dominant 11th;
75 G D F A : G dominant 9th;	G D F A : G dominant 9th;
76 Db Fb Abb Ebb : Db half diminished b9;	Db Fb Abb Ebb : Db half diminished b9;
77 A C# E G F : A dominant 7th b13;	A C# E G F : A dominant 7th b13;
78 Db Fb Abb Cbb : Db diminished 7th ;	Db Fb Abb Cbb : Db diminished 7th;
79 D F# A C# G Bb : D major 11th b13;	Bb D F# A : Bb major 7th #5;
80 G Bb D A : G minor 9th;	G Bb D A : G minor 9th;
81 G Bb D : G minor;	G Bb D : G minor;
82 G Bb D F# A : G minor major 9th;	G Bb D F# A : G minor major 9th;
83 G Bb D F# A : G minor major 9th;	G Bb D F# A : G minor major 9th;
84 G Bb D F# A : G minor major 9th;	G Bb D F# A : G minor major 9th;

Scriabin Test: Rippin's **Chords**

Data is beat number (embedded in test MIDI file) followed by root and quality code. Root is in pitch-class, where 0 is C, 1 is Db, and so on, and –1 is "off." Quality code is deciphered as follows: M7=300, m7=200, d7=100; A5=30, P5=20, d5=10; M3=20, m3=10. Data from multiple tests with varying decay and delay settings. Musical score follows raw test data.

decay	150	300	300	500	800
delay	75	75	125	75	75
1	8 12;	8 12;	8 12;	8 12;	8 12;
2	5 20;	5 20;	5 20;	5 201;	2 201;
3	7 200;	7 200;	7 200;	7 200;	2 1;
4	10 102;	10 102;	10 10;	10 102;	10 102;
5	3 2;	3 2;	3 2;	1 10;	1 10;
6	8 12;	8 12;	2 20;	0 0;	0 0;
7	5 0;	5 0;	2 20;	5 0;	2 1;
8	2 0;	7 20;	2 20;	7 20;	2 0;
9	9 0;	2 20;	2 20;	2 20;	2 20;
10	2 31;	0 22;	4 1;	0 22;	9 221;
11	2 31;	0 20;	10 0;	0 20;	0 20;
12	2 31;	10 20;	10 0;	2 31;	2 31;
13	0 200;	0 200;	10 0;	0 200;	2 230;
14	3 300;	3 10;	3 10;	3 10;	3 10;
15	3 300;	8 2;	8 2;	2 200;	2 200;
16	3 300;	0 20;	0 20;	5 20;	5 20;
17	10 20;	2 200;	10 20;	2 31;	2 31;
18	0 20;	0 20;	0 20;	0 20;	7 0;
19	2 0;	2 0;	2 0;	2 0;	2 0;
20	2 1;	2 221;	5 22;	2 220;	2 220;
21	2 1;	2 1;	2 1;	2 1;	2 1;
22	2 1;	9 20;	9 20;	9 20;	2 20;
23	2 1;	2 20;	2 20;	2 20;	2 20;
24	2 1;	2 0;	9 20;	2 0;	2 0;
25	2 1;	3 2;	3 2;	9 200;	9 200;
26	2 1;	1 31;	1 31;	1 31;	1 31;
27	2 1;	2 1;	2 1;	2 1;	2 1;
28	2 1;	2 20;	2 20;	2 20;	2 20;
29	2 1;	2 0;	2 0;	2 0;	2 0;
30	2 1;	1 2;	1 2;	7 200;	2 1;
31	10 2;	10 2;	10 2;	10 2;	10 2;
32	9 20;	9 20;	9 20;	9 20;	9 20;
33	5 100;	5 20;	5 20;	5 20;	5 20;
34	5 100;	5 20;	5 20;	-1 -1;	-1 -1;
35	5 100;	10 2;	10 2;	2 0;	2 0;
36	5 100;	9 200;	9 200;	5 2;	2 20;
37	5 100;	1 0;	1 0;	1 0;	1 0;
38	5 100;	7 0;	7 0;	4 1;	1 10;
39	5 100;	11 0;	11 0;	7 2;	7 2;
40	5 100;	7 20;	7 20;	7 20;	7 20;

```
41 5  100;     9  1;     5  1;     5 22;     2 220;
42 5  100;     1  12;    1  12;    1 12;     1 12;
43 7  200;     7  200;   7  200;   7 200;    7 200;
44 5  12;      5  12;    5  10;    5 12;     5 12;
45 10 102;     10 102;   10 102;   4 201;    1 1;
46 9  1;       5  0;     9  1;     5 22;     5 22;
47 1  12;      5  0;     1  12;    1 12;     5 0;
48 7  0;       5  0;     7  0;     7 0;      7 0;
49 7  0;       5  0;     0  0;     0 0;      0 0;
50 0  0;       0  22;    0  22;    0 22;     0 22;
51 0  0;       0  0;     0  22;    0 22;     0 22;
52 0  0;       0  0;     -1 -1;    0 0;      0 0;
```

