

UC Santa Cruz

UC Santa Cruz Electronic Theses and Dissertations

Title

Embedded Soundscaping: Audio-reactive Instrument Design for Generative Sound and Music, Interactive Sound Installations, and Acoustic Experimentation

Permalink

<https://escholarship.org/uc/item/2d33344j>

Author

Tooby, Scott M.

Publication Date

2017

Supplemental Material

<https://escholarship.org/uc/item/2d33344j#supplemental>

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

**Embedded Soundscaping: Audio-reactive Instrument Design for
Generative Sound and Music, Interactive Sound Installations, and
Acoustic Experimentation**

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF FINE ARTS

in

Digital Arts and New Media

by

Scott Tooby

June 2017

The Thesis of Scott Tooby
is approved:

Professor Larry Polansky, Chair

Yolande Harris, Ph.D.

Professor Mircea Teodorescu

Tyrus Miller

Vice Provost and Dean of Graduate Studies

Copyright © by
Scott Tooby
2017

TABLE OF CONTENTS

Table of Contents.....	iii
List of Appendices.....	vi
Abstract.....	vii
Acknowledgements.....	viii
 1. Introduction.....	 1
 2. The Sonic Mirror Instrument System.....	 10
2.1 Introduction.....	10
2.2 Software Instrument Design.....	12
2.3 Operating Modes.....	16
2.3.1 Interactive Call and Response Mode.....	16
2.3.2 Recursive Acoustic Convolution (RAC) Mode.....	19
2.4 On-going and Future Development.....	24
 3. Influences and Context.....	 26
3.1 Introduction.....	26
3.2 David Dunn's Sonic Mirror and bark beetle research.....	27
3.3 Alvin Lucier, I Am Sitting in a Room (1969).....	34
3.4 Gordon Mumma's Sound Modifier Console.....	39
3.5 Adam Basanta, A Room Listening to Itself (2015).....	43
3.6 Krzysztof Cybulski, Feedboxes (2017).....	46
 4. Embedded Soundscaping Implementation and Exhibition.....	 49
4.1 Introduction.....	49
4.2 Implementation 1 – Hardware Instruments.....	49
4.2.1 Technical Overview.....	51
4.2.2 Exhibition Documentation.....	54
4.3 Implementation 2 – Room Installation.....	67
4.3.1 Interactive Call and Response Mode (in a room).....	69
4.3.2 Recursive Acoustic Convolution (in a room).....	72
4.4 Implementation 3 - Digital Material Resonance Experiment.....	75
 5. Art-Science Research: Digital Material Acoustics.....	 79
5.1 Introduction.....	79
5.1.1 Background.....	79
5.1.2 Digital Materials and Digital Cellular Solids.....	80

5.2 Digital Material Resonance Experiments.....	81
5.2.1 Background.....	81
5.2.2 Experimental Setup.....	84
5.2.3 Results and Discussion.....	89
5.2.4 Conclusions and Future Research.....	92
6. Conclusion.....	93
Appendices.....	99
Supplemental Files.....	237
References.....	238

LIST OF FIGURES

Figure 1: Sonic Mirror Instrument System.....	10
Figure 2: Sonic Mirror GUI.....	12
Figure 3: Basic Recursive Acoustic Convolution (RAC) process.....	22
Figure 4: RAC process with constant rate time-stretching.....	23
Figure 5: David Dunn's Sonic Mirror.....	29
Figure 6: David Dunn with a parabolic microphone and his 'Sonic Mirror', 1984.....	29
Figure 7: Alvin Lucier performing <i>I Am Sitting in a Room</i>	34
Figure 8: Gordon Mumma constructing the sound-modifier console for the Osaka Expo '70 Pavilion, New York, February 1970	39
Figure 9: E.A.T.'s full-scale spherical mirror model for the Pepsi Pavilion at Expo '70.....	40
Figure 10: <i>A Room Listening to Itself</i> at the Center for Contemporary Arts in Santa Fe, NM, 2015.....	43
Figure 11: Feedboxes.....	46
Figure 12: Sonic Mirror Hardware Instrument v2.0.....	49
Figure 13: Components of Sonic Mirror Hardware Instrument v2.0.....	51
Figure 14: Laser engraving of 3D spectrogram on side face of wooden Sonic Mirror.....	52
Figure 15: Laser engraving of 2D spectrogram on top face of wooden Sonic Mirror enclosure.....	53
Figure 16: Laser engraving of audio waveforms on side face of wooden Sonic Mirror enclosure.....	53
Figure 17: Sonic Mirror v1.0 and viewers at the Cosmos exhibition, Santa Cruz Museum of Art and History, May 2016.....	56
Figure 18: Two Sonic Mirrors installed at the MFA Now Exhibition, Root Division Gallery, San Francisco, March 2017.....	58
Figure 19: Sonic Mirrors installed at the 4th Annual LAST Festival in San Jose, April 2017.....	59
Figure 20: Four Sonic Mirrors installed at the Stanford CEMEC concert, April 2017	61
Figure 21: Viewer interactions with the Sonic Mirrors, Stanford CEMEC concert, April 2017.....	63
Figure 22: Sonic Mirrors installed at the UCSC CEMEC concert, April 2017.....	63
Figure 23: Sonic Mirrors at <i>ultraSHIFT</i> , Digital Arts and New Media MFA Show, UC Santa Cruz, April 2017.....	64
Figure 24: Sonic Mirror Hardware Instrument Installation Configuration, <i>ultraSHIFT</i> , Digital Arts and New Media MFA Show, UC Santa Cruz, April 2017.....	64
Figure 25: Children interacting with the Sonic Mirror hardware instrument installation at the <i>ultraSHIFT</i> exhibition.....	65

Figure 26: Visitors listening to the Sonic Mirror Room Installation at the <i>ultraSHIFT</i> exhibition, April 2017.....	67
Figure 27: Sound system configuration of the Sonic Mirror Room Installation at the <i>ultraSHIFT</i> exhibition (overview).....	68
Figure 28: Sound system configuration of the Sonic Mirror Room Installation at the <i>ultraSHIFT</i> exhibition (side view).....	68
Figure 29: Interactions with the Sonic Mirror room installation at the <i>ultraSHIFT</i> exhibition...	69
Figure 30: Sonic Mirror Digital Material Resonance Experiment at the <i>ultraSHIFT</i> Digital Arts and New Media MFA 2017 exhibition.....	75
Figure 31: Embedded piezo disc transducers were used to transmit and record sound through the robot's digital material structure.....	76
Figure 32: Raspberry Pi and connected audio circuitry used to operate the automated Digital Material Resonance Experiment installation.....	76
Figure 33: Examples of digital materials.....	80
Figure 34: Digital cellular solid voxels used in DANCER Lab research.....	82
Figure 35: Worm Robot Position 1: Configuration 1 & 2, Unbroken & Broken Voxels.....	86
Figure 36: Experimental setup of RAC testing within quadrupedal robot.....	87
Figure 37: Audio hardware configuration for RAC experiment within DCS structures	87
Figure 38: Recursive acoustic convolution process within the DANCER Lab quadrupedal robot's digital material structure.....	88
Figure 39: Spectrogram of 8 repetitions of the RAC process within DANCER's Worm Robot – Configuration 1 – Unbroken Voxels.....	89
Figure 40: Spectrogram of 3 repetitions of the RAC process within DANCER's Worm Robot – Configuration 2 – Broken Voxel.....	89
Figure 41: Spectrogram of 20 repetitions of the RAC process within DANCER's quadrupedal voxel robot. Initial sound source: five seconds of sustained white noise.....	90

LIST OF APPENDICES

Appendix A: Sonic Mirror Program Code.....	99
Appendix B: Hardware Components and Circuits.....	229
Appendix C: Select Laser Cut and Engraving Designs for Sonic Mirror Hardware Instrument (Implementation 1).....	235

ABSTRACT

Embedded Soundscaping

Scott Tooby

Embedded Soundscaping encompasses interdisciplinary research and a series of three sound installations realized with the *Sonic Mirror*: an interactive digital audio software instrument created for applications in generative sound and music composition, sound installation design, and acoustic experimentation. Capable of real-time digital audio recording, analysis, and synthesis, the instrument's functionality is open-ended and highly reconfigurable for both performative and autonomous contexts.

Embedded Soundscaping presents the Sonic Mirror in two primary functional modes: first, as an interactive instrument that automatically transforms the sounds of its environment into musical soundscapes; second, as an automated system for performing processes of recursive acoustic convolution to reveal the resonant frequencies of physical structures.

The latter of these functions has been applied in the context of an interdisciplinary research collaboration with engineering researchers of UC Santa Cruz's Dynamics, Autonomous, Surface Engineering, and Robotics (DANSER) Lab, to explore the feasibility of using acoustic means to analyze the structural health of *digital cellular solids*, a type of *digital material*. Created by researchers at NASA's Coded Structures Lab, digital cellular solids are currently the strongest ultra-light fiber composite digital material available for applications in fields ranging from robotics to aerospace engineering.

Altogether, this work integrates elements from the fields of electronic music, acoustics, engineering, sound art, and soundscape ecology to define *Embedded Soundscaping* as a technical process and artistic practice for using technology to create interactive audio systems that interface with the soundscapes of different environments.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Professors; Larry Polansky, Mircea Teodorescu, and Yolande Harris, my thesis committee, for their invaluable guidance, time, and thoughtful critiques throughout this project research.

Additionally, I would like to thank the DANM MFA program staff; Kristin Erickson, Kristen Gillette, Steve Gerlach, Bennett Williamson, Felicia Rice, as well as DANM directors (past and present); Michael Chemers and Edward Shanken, for their tireless efforts supporting this one-of-a-kind digital arts research institution.

My grateful thanks are also extended to Professor David Dunn for his invaluable guidance and encouragement throughout the realization of this project, as well as for his work in the fields of music, art, and science, which inspired the initial concepts underpinning this project.

Many thanks to Scott Makson, David Kant, Andrew Smith, Tyler Clinkaberry, and Katelyn Stone for your knowledge and technical support. Without you, much of the work detailed here would not have been possible. I would also like to thank the DANM 2017 cohort and faculty Jennifer Parker, Soraya Murray, Jennifer Gonzalez, Warren Sack, and Elliot Anderson, for the knowledge, stimulating perspectives, opportunities, laughter, and friendships I have been fortunate to experience with you during my past two years here at DANM.

Lastly, I wish to thank my parents and Sayuri for their unconditional support.

1. Introduction

"All the waveforms faithfully transmitted to our auditory cortex by the ear and its mechanisms constitute our immediate soundscape.

Though we may not be of it, we are in it." — Pauline Oliveros

The whisper of wind through fields of grass; the snap of a twig in a forest; the explosive roar of thunder; the cacophony of city traffic; the trills, whistles, warbles, and crooning's of birdsong; dogs barking; babies crying; bacon frying; and on, and on, we are always immersed in a universe of sound. Sound perceptualizes and mediates interactions within an environment. All of the component sounds we hear in our environment sum to create *the soundscape*.

Described herein is a body of interdisciplinary research entwined with this notion of the soundscape that I describe as *Embedded Soundscaping*. Embedded soundscaping is comprised of several aspects: it is a technical process, an artistic practice, and the name of a collection of sound installations realized with a new electronic instrument to experiment with and exhibit some of the processes and practice of embedded soundscaping.

In regards to the technical process, embedded soundscaping involves the installation and operation of autonomous, audio-reactive, digital electronic instruments within an environment, which use sound to intentionally interact with the soundscape in response to its sonic activity. These instruments may be of an "embeddable" nature in the sense that they are designed to be installed within a physical environment and may also employ modern

embedded computing and digital audio technology. By incorporating digital audio software and hardware technology, these instruments can interact with their enclosing soundscape through processes of real-time audio processing, recording and sampling, synthesis, and audio feedback. As a result, such instruments can make sonic activity in the environment more apparent and embodied, and present a platform that facilitates automated real-time sonic interaction between listeners, the instrument, and their environment's soundscape.

The practice of embedded soundscaping involves the design and implementation of audio-reactive instruments, as described above, that can intentionally effect and interact with the sonic activity of a soundscape. This is an emerging and experimental practice, but from my experiences so far, it presents a description of related techniques for using audio technology to augment, facilitate interactions with, and/or make listeners more aware of the soundscape of their environment. For an instrument to be able to “intentionally” interact with the soundscape, it must be able to perceive and react to sonic patterns of activity occurring within it. To help further explicate the artistic practice dimension of embedded soundscaping, I shall clarify the origins of the term “soundscape.”

“Soundscape” as understood today was first introduced by Boston city planner Michael Southworth in 1969 in a research study of the effects and perception of Boston's soundscape on humans (Southworth, 1969). Eight years later the definition was expounded by composer R. Murray Schafer in his influential book, *The Tuning of the World*, wherein he defines the features

of the soundscape in an argument inspired by Pythagorean and Cagayan ideas of the cosmos as a universal musical composition, and formulates the principles of “acoustic ecology” out of concepts drawn from the fields of modern literature, music, and audio theory (Schafer, 1977; Schafer, 2004). Schafer's framework helped seed the beginnings of the interdisciplinary fields of “acoustic ecology” and “soundscape ecology” wherein other composers, artists, bioacousticians, and researchers have continued to advance research into the effects of environmental sound on humans and animals (Wrightson, 1999; Truax, 1978; Pijanowski et al, 2011; World Forum for Acoustic Ecology, 2017).

Bioacoustic researcher and field recording artist Bernie Krause has since elaborated on Schafer's definition of “soundscape” to incorporate new categorizations of sound in terms of “geophony,” “biophony,” and “anthropophony.” This terminology classifies all sounds generated by non-biological, biological, and human sources, respectively (Krause, 2008).

Crucially, a significant definition of “soundscape” was advanced by composer and acoustic ecologist Barry Truax, who emphasizes a communicational approach to acoustic studies and the significance of organisms' listening perception in relation to the information being communicated within a soundscape. Truax uses the term “soundscape” to connote “how that environment is understood by those living within it,” and that sound mediates or creates relationships between listeners and the environment (Truax, 1984, p. 9-11).

Krause's meticulous field recording research further demonstrates this

concept of soundscape acting as a mediating element between the interconnected communications of animals' bioacoustic signals in an ecosystem through his "niche hypothesis." From his extensive field recording career, Krause has documented the soundscapes of an immense array of ecosystems. Through his analysis of audio recordings and their resulting spectrogram data, he has observed the pattern that all sound-making animals in an ecosystem occupy distinctive "niches" in the frequency and temporal domains of the audio spectrum to maximize their communicative abilities, and the cumulative soundscape of an ecosystem is indicative of its general health. Krause posits that to attempt to interpret the communicative significance of the individual calls of a specific animal within a larger ecosystem is a futile task unless one considers it within the larger context of the collective soundscape (Krause, 1987; Krause, 1999; Krause, 2002).

This concept that an integral component of a soundscape is the information of relations encoded within the totality of its constituent sounds has influenced my conception of the artistic practice of embedded soundscaping. In other words, if a soundscape is partially a result of the auditory signals of collectives of organisms, these organisms must in one way or another be understanding the systems of relations encoded within the soundscape. So, in regards to the process and practice of embedded soundscaping, if we are to create instruments to interface with the soundscape, these instruments must in some form or another be sympathetic, perceptive, and/or responsive to the dynamics of their enclosing soundscape in order to shape or interact with it in an intentional way. From my collective

practice developing the instruments, sound art, and research chronicled within this thesis, I have come to the realization that the mechanism of audio feedback is key to designing instrument systems that can effectively interact with a soundscape as a form of embedded soundscaping.

At a more general level, this thesis encompasses a constellation of interdisciplinary research in sound, acoustics, and the cross-pollination of creative inspiration between the fields of music, sound art, and science. The primary element of study connecting these seemingly disparate areas is the process of audio feedback, which is intrinsic to the practice of embedded soundscaping. Specifically, at an aesthetic and conceptual level, I am interested in the emergent interactions that audio feedback can generate in physical and biological systems. In other words, the questions are asked: how can different forms of audio feedback change and/or reveal phenomenon and features in an environment? What, if anything, can be observed and learned from experiments involving audio feedback in relation to a soundscape? Can embedded soundscaping technologies utilized for creative experimentation with audio feedback be applied in the context of scientific research and applications?

To clarify, I would like to define the term “audio feedback” in a broad sense so as to encompass not only the conventional definition of the phenomenon that can occur in an audio system when its output is directed back into its input (e.g. the familiar, sometimes painfully unsettling oscillating tone generated when a microphone is held too close to a connected speaker), but to also include the simple process of playing a recorded sound from an

environment back into said environment. In essence, to treat the larger environmental soundscape as a form of audio system whose audio output can be fed back into itself, allowing for the inclusion of the basic process of audio recording and playback as a form of delayed environmental audio feedback, which can be facilitated by even the most basic audio recording technology (e.g. a tape or digital audio recorder).

Through the creation of a new software instrument, the *Sonic Mirror*, I have developed an open-ended, automatable, and interactive digital audio system that can implement experiments with environmental audio feedback and represents a prototype of an embedded soundscaping instrument. It can be utilized for the creation of interactive sound installations, musical performances, and potentially acoustic testing for engineering applications, which I will explain in more detail below. Presently, I have configured the Sonic Mirror to automatically generate musical soundscapes from the sounds of its environment through interactive processes of audio feedback and synthesis. It can also perform automated processes of successive audio recording and playback to probe and generate resonant frequency responses in physical structures.

In the course of developing this thesis, a few notable works by composers of modern experimental electronic music have influenced my own work by exhibiting how various forms of audio feedback can manipulate animal and insect behavior (at times with deadly results) or alternatively cause the emergence of resonant frequencies in physical spaces. Of primary influence is David Dunn's original development of his Sonic Mirror instrument

and the pieces of music he created with it, and also the dramatic results from his research into the effects of sound on invasive bark beetle species and its applications as a means of pest management. Additionally, Alvin Lucier's piece, *I Am Sitting in a Room*, has acted as a secondary source of inspiration demonstrating how the effects of successive audio recording and playback can evoke the resonant frequencies of physical spaces.

Between these works I have noticed a pattern in how artists can potentially act as cross-pollinators of ideas and technical skill transfer between artistic and scientific fields. This realization has generated another significant thread I'd like to address in my thesis about the intermediary function artists can perform within art-science research.

In parallel with my development of the Sonic Mirror instrument system, I have engaged in an interdisciplinary research collaboration with the Dynamics, Autonomous Navigation, Surface Engineering, and Robotics (DANSER) Lab at UC Santa Cruz to assist with the study of *digital cellular solids*, a new material technology developed by researchers of NASA's Coded Structures Lab for applications in aerospace and robotics engineering. In traversing the fields of sound art, electronic music, and engineering, I discovered that I could apply elements of the Sonic Mirror audio technology I have developed, which was initially conceived in an artistic and musical context, towards the DANSER Lab's engineering research to assist with the study of the unique properties of digital cellular solids via the examination of their resonant frequencies.

Between this engineering research and the variety of interactive sound

installations I have designed with the Sonic Mirror instrument, I have been able to conduct a limited survey of the different effects that processes of audio feedback can achieve in human interactions and acoustic spaces. More generally, this research exemplifies the facets of the technical processes and artistic practice of embedded soundscaping. Hereafter, I will attempt to answer the questions this thesis has raised through an examination of the technology I produced, its exhibitions, and potential engineering applications. I will also reflect on pieces of music and interactive sound art that have either influenced my own work or have become known to me after the exhibition of this work, which similarly integrate elements of audio feedback and could be framed in terms of the process and practice of embedded soundscaping.

Section 2 of this thesis includes a detailed overview and technical description of the functionality of the primary product of this work, the Sonic Mirror instrument. Section 3 outlines a collection of artistic influences within the fields of experimental music and sound art to contextualize the development and implementations of the Sonic Mirror instrument and the practice of embedded soundscaping. In this section, I also examine two examples of creative interchange between artistic and scientific fields, and chart the flow of inspiration that led to my development of an operational mode of the Sonic Mirror that was subsequently applied within my interdisciplinary collaboration with the DANSER Lab. Section 4 features descriptions and technical documentation of the different implementations of the Sonic Mirror instrument, as well as documentation of the various

exhibitions of this work. Section 5 includes a detailed accounting of the background, experimental setup, and results of my interdisciplinary research collaboration with the DANSER Lab. Section 6 offers concluding remarks and a reflection on the thesis research as a whole.

2. The Sonic Mirror Instrument System

2.1 Introduction

At its core, the Sonic Mirror is a digital audio software instrument created for applications in generative sound and music composition, sound installation design, and acoustic experimentation. Programmed with the SuperCollider language, the instrument is capable of real-time audio recording, analysis, and synthesis. Because it is a piece of digital software, its functionality is highly reconfigurable.

The principle components of a complete Sonic Mirror instrument are:

- 1) a computer operating the Sonic Mirror software
- 2) an audio input transducer (e.g. a microphone or piezo contact mic)
- 3) an audio output transducer (e.g. a speaker or PA system)

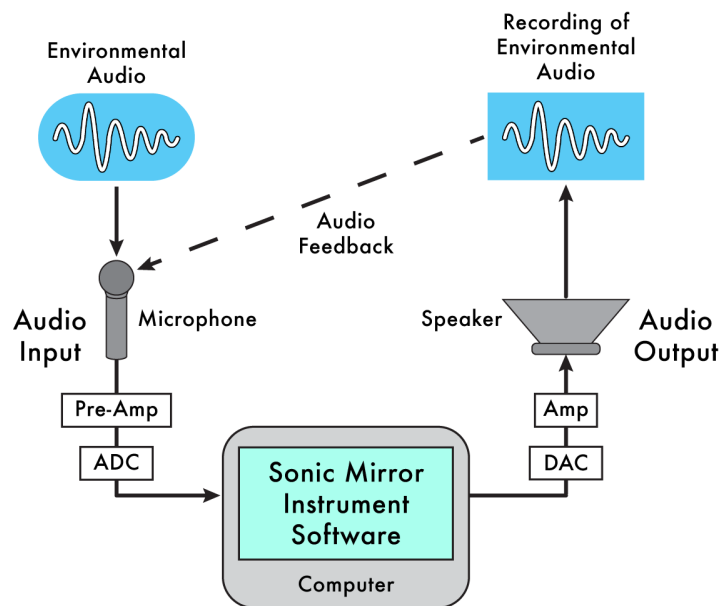


Figure 1: Sonic Mirror Instrument System

In more specific terms, the Sonic Mirror, as its name implies, is an instrument that is reflective of the sounds in its environment. It is often configured as an interactive audio feedback system and is capable of transforming sounds it records into more musical soundscapes. The instrument is interactive in the sense that its audio recording, processing, and playback functions can be triggered and sequenced in response to detected sound events (or the absence thereof).

As a general overview of one of its basic functionalities, when the instrument “hears” a sound event in the environment, it records the detected sound, optionally applies various digital audio transformations to the recording, and then plays back the resulting audio.

Integral to its functionality, the instrument features a variety of basic real-time machine listening algorithms which allow it to detect and analyze the audio features of sound events in its environment. Audio features such as loudness, pitch, transient onsets, duration, and more can be analyzed by the instrument to deduce the presence and characteristics of sound events.

Initially, the instrument was conceived as a system that could autonomously interact with its environment's soundscape through sound. To do this, it would need to be able to sense sounds and determine some salient audio features of detected sounds (e.g. pitch, loudness, duration, timbre, etc.)

In practice, its digital nature and minimal hardware requirements allows the instrument to manifest in an open-ended variety of physical configurations ranging from discrete hardware instruments to larger-scale room installations.

Three unique Sonic Mirror implementations were developed during the course of this research:

- 1) Sonic Mirror as a self-contained hardware instrument
- 2) Sonic Mirror as an interactive sound installation in a room
- 3) Sonic Mirror as an acoustic testing system for digital material structures (but this implementation can be extended to other physical structures as well)

All three of these implementations were exhibited the *ultraSHIFT* Digital Arts and New Media 2017 MFA exhibition at UC Santa Cruz in April 2017. These pieces are described in more detail in section 4, and the related digital material acoustic research is described in section 5.

2.2 Software Instrument Design

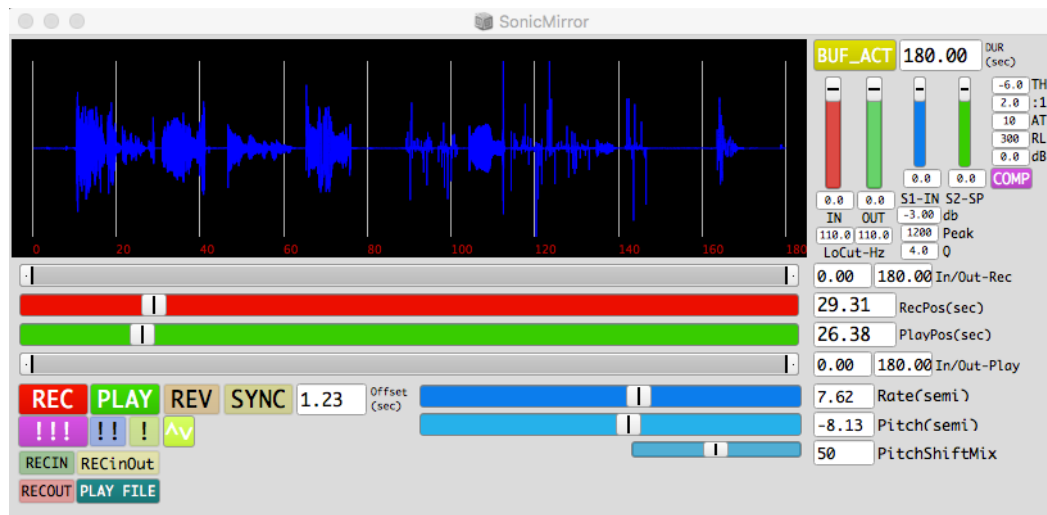


Figure 2: Sonic Mirror GUI

The Sonic Mirror software has been an active development project for

the past year, and new features are continuously being added. The project is openly shared online (Tooby, 2017b), and the full text of the current version of the program code is also documented in Appendix 1.