Scriabin Test: Rowe's **Chord**

Data is beat number (embedded in test MIDI file) followed by active pitch-classes and chord output. Data from multiple tests with varying delay settings. Musical score follows raw test data.

delay 25	75	125
1 C : C unison;	D C : D dominant 7th;	D C : D dominant 7th;
2 F C : F major;	F C : F major;	F C : F major;
3 G D F : G dominant 7th;	G D F : G dominant 7th;	G D F : G dominant 7th;
4 E G D F : E minor 7th b9;	E G D F : E minor 7th b9;	E G D F : E minor 7th b9;
5 A G : A dominant 7th;	A G : A dominant 7th;	A G : A dominant 7th;
6 A C G : A minor 7th;	D C : D dominant 7th;	D C : D dominant 7th;
7 F C : F major;	F C : F major;	F C : F major;
8 G D F : G dominant 7th;	G D F : G dominant 7th;	G D F : G dominant 7th;
9 D A C G : D dominant 11th;	D A C G : D dominant 11th;	D A C G : D dominant 11th;
10 E G : E minor;	E G : E minor;	E G : E minor;
11 C G : C major;	C G : C major;	C G : C major;
12 Bb F : Bb major;	Bb F : Bb major;	Bb F : Bb major;
13 C G Bb : C dominant 7th;	C G Bb : C dominant 7th;	C G Bb : C dominant 7th;
14 A G : A dominant 7th;	A G : A dominant 7th;	A G : A dominant 7th;
15 D C : D dominant 7th;	D C : D dominant 7th;	D C : D dominant 7th;
16 C G : C major;	C G : C major;	C G : C major;
17 Bb F : Bb major;	Bb F : Bb major;	Bb F : Bb major;
18 C G Bb : C dominant 7th;	C G Bb : C dominant 7th;	C G Bb : C dominant 7th;
19 D F C : D minor 7th;	D F C : D minor 7th;	D F C : D minor 7th;
20 A C : A minor;	A C : A minor;	A C : A minor;
21 D F : D minor;	D F : D minor;	D F : D minor;
22 A E : A major;	A E : A major;	A E : A major;
23 D F A C : D minor 7th;	D F A C : D minor 7th;	D F A C : D minor 7th;
24 D A E : D dominant 9th;	D A E : D dominant 9th;	D A E : D dominant 9th;
25 A G : A dominant 7th;	A G : A dominant 7th;	A G : A dominant 7th;
26 F A E : F major 7th;	F A E : F major 7th;	F A E : F major 7th;
27 D F : D minor;	D F : D minor;	D F : D minor;
28 D F A C : D minor 7th;	D F A C : D minor 7th;	D F A C : D minor 7th;
29 D C : D dominant 7th;	D C : D dominant 7th;	D C : D dominant 7th;
30 F : F unison;	F : F unison;	F : F unison;
31 E D : E dominant 7th;	E D : E dominant 7th;	E D : E dominant 7th;
32 A E : A major;	A E : A major;	A E : A major;
33 F C : F major;	F C : F major;	F C : F major;
34 F C : F major;	F C : F major;	F C : F major;
35 D : D unison;	D : D unison;	D : D unison;
36 A C G : A minor 7th;	A C G : A minor 7th;	A C G : A minor 7th;
37 F E : F major 7th;	F E : F major 7th;	F E : F major 7th;
38 G B : G major;	G B : G major;	G B : G major;
39 B D : B minor;	B D : B minor;	B D : B minor;
40 G D : G major;	G D : G major;	G D : G major;
41 A C : A minor;	A C : A minor;	A C : A minor;
42 F E G : F major 9th;	F E G : F major 9th;	G B F A : G dominant 9th;
43 G D F : G dominant 7th;	G D F : G dominant 7th;	G B F A : G dominant 9th;
44 G B F A : G dominant 9th;	G B F A : G dominant 9th;	G B F A : G dominant 9th;
45 E G D F : E minor 7th b9;	E G D F : E minor 7th b9;	E G D F : E minor 7th b9;
46 B D : B minor;	A C : A minor;	G F : G dominant 7th;
47 G F : G dominant 7th;	G F : G dominant 7th;	G F : G dominant 7th;
48 G F : G dominant 7th;	G F : G dominant 7th;	G F : G dominant 7th;
49 C G : C major;	E G : E minor;	C G : C major;
50 E G : E minor;	E G : E minor;	E G : E minor;
51 C E G : C major;	C E G : C major;	C E G : C major;
52 C E G : C major;	C E G : C major;	C E G : C major;