At its core, the instrument is designed around an architecture of basic audio buffers. One main audio buffer of configurable duration (e.g. 60 seconds) is available for streaming recorded audio. Audio from this buffer can then be played and variably processed via playback rate modulation, granular pitch shifting, basic filtering and dynamic compression.

A layer of basic audio feature extraction algorithms can optionally be engaged to analyze streaming audio input and detect discrete sound events. A combination of amplitude, pitch, and transient onset detection algorithms are used to reliably detect the presence of distinct sounds. If a sound is detected within the main audio buffer by the system, it can then be automatically recorded and written to the computer's memory as a high-resolution digital audio file.

As the instrument detects and samples sound events from its main buffer, it gradually accumulates a library of sound files that it can then optionally process and play back in parallel or in lieu of the main streaming audio buffer recording, processing, and playback. Effectively, during its interactive modes of operation, the Sonic Mirror exhibits two parallel “streams” of audio recording and processing: the first being the output from the live streaming buffer, and the second consisting of the combined output of sampled audio file processing and playback. Within both of these streams, a multitude of program control routines can variably modulate and sequence

the triggering of the instrument's audio effects processing and playback parameters over time.

In summary, presently, the Sonic Mirror software instrument features a number of basic audio playback and effects processing control parameters operating on each active audio buffer (e.g. the main streaming audio buffer and any additional buffers resulting from audio file playback). These effects parameters can be configured manually via the GUI controls or automated via control routines to achieve various forms of modulation synthesis (e.g. AM and FM). Additionally, the software instrument also features more advanced forms of audio processing effects which can be variably triggered in response to detected sounds or the lack thereof.

Basic playback and effects parameters (per audio buffer):

- Playback position (variable audio buffer scrubbing)
- Playback rate (varispeed)
- Pitch shift (via granular synthesis)
- Volume control
- Dynamic processing (audio compression)
- Basic filtering (EQ)

Advanced Effects:

- Modulation synthesis:

When this effect is triggered, LFO waveforms (e.g. sine, square, sawtooth, triangle, and various noise forms) are used to modulate the above-

mentioned basic effects and playback parameters of an audio buffer. This effect can also be configured to incorporate an additional LFO to modulate the initial LFO modulation source for more chaotically time-varying and dynamic modulation effects.

- Cross-synthesis (Partitioned Convolution):

In this process, the spectrum of one recorded sound is filtered by the spectrum of another. The result is a hybrid sound where the spectral or timbral characteristics of one sound are imparted on another. For example, if the instrument convolves the sound of a singing voice with the sound of rhythmic clapping, the result may sound as if the original voice sample is sounded in synchrony with the original rhythm of the claps to create the illusion of a dense choir of layered multiples of the original voice. In this particular case, the resulting sound has the effect of an audio delay, but this is achieved purely through the strict operation of audio convolution.

- Concatenative Sound Scene Audification:

In this process, a configurable number of recorded audio samples are time compressed, sequenced, and played successively in a very short amount of time without changing the original pitch of the recorded sounds. For example, 30 audio samples that may collectively sum to a total duration of 60 seconds or more are played back at a significantly increased playback rate such that the resulting playback duration spans only 10 seconds, thus rendering a basic form of sound file library audification. As the Sonic Mirror

accumulates sounds recorded in a given space over time, this specialized effect renders a temporal distillation of the larger soundscape the instrument has observed.

2.3 Operating Modes

Specific functionalities and reactive behaviors are comprised of combinations of programmed functions and routines (timed sequences of functions), collectively referred to as distinctive “modes” of operation.

As of this writing, the Sonic Mirror instrument features two primary modes:

- 1) Interactive Call and Response Mode
- 2) Recursive Acoustic Convolution Mode

2.3.1 Interactive Call and Response Mode

In this mode, the Sonic Mirror operates as a form of delayed feedback system that emits transformed versions of recorded sounds in response to detected sounds. When a sound is “heard” by the system, a transformed version will subsequently be output, rendering the effect of a call and response dynamic between the instrument and external sound source.

First, audio is continuously recorded to an audio buffer of configurable duration (e.g. 60 seconds). Basic machine listening algorithms extract features from detected sounds (e.g. loudness, pitch, duration, etc.), and based on the analysis, the system generates a musical soundscape from recorded sounds using a variety of audio synthesis techniques (e.g. pitch

shifting, time stretching, convolution, granular and concatenative synthesis).

The resulting audio output of this process is dependent on the nature of the recorded input sound. The duration of the recorded sound is the primary feature the system uses to determine which audio processing effects are applied to the sound, followed by loudness and pitch to lesser degrees. Beyond this, audio effects processing is applied in a fairly pseudo-randomized manner.

To describe the sonic qualities of this mode is a difficult task due to the fairly randomized nature of its output. One can never quite expect what the exact result may sound like. As an example though, the instrument may process the sound of a human male tenor's singing voice by pitch shifting it into a lower range and additionally modulating the pitch to impart a subtle or dramatic vibrato that didn't exist in the original signal. The sound of fingers snapping a rhythm may experience a large modulation in playback rate such that its pitch is transposed upwards an octave, resulting in a sound more reminiscent of a maraca or the chirps of a cicada. Speaking voices are a frequent form of input source by users during sound installation exhibitions, and the system's processes often dramatically shift a recorded voice's vocal register and cadence. At the extremes, a full grown adult male's voice can seemingly be transformed into the murmurings of child or chipmunk or conversely with subtler transformations, an adult female's voice may be transformed into a male tenor's voice. Ultimately, words can only go so far in describing the sonic qualities of this process and all the others described in this thesis, which is why I recommend listening to the documented recordings

(Tooby, 2017a).

Interactive Call and Response Mode - Step-by-step process:

1. Live audio input from a hardware microphone is continuously recorded to an audio buffer of a configurable duration (e.g. 60 seconds), hereafter referred to as “Buffer 0 (B0)”.
2. Audio recorded to B0 is continuously played back and processed in a variety of pseudo-randomized configurations in real-time. Playback rate, pitch, and buffer position are randomly modulated at randomized time intervals. Depending on the physical configuration of the instrument's hardware microphone and speaker, its audio output can feed back into its audio input and be re-recorded into B0, compounding the effects of the live buffer processing, which can yield highly complex and dense sonic textures.
3. Audio analysis algorithms detect the occurrence of discrete sound events within B0 and sample them as high-resolution (48kHz / 24-bit / .wav) audio files to the computer system's hard drive memory. A combination of transient onset detection, frequency tracking, pitch detection, and amplitude tracking algorithms are used to detect the presence of sound events with reliable accuracy.
4. After a sound is detected by the audio analysis algorithms and saved to the computer's memory, the resulting audio file is processed and played back. The audio processing that occurs is selected based on the relative duration of the sound and pseudo-randomized selection of one of the “advanced” audio effects processes outlined in the section above. This

process occurs in parallel with the live audio processing of B0 outlined in “Step 2” above.

5. Over time, if sounds are not detected by the instrument, it will periodically output a processed version of a previously recorded sample or perform Concatenative Sound Scene Audification (defined above) of its entire recorded audio sample library. This serves the dual function of rendering a form of sonic distillation of the instrument's soundscape over longer time periods, as well as a sonic signal potentially making nearby individuals aware of the instrument's presence and interactive nature.

2.3.2 Recursive Acoustic Convolution (RAC) Mode

In this mode, the system repeatedly records and plays back an initial sound source into an acoustic space while re-recording the result. First, a sound (e.g. a synthesized test tone or live audio input from an acoustic instrument) is played into an acoustic space and recorded. Next, the resulting recording of the original source sound being played through the acoustic space is then played back in the same acoustic space and re-recorded. This process repeats indefinitely in a recursive manner wherein the resulting recording of each playback cycle is then played back through the structure and re-recorded in the subsequent cycle. With each repetition of the process, the resonant frequencies of the total system (i.e. the acoustic space *and* audio system) are increasingly excited and revealed. The acoustic space acts as a natural filter so that with each iteration of the RAC process, the space accentuates the frequencies generated by the audio output signal that

overlap with the resonant frequencies of the space while attenuating the frequencies of the audio signal that do not.

To better understand the process of Recursive Acoustic Convolution, we must clarify the definition of “convolution.” In this context, the specific process of audio convolution is implied, and its original definition originates from the field of digital audio signal processing (Rabiner & Gold, 1975; Dolson, 1985; Oppenheim & Schaffer, 1975; Oppenheim & Willsky, 1983).

Everyone is familiar with the effects of convolution, which are often described in terms like filtering, reverberation, modulation, or cross-synthesis. For example, any filter *convolves* its impulse response with the input signal to produce a filtered output signal. In other words, the convolution of two audio signals is equal to the multiplication of their spectra and vice versa. Effectively, the convolution of two audio signals is equivalent to filtering the spectrum of one by the spectrum of the other (Roads et al, 1995, p. 424).

In the case of RAC, the convolution is occurring at an acoustic or analog level outside of the digital domain in real-time with each repetition of the process. As the process unfolds, the initial source sound (e.g. a human voice speaking) seemingly transforms into rich harmonic texture comprised of the resonant frequencies of the acoustic space and audio system. In essence, this feature of the Sonic Mirror software instrument is a digital automation of the same process pioneered in *I Am Sitting in a Room*, a piece of experimental music composed by Alvin Lucier in 1969. The influence of this piece on my development of the Sonic Mirror instrument is described in more detail in Section 3.3.

Step-by-step Procedure of the Sonic Mirror's RAC Function:

1. System audio input is continuously recorded into an audio buffer
2. Sound detection algorithms monitor the streaming audio input
3. When a sound is detected by the sound detection algorithms, it is recorded and saved in the system's memory as a digital audio file. It is referred to as the “source sound.”
4. The recorded source sound is then played back through the system's audio output (with optionally configurable playback rate modulation, pitch shifting, and other audio effects processing) and simultaneously re-recorded by the system's audio input.
5. The preceding step is repeated N number of times, always playing back the previous recording with each repetition.

The step-by-step process of RAC for automated performance is the same as above except a pre-recorded or synthesized source sound is initially output by the Sonic Mirror before proceeding with steps 3 – 5. This automated version of RAC occurs in Implementation 3 and the digital material research experiments described in sections 4.4 and 5.2.2, respectively. In these cases, white noise and sawtooth wave sweeps are used to test the impulse response and maximally excite the resonant frequencies of digital material structures.

The following diagram symbolically visualizes the process of basic RAC. Here “*” signifies the operation of acoustic convolution:

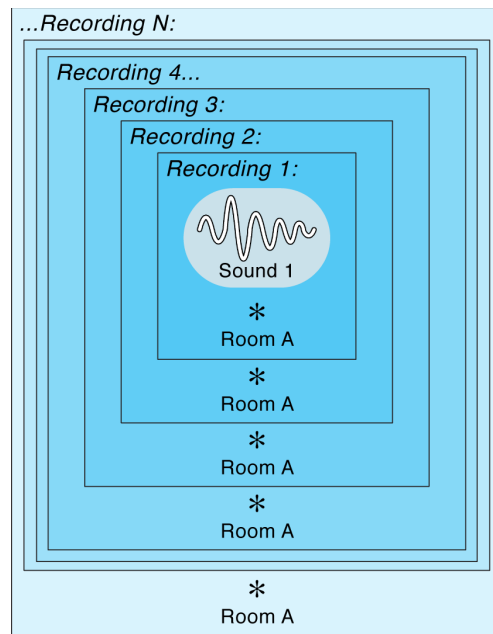


Figure 3: Basic Recursive Acoustic Convolution (RAC) process

The RAC process performed in Implementation 2 of the *Embedded Soundscaping* installations (see section 4.3) also involves pitch invariant time-stretching with each iteration. Specifically, the preceding recording played back in a given iteration is played at a slower playback rate with compensatory upwards pitch shifting to preserve the recorded sound's original pitch. During the course of development, I found that incorporating the use of subtle amounts of time stretching seemed to help accelerate the apparent effects of the RAC process.

In its most basic form (as in the case with Lucier's "I Am Sitting in a Room") the RAC process can require longer amounts of time on the order of tens of minutes (e.g. tens of repetitions) before its transformative effects become apparent, whereas the RAC process with time stretching enables the effects of the process to become apparent within the first several iterations.

As each audio recording is time-stretched slightly during each playback cycle, the total playback duration increases with each repetition, allowing listeners more time to hear the effects of the process during its earlier phases. Additionally, playing recorded sounds more slowly within the acoustic space they're being convolved with seems to exacerbate the filtering effects of the space's acoustics, which further contributes to the apparent acceleration of the effects of the RAC process.

The following diagram symbolically visualizes the process of RAC with time-stretching. (“*” signifies the operation of acoustic convolution):

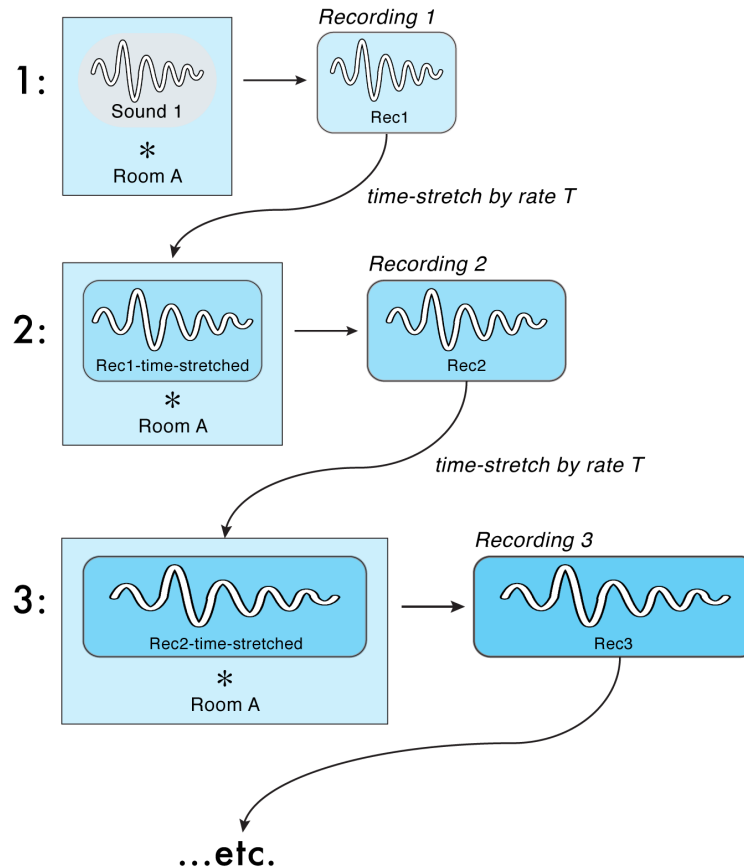


Figure 4: RAC process with constant rate time-stretching

This mode is operational within Implementation 2 and Implementation 3 of the *Embedded Soundscaping* installations described in sections 4.3 and 4.4. It also promises to be an effective means of detecting the resonant frequencies of a structure, which could potentially allow the Sonic Mirror instrument to be utilized for acoustic testing and monitoring applications. The application of the Sonic Mirror for this purpose has been researched in the context of an interdisciplinary engineering collaboration, which is described in detail in section 5.

2.4 On-going and Future Development

As mentioned earlier, this software is still an active development project, and a number of features are planned for future implementation:

- Automated acoustic impulse response sampling
 - Offline Recursive Convolution using sampled impulse responses
 - Dynamically configurable multi-channel audio throughout the software's architecture (large parts of the system are currently constrained to mono)
 - Longer term auditory event memory
 - OSC support to enable the outputting of audio feature data as OSC for local inter-application communication and over a network.
 - Interfacing with machine learning tools like Wekinator or Google's Magenta to expand the Sonic Mirror's audio analysis and synthesis capabilities (Wekinator, 2017; Magenta, 2017).
- To be able to train future versions of the Sonic Mirror instrument to

recognize sound events using machine learning algorithms could prove to be a monumental leap forward in terms of capabilities and development speed. An initial proof-of-concept of a derivative SuperCollider project demonstrates an earlier version of the Sonic Mirror software interfacing with the Wekinator machine learning software tool (Tooby, 2016).

3. Influences and Context

3.1 Introduction

This section outlines artistic influences and how they informed the development of the Sonic Mirror instrument and its various implementations. Aside from the initial functional inspiration of David Dunn's original Sonic Mirror instrument, most of the other works listed did not become known to me until after the showing of my work at the *ultraSHIFT* Digital Arts and New Media 2017 MFA exhibition. While I may not have consciously been aware of many of the works detailed in this section during the development of my thesis, I've selected them as points of comparison because they all engage with systems of audio feedback in one form or another, and as a consequence exhibit characteristic elements of the process and practice of embedded soundscaping to varying degrees. While I don't contend the qualities I highlight are the primary foci of the artists whose works I have included for comparison, the common thread I have observed between my own work and these others is a recurring exploration of the emergent interactions that can occur as a result of processes of audio feedback, both in the contexts of the physics of sound as well as the behavioral responses of living organisms. In plotting these interconnections and similarities of process between artistic influences, I aim to clarify and better contextualize the practice of embedded soundscaping.

Upon further reflection, the defining factor that was perhaps most influential on the development of this work was my immersion in the interdisciplinary culture of the Digital Arts and New Media program and the larger research institution of the University of California, Santa Cruz.

Occupying roles in both a soundscape ecology lab as well as a robotics engineering lab exposed me to a broad spectrum of research and discourse while traversing the fields of bioacoustics, soundscape ecology, sound art, electronic music, biologically inspired robotics design and engineering. At this intersection of science and art, I experienced a routine free-flow of ideas and transferable technical skills from one field to another. This experience ultimately led to the third implementation of the Sonic Mirror used for the acoustic study of digital materials (described in more detail in section 3.3, 4.4 and 5).

As an additional thread of context, a pattern of artists and composers acting as cross-pollinators of ideas and technical processes between artistic and scientific fields (or vice versa) became increasingly evident to me during the course of my research. This is a lens by which I'd like to examine some of the included works, and I will elaborate further on this topic in subsequent sections.

3.2 David Dunn's Sonic Mirror and bark beetle research

David Dunn is a composer, field-recordist, and bioacoustic researcher whose work spans the realms of music, interactive environmental sound experiments, and more recently, entomological research of the applications of acoustic technology for pest control. An underlying theoretical current connects his work to nonmusical disciplines such as experimental linguistics, cognitive ethology, cybernetics, and systems philosophy. A central aspect of Dunn's environmental sound work entails applications of audio technology to facilitate interactive experiments in interspecies communication and observe

the emergence of systems of mind in natural systems (Dunn & Lampert, 1989). In Dunn's own words:

“My own work has focused upon using advanced audio technology as a tool to facilitate interaction with other living systems. Through combinations of analog, digital and traditional sound-generating devices, I have designed real-time performance interactions in wilderness spaces where the resulting events are reflective of a larger system of mind, which includes myself and other living systems. I place the emphasis upon understanding the resultant sounds as contextually bound; they are the evidence of purposeful, living systems with attributes of mind. One of the metaphors I have adopted to describe this work is that I am applying current technology toward a rediscovery of 'natural magic'. In this case, it is a marriage of music and electronic technology, which serves to invoke a relationship to nature that is both ancient and contemporary. The most recent example of this mode of thinking is a current project entitled *Sonic Mirror*. Rather than a specific composition, this project is an experiment in systems integration which continues my previous work in the application of musical structures toward real-time interaction with non-human living systems. Ultimately, it is planned that the project will culminate in a stationary cybernetic sound sculpture capable of processing acoustic data from a specific environment and then communicating transformations of that data back to the environment. The sculpture eventually would function as an autonomous system, structurally coupled to its surrounding environment in a manner that might allow for 'learning' between components.” (Dunn, 1988, p. 381)

Dunn began development of his Sonic Mirror system in 1981 and completed an initial prototype in 1983. It was a digital audio recording, processing, and playback system operated by a MOS 6502 CPU with a math co-processor chip. Programmed in HEX assembly code, the system featured 12k of memory, and was capable of recording only four to five seconds of 8-bit mono audio (David Dunn, personal communication, June 4, 2017).



Figure 5: David Dunn's Sonic Mirror

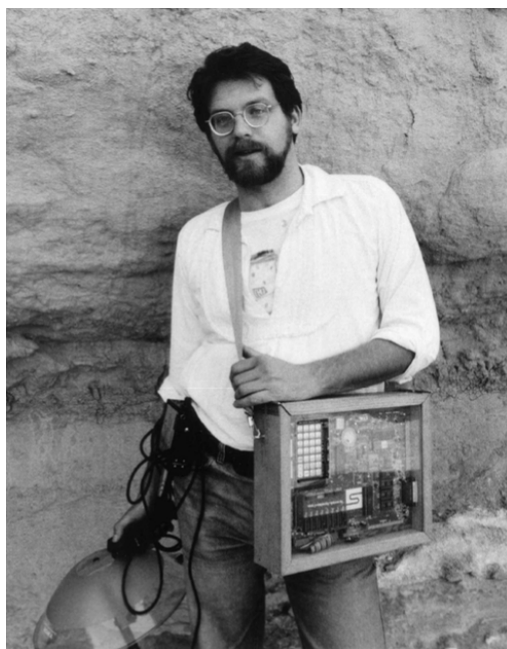


Figure 6: David Dunn with a parabolic microphone and his 'Sonic Mirror', 1984

(Photo: Liz Rymland) (Source: Dunn & Van Peer, 1999)

One of Dunn's earliest experiments with the Sonic Mirror produced *Entrainments 1 and 2* (Dunn, 1996), a pair of pieces derived from the

instrument's processed output of recorded bioacoustic interactions in a forest:

“In the case of *Entrainments 1*, I simply sat in a location and played a single sine-wave oscillator, optimizing the acoustical resonance of the location. It was a very simplistic procedure, the frequency being manipulated over an hour's time. Twenty minutes into doing this, a blue jay came and flew within a foot of my face, and started to scold me. It was very aggressive and very focused, I assume in response to the sound that I was making. The next step was to build upon that in various levels over time. I took that recording – the blue jay with sine-wave oscillations – and fed that back into the location, doing subsequent recordings over time. Each layer became a document of the changes that accrued....It took me several months to do this piece, to build up these layers. By the end it began to function as a chaotic attractor, a basin of attraction to the wildlife that existed there. Rather than being repelled, they were being attracted. I'd add a layer and I'd see these wonderful numbers of birds flying into the location and just listen...This work was a very didactic process, to simply see the transformations and changes that can occur. It is probably similar to what humans have always done as part of the heritage of what we have called music. I don't make claims for this being something extraordinary. In fact I am making claims for it as something highly ordinary.” (Dunn & Van Peer, 1999, p. 66)

Dunn's accounts of performing a series of repeated environmental audio recordings and playback and its effects on birds are very compelling. This is admittedly what initially peaked my interest in his Sonic Mirror project and the general effects of this process of environmental audio feedback.

After Dunn developed an initial Sonic Mirror prototype in 1984, he proceeded to conceive of expanding it into a larger series of stationary cybernetic sound sculptures that could be networked via satellite telecommunications technology to enable the processing and exchange of acoustic and electromagnetic data within and between outdoor environments.

But at the time, technological limitations made the project unfeasible (Dunn & Van Peer, 1999, p. 66).

Being exposed to Dunn's concept of the "Sonic Mirror" initially seeded my interest in designing a new implementation of the instrument for the purposes of carrying out interactive and automated sonic experiments in both natural and unnatural settings (e.g. an art gallery with humans). Additionally, understanding that modern computing and networking technology has advanced considerably since the '80s was also another motivator emboldening me to pursue the creation of a new version of the instrument. Similar to Dunn, I don't aim to ascribe any form of mystical or pseudo-spiritual underpinnings to this work, nor do I claim that the interactions my version of the instrument generates with humans in an art gallery or similar contexts is comparable to interactions with animals in a natural setting, or that it reveals some deep insight about nature. At a conceptual and aesthetic level, I am interested in the emergent outcomes created within physical and biological systems (e.g. acoustic spaces, humans, and animals) through interactions with processes of environmental audio feedback.

Keeping with this train of thought, Dunn's research into the effects of sound on bark beetle reproduction impressed upon me another compelling example of how environmental audio feedback (in this instance the communicative sounds of bark beetles) can dramatically effect the behavior of animal species.

In the early 2000s, Dunn began noticing a dramatic amount of tree deaths in pinyon pine forests around his home state of New Mexico. Out of a

practical concern, Dunn wondered if there was some form of sound related to this activity, which led him to craft a unique transducer solution out of a repurposed meat thermometer that allowed him to record sounds inside trees. Dunn discovered extensive amounts of bark beetle activity within the trees he probed, and he began to meticulously record these sounds and amass a large collection of bark beetle audio recordings over the course of two years. From these field recordings, the sheer magnitude of the bark beetle activity Dunn observed suggested a connection to global climate change (Dunn & Crutchfield, 2009). After reading numerous scientific articles, combined with his own recordings, Dunn publicized his findings in the form of the composition *The Sound of Light in Trees* (2006), a massive sonic collage of his bark beetle recordings depicting the sound world of an infested tree (Gottschalk, 2016, p. 64-65; Dunn, 2006).

Dunn's CD caught the ears of entomologists Richard Hofstetter and Reagan McGuire soon after its release, and they contacted Dunn to inquire about replicating his recording technique for their ongoing bark beetle research. At the time they were the only researchers investigating sound in relation to bark beetle behavior and population dynamics.

Soon after their initial meeting they engaged in collaborative research and the results showed that the sounds of unprocessed bark beetle calls could significantly disrupted bark beetle reproduction and behavior (Hofstetter et al, 2013). Dunn describes the process in detail:

“We played these sounds back to them in phloem sandwiches, which we invented to observe and document bark beetle behaviors in real time with microscopes and video. We noticed some extraordinary things that seemed to

suggest, in one experiment, they were using sound as weaponry to manipulate each other with cross-species effects. And that was entirely when playing back their own sounds. We used the combination of their own sounds, which triggered biological readiness, and sounds from non-linear chaotic oscillators, which were loud enough to mask their own communication and probably be interpreted. What occurred with a combination of these sounds was complete neural breakdown. In a very short period of time all their normal biological behaviors stop and they curl up and die, resulting in 10% attrition at the level of cannibalism. Even within the same species they start attacking each other.” (David Dunn, personal communication, November 18, 2015)

More accounts from Dunn's bark beetle research collaborations are also detailed in Nikiforuk's (2011) book, *Empire of the Beetle*.

In understanding the implications of Dunn's beetle research, I was amazed to learn how the relatively simple process of playing the bark beetle's own bioacoustic signals back at them could cause such dramatic changes in their behavior. In this particular case, the effects of environmental audio feedback have revealed a promising application for pest management. Furthermore, the chain of events leading to this research was initiated by an initially creative and practical impulse on Dunn's part to investigate the sound world of tree interiors. I consider this to be a dramatic example of how artistic experiments can give rise to significant science research and real-world problem-solving applications.

Dunn's beetle research and his Sonic Mirror instrument were the inspirational impetuses that initially led me to conceive of designing an automated system for recording and playing back environmental audio. At first, I approached the idea purely as a design challenge to create a system that may be capable of being implemented in bark beetle and other

bioacoustic research contexts. But as the idea matured I also became interested in designing an interactive audio system that could interface with humans as well. This thought process ultimately led to my inspiration to develop a newer version of the Sonic Mirror instrument. And upon further examination, Dunn's early experiments with his Sonic Mirror instrument in nature demonstrate a representative example of the practice of embedded soundscaping insofar as his use of a specifically designed electronic instrument to interact with a natural soundscape through a system of environmental audio feedback.

3.3 Alvin Lucier, *I Am Sitting in a Room* (1969)



Figure 7: Alvin Lucier performing “I Am Sitting in a Room” (Source: Lovely Music Ltd., 2017)

Alvin Lucier (b. 1931) is a composer of experimental music and sound installations. Many of his works explore acoustic phenomena and auditory

perception. He was a co-founding member of the influential Sonic Arts Union, which included Robert Ashley, David Behrman, and Gordon Mumma, and he has also been a long-time professor of music at Wesleyan University (Lucier, 1998; Lucier, 2017). Much of Lucier's work is influenced by science and explores the physical properties of sound itself: resonance of spaces, phase interference between closely tuned pitches, and the transmission of sound through physical media (Tenney, 1995).

I Am Sitting in a Room (1969) (Lucier & Simon, 2012) is considerably one of his more famous compositions. When this piece is performed, Lucier records himself reciting a text outlining the specific process underlying this piece. He then plays the recording of his speech back into the performance room and simultaneously re-records the played recording of his speech. He then repeats this process recursively such that the previous recording of each cycle is played back and re-recorded. After successive repetitions of this process, the recordings of his voice increasingly excite the resonant frequencies of the room and seemingly transform into a harmonious, sinusoidal, droning texture. Effectively, the recording of Lucier's voice acts as an impulse source to excite the resonant frequencies of the performance room through a process of recursive acoustic convolution.

My awareness of this piece in turn inspired the creation of the 'Recursive Acoustic Convolution (RAC) mode (section 2.3.2) of the Sonic Mirror software instrument after about six months of its prior development. By this point, the software's audio recording and processing infrastructure was sufficiently in place and I felt confident I could implement an automated version of Lucier's

piece to further explore the effects of this process in different acoustic spaces. Once I had established a working prototype, I augmented Lucier's original process by introducing a slight amount of digital audio time stretching with each playback step (e.g. pitch invariant playback rate reduction). I found that this seemed to accelerate the buildup of acoustic resonance inherent in this process, and I began experimenting with different instruments as impulse sources within a large multimedia performance space in the Digital Arts Research Center at UC Santa Cruz. It was here where I began to internalize through repeated testing and performance how this process of recursive acoustic convolution can reveal the resonant frequencies of an acoustic space. While I was developing and experimenting with the Sonic Mirror's RAC mode in a room installation setting, I was concurrently involved with the DANSER Lab's digital cellular solids research (section 5). It was around this time that the idea occurred to me to attempt to adapt the RAC process for operation in digital materials as a means of analyzing their resonant frequencies, which was a task the DANSER Lab was interested in conducting experimentally to verify their computational models of the resonant properties of digital material systems.

Throughout the course of this research I was under the impression that I had been inspired by a work of experimental music and adapted its process for engineering research. However, I have now come to learn of the deeper history behind Lucier's development of *I Am Sitting in a Room*, which I feel adds an interesting contextual layer of depth to its influence on my own adaptation of this work. After exhibiting the Sonic Mirror in its various

implementations at the *ultraSHIFT* MFA show (section 4) and conducting additional research about Lucier's piece for this thesis, I was intrigued to learn that Lucier originally learned about the acoustic process he implemented in *I Am Sitting in a Room* from Edmond Dewan, Lucier's colleague from the Physics Department at Brandeis University (Kahn, 2013, p. 86). Dewan learned about this specific acoustic process after observing a demonstration conducted by engineer and founder of Bose Corporation, Amar Bose, during a presentation at Air Force Cambridge Research Labs. Bose originated this successive recording and playback technique as a means of testing the frequency response and detecting unwanted peaks in the performance of speaker systems. Dewan eventually relayed knowledge of this technique to Lucier a few years later, who at the time was working in the Music Department at Brandeis University, and this idea inspired Lucier's creation of *I Am Sitting in a Room* (Ibid, p. 104).

In realizing that Lucier was inspired by an idea that originated as an acoustic engineering test, I found significance in tracing the full trajectory of how this specific acoustic testing process has traversed scientific and creative fields in relation to my own research. This process conceived by Bose, was adopted by Lucier for *I Am Sitting in a Room*, which in turn inspired my augmentation of the Sonic Mirror instrument to automate and augment the same process with modern digital audio processing techniques, which subsequently led to the idea to adapt the process for the DANSER Lab's digital material research. So in essence, the flow of inspiration and idea transfer was from the field of acoustic engineering to music, music to sound

art, and sound art to engineering. Or in more general terms, the idea's trajectory travels in a relative full-circle passing from the sciences to the arts, within the arts, and finally from the arts back to the sciences.

Lucier's adoption of an engineering test process for the creation of arguably his most well known composition serves as another example of how artists can act as vectors of idea transfer between scientific and artistic fields. Conversely to Dunn's case, with Lucier, the flow of transfer was from the sciences to the arts. And in my exposure and adaptation of Lucier's piece, I eventually was able to propagate the concept of this acoustic testing process towards a new engineering research application.

In relation to the concept of embedded soundscaping, this piece demonstrates how its process of recursive audio feedback can generate a cumulative emergent response within the soundscape of the acoustic environment it occupies. It is, in certain respects, an exercise in process that can be employed by embedded soundscaping systems to interact with a soundscape and reveal its latent acoustic properties.

3.4 Gordon Mumma's Sound Modifier Console (1970)



Figure 8: Gordon Mumma constructing the sound-modifier console for the Osaka Expo '70 Pavilion, New York, February 1970 (Photo by Barbara Dilley Lloyd) (Source: Mumma, 2015)

Gordon Mumma (b. 1935) is a composer and electronic music pioneer who lived in Ann Arbor, Michigan from 1953 to 1966, where he was co-founder of the Cooperative Studio for Electronic Music and also the ONCE Festivals of Contemporary Music. Since 1966 he has performed with the Sonic Arts Union, whose members include Robert Ashley, David Behrman and Alvin Lucier. In addition to his extensive academic career, Mumma is widely known for developing his own electronic instruments for his compositions, and has coined the term “Cybersonics” to describe his practice of live electronic processing of acoustic sound sources as a form of “collaboration” with technology (Mumma, 2017; Mumma, 2015a).



Figure 9: E.A.T.'s full-scale spherical mirror model for the Pepsi Pavilion at Expo '70 (Source: Packer, 2003)

Shortly before the Expo '70 in Osaka, Japan, Mumma was commissioned by the Experiments in Art and Technology (E.A.T.) collective to develop a unique sound processing system for their production of a massively elaborate feat of engineering in the form of a 90-foot, 210 degree spherical mirrored dome structure made of reflective mylar and an audiovisual multimedia installation for the expo's Pepsi Pavilion. The interior of the dome was large enough to create three-dimensional “real” image reflections hovering in and around the audience (Packer, 2003). 37 loudspeakers were positioned in a rhombic grid surrounding the apex of the dome. Connected to all of these speakers was “the Sound-Modifier Console,” a one-of-a-kind analog instrument designed to perform real-time audio effects processing of sounds and musical performances for the entire pavilion installation.

The sound-modifier console was capable of operating on both pre-recorded tape as well as live audio input. It featured eight channels whose inputs could be summed from up to four audio sources each, for a total of 32 possible sound source inputs. Each of the eight sound processing channels featured 12 distinct controls inclusive of frequency modulation, amplitude modulation, and filtering stages. For all of these effects stages, certain parameters such as the pitch shift amount, vibrato amount, amplitude modulation rate, and filter modulation rate could be modulated based on the amplitude envelope of the original sound source input.

Based on Mumma's descriptions, the sonic results could be finely tuned from the subtle to extreme such that an original sound source (e.g. musical sounds from a live instrumentalist) could be rendered into an unrecognizable output, especially if extreme frequency modulation settings were enabled (Mumma, 2015b, p. 70). Based on video documentation of the Pepsi Pavilion, it appears that the immersive atmosphere created by the E.A.T. collective in combination with Mumma's sound-modifier console creates a vast all-encompassing mirror-scape of light and sound reflective from the local environment (Saarinen et al, 1969).

The specific detail that I'm most intrigued by are Mumma's technical descriptions of how the sound-modifier console at times would be processing and diffusing live sounds picked up from within the dome. I would presume that at certain moments, sounds from the crowd would be processed and diffused throughout the space by the console to create what could be described as a form of sonic "fun house" mirror effect. Additionally, as with

any system of audio feedback, intrinsic to the sound quality are the unique resonant properties of the pavilion dome and the configuration of the microphone and speaker arrays (Mumma, 2015, p. 71).

By most accounts, E.A.T.'s mirror dome experience was an intensely reflective environment of light and sound, both in terms of the naturally occurring optics and acoustics of the space, but also in regards to the 'cybersonic' nature of Mumma's sound-modifier console, which at times was 'reflecting' the sounds of visitors in the room through time and space via networks of analog audio systems, circuits, and audio feedback. I contend the sound-modifier console's ability to create a sophisticated soundscape emergent from an interactive audio feedback network is central to making the Pepsi Pavilion installation an ultimate form of sonic mirror. Furthermore, in understanding that Mumma designed the console for specific locational use and the purpose of interacting with the soundscape of the Pepsi Pavilion installation, this makes for another historical example of the process and practice of embedded soundscaping. Indeed, Mumma's definition of "cybersonics" as "live electronic processing of acoustic sound sources" (Mumma, 2015a) represents an antecedent to my construction of "embedded soundscaping," and based on his definition, embedded soundscaping instruments could be considered a form of cybersonic instrument.

3.5 Adam Basanta, *A Room Listening to Itself* (2015)



Figure 10: *A Room Listening to Itself* at the Center for Contemporary Arts in Santa Fe, NM, 2015 (Photo by Angie Rizzo) (Source: Basanta, 2015b)

Adam Basanta (b. 1985) is a sound artist, composer, and performer of experimental music. His sound installations frequently aestheticize and musicalize audio feedback systems in relation to their enclosing acoustic environments. His sound and audiovisual installations have been presented across North America and Europe in galleries including Carol/Fletcher Gallery (London), Serralves Museum (POR), Edith-Russ-Haus für Mediakunst (GER), Vitra Design Museum (GER), Laboral (ESP), Gallery B-312 (Montréal), Foundry Darling (Montréal), Titanic Gallery (FIN), The Center for Contemporary Arts Santa Fe (USA), and BIAN 2014 Digital Art Biennale (Montréal), and have been awarded in the Prix Ars Electronica 2013 (honourable mention, Hybrid Art category) and the 2014 Edith-Russ-Haus Awards for Emerging Media Artists.” (Basanta, 2015a).

“A Room Listening to Itself,” utilizes a multitude of strikingly arranged

microphones and speakers to transform the gallery space into a giant resonator that amplifies acoustic activity and inactivity through a system of tuned microphone feedback and recursive amplification networks. Custom software constantly monitors the sounds of the room and re-calibrates the connected audio system in reaction to acoustic events to maintain a sonic equilibrium.

The result is a sonic environment which, in the artist's words, "...musicalizes and modifies the acoustic response of the room." Additionally, each microphone signal is played through several speakers, which are then amplified by neighboring microphone and speaker pairs that altogether form the recursive amplification network which amplifies the natural resonant frequencies of the room and introduces new resonances as a result of the location and distance between microphones and speakers, as well as the unique physical properties of the speaker cones. Gallery visitors modify the system's acoustic properties both through the presence of their bodies interrupting the flow of sound through the room, as well as through their intentional and unintentional sounds. Visitors' footsteps, conversations, and direct sound making into microphones allows them to become part of the feedback loop and participate in the systems of acoustic relationships embedded within the gallery space (Basanta, 2015b).

In discovering this piece by Basanta I was at first stunned by the elegance of its minimalist appearance and the beauty in its deceptively pure and simple soundscape generated from the continual flux of the feedback network's attempts to achieve equilibrium. The parallels with my

implementation of the Sonic Mirror as an interactive room installation are evident through Basanta's description of the technical implementation of his computer controlled piece, however his work features a much more complex infrastructure of multi-channel audio to compose the larger feedback network. Sonically, his work produces very different results from the Sonic Mirror system, however it's not difficult to imagine that when Basanta's piece is in the presence of gallery visitors it inspires similar forms of intentional sound making that I have observed in peoples interactions with the Sonic Mirror (e.g. speaking, yelling, and singing directly into the microphones). In reflecting on the recurring patterns of interactive dynamics that pieces like Basanta's and my own provoke from people, I can't help but surmise that humans at some innate level of natural curiosity and playful impulse are fundamentally drawn to explore the dynamics of feedback systems through interaction. I feel like this serves as another example of how systems of audio feedback can generate emergent behavior in not only physical systems (e.g. the resonant interactions of the gallery space), but also biological systems as evidenced by the provocation of directed human behavior to perturb and interact with the system through sound making. And because this piece at its core utilizes a custom designed instrument to modify and interact with the soundscape of a specific environment through the mechanism of audio feedback, again, herein lies another example of the process and practice of embedded soundscaping.

3.6 Krzysztof Cybulski, *Feedboxes* (2017)

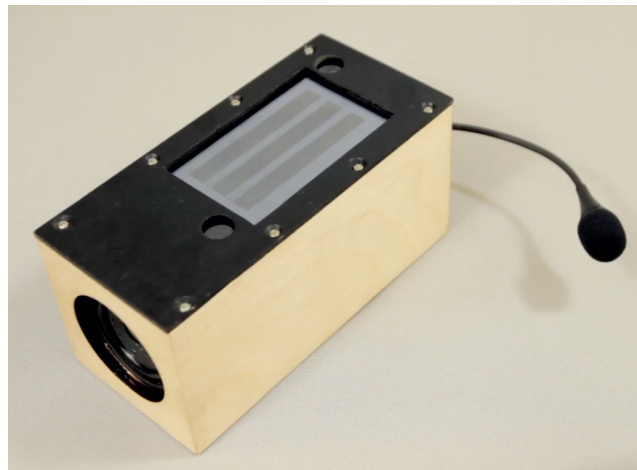


Figure 11: Feedboxes (Source: Cybulski, 2017a)

Cybulski (b. 1980) is a musician and sound artist who develops unconventional electronic instruments, musical control interfaces, and software environments. He is a co-founder of new media art collective “panGenerator,” has received an honorary mention at Prix Ars Electronica in 2017 in the category “digital musics & sound art,” has been awarded the art

scholarship from Polish Ministry of Culture in 2017, and has presented his work at such events as NIME, Warsaw Autumn Festival, Biennale Wro, and Ad Libitum Festival (Cybulski, 2017a).

I was particularly struck by the remarkable resemblance that Cybulski's Feedbox instruments share with my own hardware instrument implementation of the Sonic Mirror (section 4.2). In terms of hardware, the Feedboxes feature similar components, however it's the specific functionality of the Feedboxes that sets them apart from my own work.

Unlike my Sonic Mirror instruments, the Feedboxes use audio input to generate and trigger purely synthesized musical output in the way of rhythmic and harmonic patterns. Current Feedbox designs feature two different operating functions, a 'Rhythm box' and a 'String box.' These instruments present a “one to many” control mapping scheme wherein the volume of incoming audio signals determines multiple musical output features like the rhythmic density and pitch of samples of the output beat, or the quantity and sequencing of played 'strings' and their harmonic content of each of the different Feedbox models, respectively. Unlike a more conventional synthesizer, these instruments are autonomous in their operation and are designed to facilitate improvisatory musical interactions wherein sound operates as the control input – much in the same way that human musicians act when accompanying one another in an ensemble. Multiple Feedboxes can also be physically configured into feedback arrangements to indefinitely sustain continuously varying musical output (Cybulski, 2017b).

Similar to some of my own motivations for instrument development,

Cybulski's Feedboxes reflect a desire to create new forms of electronic instruments that foster more natural modes of music and sound making by constraining the channel of control to the raw medium of sound. Intrinsic to this is a skillful utilization of audio feedback. In this example, audio feedback is giving rise to musical output through human-machine interaction.

Additionally, because the Feedboxes are designed to react to sound as a form of control input triggering their sonic output, as instruments, they exemplify the process of embedded soundscaping since they generate sound in response to the sonic activity of their enclosing soundscape. Theoretically a multiplication of this instrument technology could be deployed within interior or exterior physical environments to create larger scale sound installations of embedded soundscaping instruments that are rendering musical output in response to the activity within their soundscape.

4. *Embedded Soundscaping* Implementation and Exhibition

4.1 Introduction

This section presents technical descriptions and exhibition documentation of the three different implementations of the Sonic Mirror instrument developed as part of this thesis. Collectively, the implementations form a series of distinct sound installations. The presentation order corresponds to the chronological order of the instrument implementation's development history: first, the hardware instrument prototypes; second, the room installation; and third, the acoustic testing system for experiments with digital materials. Ultimately, words can only go so far in describing any sonic experience, so I recommend consulting the recorded documentation of the implementations of this work (Tooby, 2017a).

4.2 Implementation 1 – Hardware Instruments



Figure 12: Sonic Mirror Hardware Instrument v2.0

The first implementation is a hardware instrument featuring the Sonic Mirror software instrument operating on a Raspberry Pi computer embedded within a wooden enclosure with a built-in microphone and speaker. This was the initial incarnation of the instrument, and the design has undergone multiple evolutions throughout the history of this project.

This implementation of the Sonic Mirror features the 'Interactive Call and Response Mode' whose functions are described in detail in section 2.3.1. The sonic output of this instrument is primarily dependent on the input from environmental sound sources it records. Frankly, it's difficult to describe in specific terms what this instrument *sounds* like, but generally speaking, it generates a transformed sonic reflection (e.g. processed audio feedback) of the sounds it records in its environment. After the instrument detects and records an external sound, the resulting output may be stretched or compressed in time, its pitch may be increased, decreased or variably modulated over time. Alternatively, a sound may be convolved with another previously recorded sound to produce a hybrid version of the two sounds wherein the spectral characteristics of one are imparted on the other. And various permutations of these types of effects are also possible.

At another level, the resulting processed output from the instrument's speaker will often feedback into its microphone and be re-processed, further compounding the automated effects processing and causing the rapid buildup of complex sonic textures from an initial source sound.

4.2.1 Technical Overview

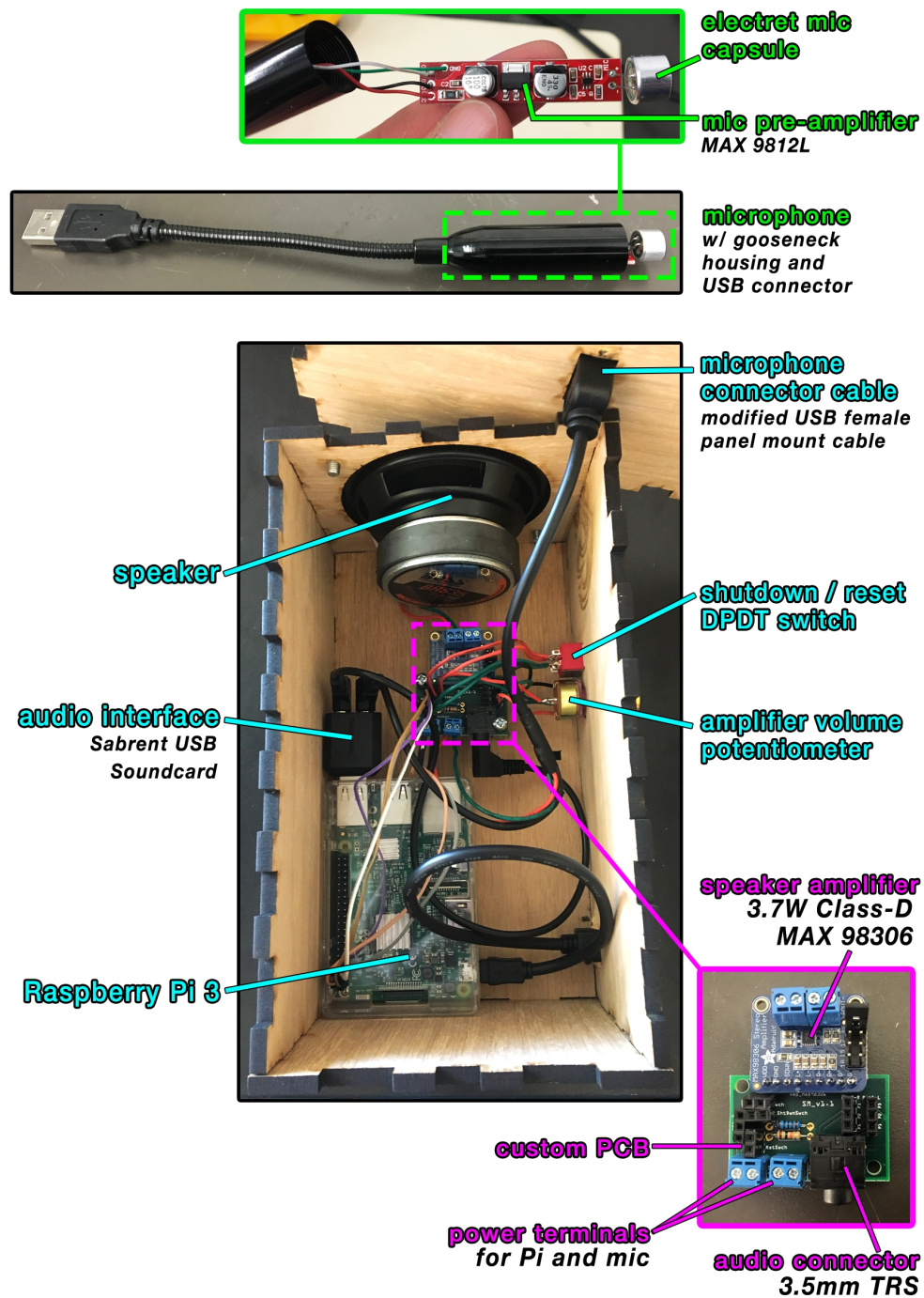


Figure 13: Components of Sonic Mirror Hardware Instrument v2.0

The primary components of the Sonic Mirror hardware instrument are a Raspberry Pi computer (v3), a USB digital audio interface (Sabrent sound card), an electret condenser microphone, a speaker (3.5in. / 50W / 4 Ω), an electret mic pre-amplifier circuit (MAX9812L), a speaker amplifier (3.7W Class-D MAX98306), and a 5V 2.5A power supply.

The hardware enclosure is made of laser cut and engraved plywood featuring different audio visualizations derived from an audio recording of the spoken words, “Sonic Mirror.” Each visualization is engraved on the side and top faces of the enclosure and include a time-domain waveform, a 2D spectrogram, and a 3D spectrogram.

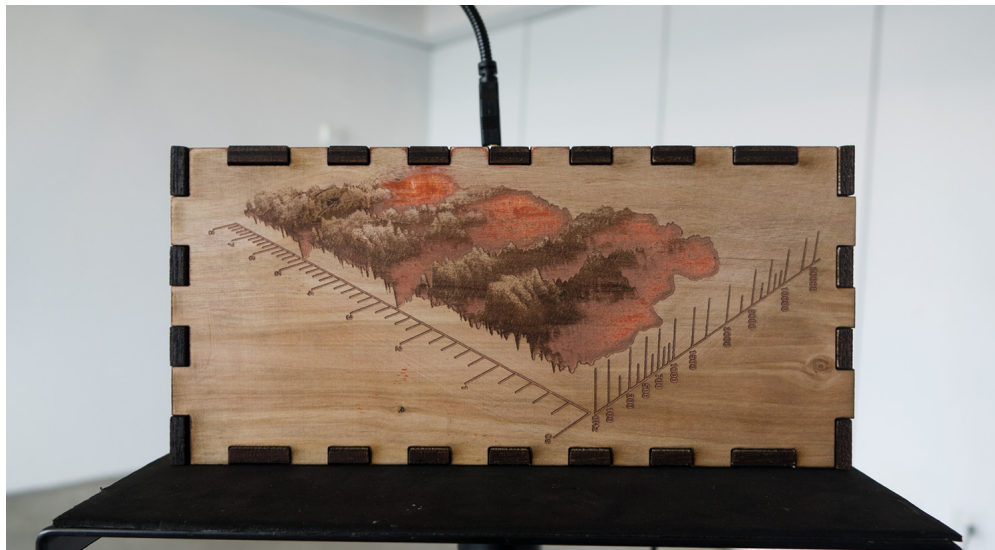


Figure 14: Laser engraving of 3D spectrogram on side face of wooden Sonic Mirror enclosure



Figure 15: Laser engraving of 2D spectrogram on top face of wooden Sonic Mirror enclosure

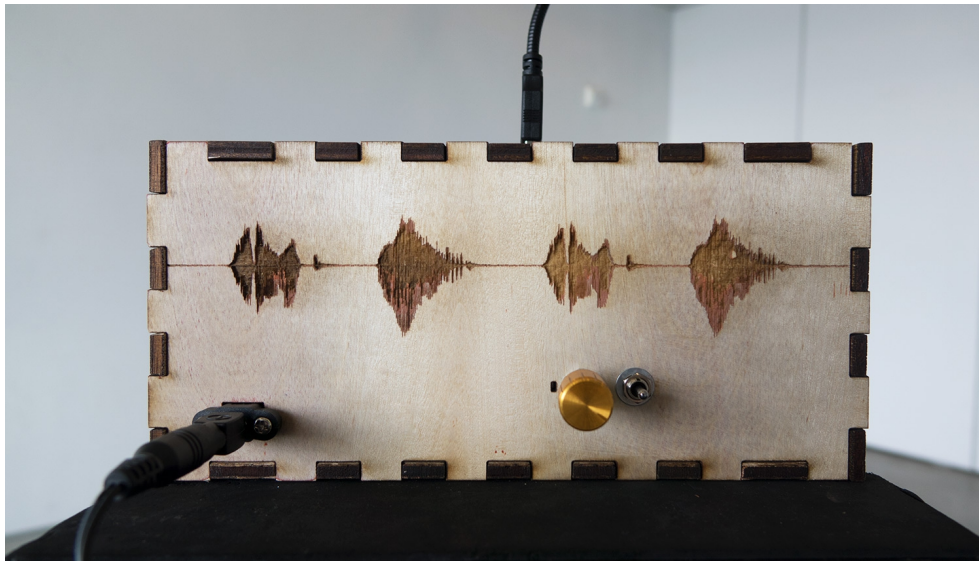


Figure 16: Laser engraving of audio waveforms on side face of wooden Sonic Mirror enclosure

4.2.2 Exhibition Documentation

The development of the Sonic Mirror hardware instrument began in April 2016. Its design has undergone a series of iterations over the course of frequent exhibitions. Technical constraints of venues and exhibition deadlines prompted many of the design decisions as well as observational feedback of audience interactions with the instrument.

As of this writing, multiple generations of Sonic Mirror hardware instrument prototypes, in increasing quantities, have been shown in six exhibitions spanning May 2016 to April 2017. These exhibitions have all been great forms of 'field tests' for these instruments, and by my estimates have facilitated between at least 500 - 800+ unique interactions between people and the instruments over the course of a total of approximately 114 hours of continuous exhibition time.

The exhibition venues willing to display this work included more traditional fine art exhibitions, digital art and new media exhibitions, and experimental electronic music concerts. I think this is indicative of the flexible nature of the medium of interactive sound art, and also its liminal condition in artistic fields today.

Across most of these exhibitions I noticed the Sonic Mirror hardware instruments had a tendency to evoke certain recurring behaviors in viewers who interacted with them. Viewers speaking and singing into the instrument's microphone was by far the most common form of interaction, followed by whistling, and the playback of songs and audio from smart phones. But what I find of interest was how these instrument's reliably provoked displays of sound making in people that wouldn't otherwise occur in the specific contexts

of a gallery or exhibition venue.

The intrinsic audio feedback process of these instruments was sometimes conducive of elaborate and borderline obscene vocal displays in people of all ages. For many people I observed during the course of these exhibitions, it was the experience of hearing their own voice transformed dynamically in real-time by the instrument that seemed to foster a desire to explore their innate vocal abilities in new and creative ways. I don't wish to proclaim that these occurrences indicate any deep insight about human nature, but to me the significance in these observations was the realization of how the designed behavior of an autonomous interactive audio system can reliably cause people to emit specific patterns of sounds and behaviors they wouldn't normally in the absence of the system, and also to take more notice of not just their own sounds but sounds from their surrounding environment as well, if only for the brief moment of exposure to the instrument. At a larger level though, what I find most significant is that all of these observed behaviors in people were ultimately instigated by the basic mechanism of audio feedback.

***The Cosmos* – Santa Cruz Museum of Art and History (May 2016):**

The first generation Sonic Mirror hardware instrument prototype was shown at the *The Cosmos* exhibition at the Santa Cruz Museum of Art and History in May 2016. Its hardware design was considerably more of a proof-of-concept at these early stages of development, and was comprised of a consumer USB microphone, a Raspberry Pi v2, and similar basic audio

circuitry and speaker hardware outlined above in the technical overview.



Figure 17: Sonic Mirror v1.0 and viewers at the *Cosmos* exhibition, Santa Cruz Museum of Art and History, May 2016

The Sonic Mirror was the only featured piece of sound art in the exhibit, which featured art by local Santa Cruz artists as well as research displays by UCSC astronomers. The exhibit was described as featuring:

“...interdisciplinary artists and scientists whose work exists within the delicate nexus that connects our understanding of the Cosmos with our ability to creatively engage with it, thereby pushing the bounds of the way we conceive of, and express, our place within the universe.” (Wallace, 2016)

It was here where I first glimpsed the sheer novelty and entertainment value that this form of instrument could engender through a continuous series of interactions with many different people of all ages in a very public setting.

At this point in its development, the Sonic Mirror's software functionality was quite limited in comparison to today's version, but its core functionality of basic delayed audio feedback and pitch modulation was present, which sufficiently captured the attention of viewers. Out of nearly all the other exhibitions, this one continues to stand out in my memory for the particularly uninhibited vocal interactions the instrument encouraged from visitors at the gallery. Multiple individuals each spent extensive 10 - 20 minute periods interacting with the instrument with their voices. One individual even made a serious offer to purchase the instrument outright. The intensity and enthusiasm of people's responses to this instrument helped encourage its continued development and inform the design of many new functionalities for subsequent exhibitions (Tooby, 2016b).

MFA Now – Root Division Gallery (March 2017):

The MFA Now exhibition occurred at San Francisco's Root Division gallery throughout the month of March 2017. It featured works spanning the mediums of painting, photography, digital print, video, sculpture, and new media by 25 artists from university MFA programs around the greater California Bay Area (Root Division, 2017). It was here where the newest v2.0 design of the hardware instrument was displayed in public for the first time, and also the first occasion featuring two instances of the instrument.



Figure 18: Two Sonic Mirrors installed at the MFA Now Exhibition, Root Division Gallery, San Francisco, March 2017

This was by far the most traditional fine art gallery context where the Sonic Mirror hardware instruments have been shown. As a result, I noticed some viewers had a tendency to regard the work more as pure sculptural display or concept piece than an actual functional sound sculpture or electronic instrument, which seemed to hinder the amount of interactions at times. But fortunately, even during the most crowded and noisy period of the reception, many others appeared to comprehend the sound-making function of the piece, most likely indicated by the built-in microphone and speaker hardware, and engaged with it in close proximity. To me this reaffirmed the instrument's abilities to engage people's interactions despite high levels of ambient noise, and also highlights its adaptive flexibility to varying sonic environments. The instruments are able to operate in both quiet and noisier settings, it simply changes the dynamics and nature of the piece.

4th Annual LAST (Life Art Science Technology) Festival – San Jose State University (April 2017):

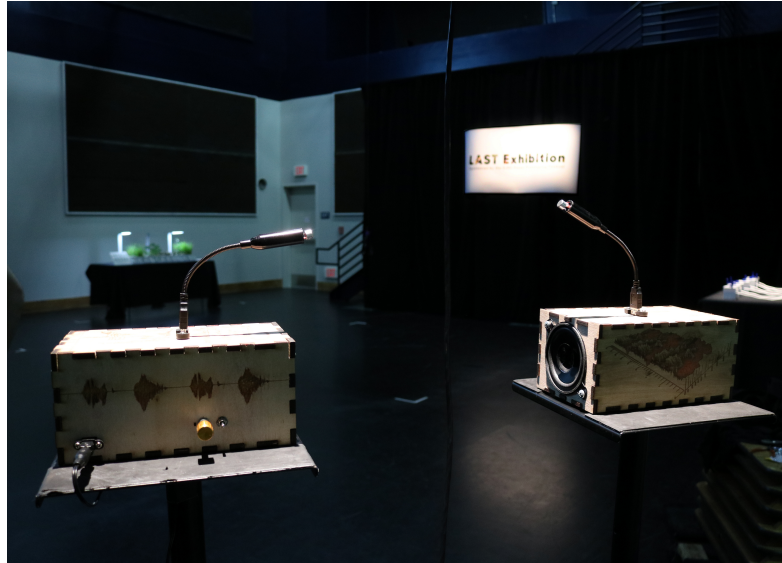


Figure 19: Sonic Mirrors installed at the 4th Annual LAST Festival in San Jose, April 2017

The L.A.S.T. Festival is an interdisciplinary arts festival showcasing work at the intersection of art, science, and technology. In the words of the festival organizers, its mission is to “help reshape the cultural environment of the 21st century towards a multidimensional form of individual and social creativity.” (LAST Festival, 2017).

This was the first digital art and new media festival where I showed the Sonic Mirror hardware instruments, and it struck me as a significant change in venue in comparison to the preceding MFA Now exhibition. Given the predominantly technological and interactive nature of the festival, the hardware instruments experienced their first dose of especially rough physical handling from audience interactions. No amount of “do not touch” disclaimers

would have been able to stop the frequent and repeated instances I observed of children and adults alike manipulating the instrument's volume and power controls, and in some extreme cases, scratching and twisting their delicate microphone capsules. Miraculously, no serious damage occurred to the instruments, but out of the large quantity of physical interactions that occurred, I repeatedly noticed the curious behavior of people waving their hands in front of the instruments microphones as if they expected there to be some visual or infrared sensing component to the piece. I think this behavior may have been instigated by the instrument's specific installation configuration. The two Sonic Mirrors were positioned on speaker stands directly facing each other with about 18 inches of distance between them. To me these observations signify one of the many common challenges that interactive sound art contends with in public art festival contexts – the challenge of how to effectively guide the interactions of an unexpected audience purely with sound. I don't claim to have worked out an effective solution yet, but this anecdotal reaffirmation of the visual dominance of our society impressed upon me the importance and difficulty of strategizing an interactive sound installation's presentation to best communicate its functionality to a general public audience. Yet regardless of the intensive physical handling, people generally comprehended the instrument's sonic functionality and engaged in extensive aural interactions throughout the festival's duration. In this particular instance though, the instrument's physical hardware components seemed to be the cause initiating many interactions.

***California Electronic Music Exchange Concert (CEMEC) – CCRMA,
Stanford University and UC Santa Cruz (April 2017):***

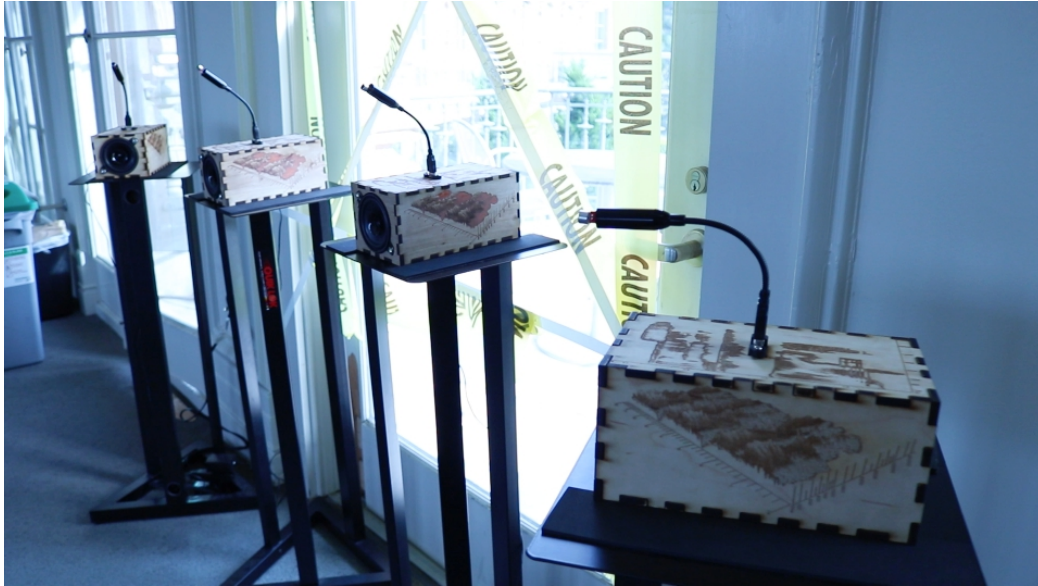


Figure 20: Four Sonic Mirrors installed at the Stanford CEMEC concert, April 2017

The California Electronic Music Exchange Concerts is a series of events designed to foster connections and community between graduate computer and electronic music programs in California. The 2017 series of CEMEC included programs from the universities of UCSD, UCSB, UCSC, Mills, CCRMA/Stanford, and CalArts. Composers and musicians whose work has been selected for inclusion in the CEMEC series are encouraged to travel and present their music in concerts at different institutions from their own, and in the process gain exposure to the larger statewide community of academic computer and electronic musicians.

I presented the Sonic Mirror hardware instruments at the CCRMA (Stanford) and UCSC legs of the concert series as a sound installation book-

ending each concert. I installed four hardware instruments at the CCRMA concert and five at the UCSC concert, making each of these events 'firsts' in terms of the sheer quantity of instruments I publicly displayed at any one time. I also debuted new software functionality at these events, making them great opportunities to field test at the eve of the Digital Arts and New Media MFA show, *ultraSHIFT*, detailed below.

Notably, these were the first electronic music concert venues I had presented the Sonic Mirrors at, and naturally I experienced a significant change in atmosphere and audience temperament in comparison to the various art shows and festivals I had exhibited this work at previously. Seeing as how these were both concerts featuring experimental new electronic music, the attendance was quite low, but consisted almost exclusively of musicians and makers more knowledgeable about the technical mechanisms underlying my instruments than the average museumgoer I had encountered at the previous exhibitions. For these reasons, these concerts made for very engaging and informational feedback sessions to observe the kinds of interactions that the Sonic Mirrors could evoke from electronic musicians. The CCRMA concert was particularly noteworthy since there are an abundance of percussion instruments regularly stored around the building. A couple viewers played cymbals, gongs, and shakers in the presence of the instruments. This diversity of timbral sources produced refreshingly new responses and sounds from the instruments, and for me, beckoned towards whole new avenues of musical and compositional applications for Sonic Mirror instruments. The small audience sizes also provided the most intimate

and quiet installation experience I've had to date, which allowed viewers to listen and observe the instrument's processes and soundings at a more detailed and subtle level than usual. Altogether, these concerts instilled in me the value of further exploring the exhibition and adaptation of these instruments for musical listening and performance contexts.



Figure 21: Viewer interactions with the Sonic Mirrors, Stanford CEMEC concert, April 2017



Figure 22: Sonic Mirrors installed at the UCSC CEMEC concert, April 2017

***ultraSHIFT* – Digital Arts and New Media MFA Exhibition (April 2017):**



Figure 23: Sonic Mirrors at *ultraSHIFT*, Digital Arts and New Media MFA Show, UC Santa Cruz, April 2017

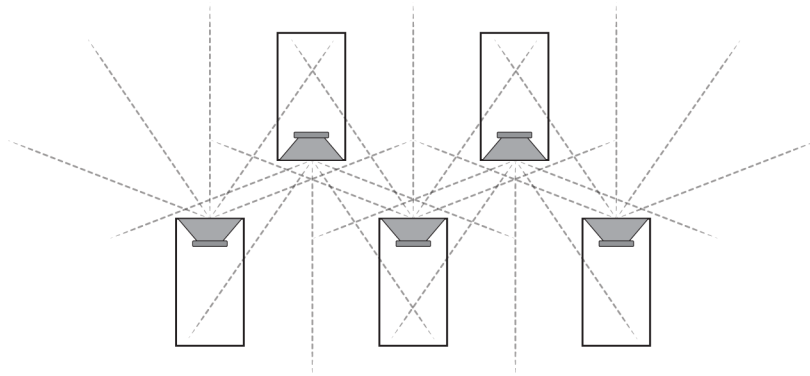


Figure 24: Sonic Mirror Hardware Instrument Installation Configuration, *ultraSHIFT*, Digital Arts and New Media MFA Show, UC Santa Cruz, April 2017

ultraSHIFT was the Digital Arts and New Media 2017 MFA exhibition, and it featured a culmination of the Sonic Mirror's development to date. Five hardware instruments were configured as a sound installation within a large hallway of the Digital Arts Research Center at UC Santa Cruz. The instruments were positioned in two rows facing each other allowing them to readily feedback amongst themselves and become a form of larger self-sustaining feedback network. If one out of the five instruments recorded and played back a sound, it would set its neighbors off in the same process and create a larger chain reaction of calls and responses throughout the network of instruments. The audio feedback occurring between the instruments would often permit a sound to be continuously transformed and heard throughout the collective for considerable lengths of time after its initial recording.



Figure 25: Children interacting with the Sonic Mirror hardware instrument installation at the *ultraSHIFT* exhibition

An observation from the *ultraSHIFT* exhibition I feel is particularly noteworthy was the occurrence of very distinctive harmonic drones generated by the instruments during periods of relative silence. Intermittently during the exhibition I would hear the collective of instruments humming in harmony with themselves and the ambience of the hallway's ventilation system. I suspect this is primarily the result of the pitch modulated output from the Sonic Mirror's live input buffer, and perhaps also the residual low-level audio feedback that accumulates over time due to the physical coupling of the instrument's speaker and microphone to its wooden enclosure. While I haven't conducted thorough tests yet, I believe the vibrations of the speaker cone cause low frequency sound waves (e.g. frequencies less than approximately 120 Hz) to travel through the wooden enclosure directly to the adjacent gooseneck microphone housing and enclosed capsule, which is re-recorded, re-played, and results in the accumulation of low frequency feed back over time. Essentially, I was hearing emergent tones from the self-resonating enclosure and sound system of the instrument. Effectively, the emergent voice of the Sonic Mirror hardware instrument.

Upon reflection, I believe this activity may have been occurring to lesser degrees throughout the instrument's development phases, however for some reason, the effects of the instrument's self-resonating properties were most apparent at the *ultraSHIFT* exhibition, and seemed to have a tendency to converge on harmonic intervals of the ambient drones of the installation hallway's ventilation system – rendering a soothing, whistling, droning choir-like effect during periods of relative acoustic tranquility.

4.3 Implementation 2 – Room Installation



Figure 26: Visitors listening to the Sonic Mirror Room Installation at the *ultraSHIFT* exhibition, April 2017

The second implementation presented at the *ultraSHIFT* exhibition featured the Sonic Mirror as an immersive sound installation at the scale of a large white-box gallery room. A microphone suspended in the center of the room recorded the room's ambience along with sounds made by humans interacting with the piece. The mic was connected to a computer operating the Sonic Mirror instrument software, and the processed audio output was sent to a quadrophonic surround sound system. The result was an interactive acoustic feedback network that modulated sounds and its audio processing behavior in response to the sounds heard in the room.

The installation alternated between the two operational modes detailed in section 2.3 (“Interactive Call and Response Mode” and “Recursive Acoustic Convolution Mode”).

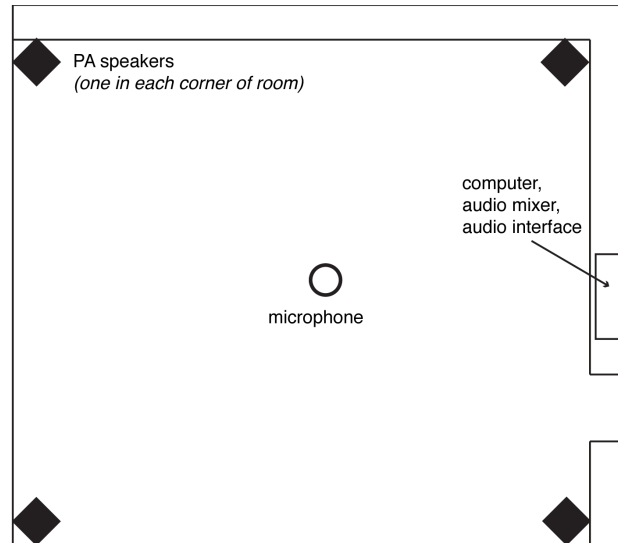


Figure 27: Sound system configuration of the Sonic Mirror Room Installation at the *ultraSHIFT* exhibition (overview)

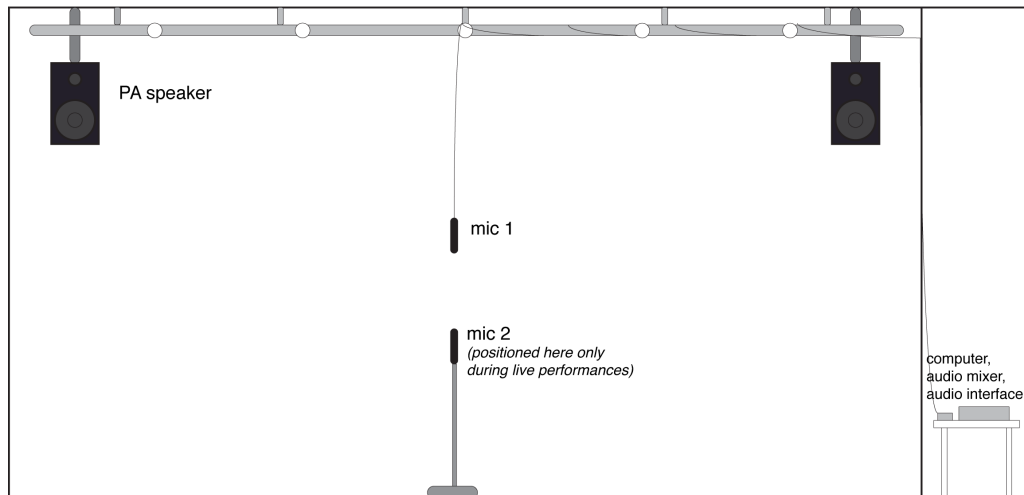


Figure 28: Sound system configuration of the Sonic Mirror Room Installation at the *ultraSHIFT* exhibition (side view)



Figure 29: Interactions with the Sonic Mirror room installation at the *ultraSHIFT* exhibition

4.3.1 Interactive Call and Response Mode (in a room)

For a majority of the installation's operation, the Interactive Call and Response Mode was engaged. Effectively, this is the same operational mode featured within the hardware instruments (Implementation 1) at the increased scale of a room.

The sonic quality of this piece is dependent on the audio input recorded by the instrument system, and for this reason is fairly open-ended. For example, if a participant speaks in the room, the system will subsequently play back the recorded speech with any one of the possible audio processing effects outlined in section 2.3.1. The speech recording's playback rate and/or pitch (independent of playback rate) may be variably modulated over time, resulting in a subtly or dramatically transposed version of the original sound. When modulated by the system, a participant's speech sounds may seemingly be transformed into an extremely low-pitched bellowing groan, or

conversely, a higher-pitched vibrato-laden “chipmunk” voice, and of course a broad spectrum of more subtle pitch transpositions between the programmed extremes.

Convolution (cross-synthesis) is another possible effect that the system frequently applies to recorded sounds. Continuing with the example of human speech as a sound source, the system may convolve a recording of a person's speech with a previously made recording of someone clapping, which could render a hybrid sound with spectral characteristics of both the speech and clapping sounds. The sonic result of human speech convolved with broadband percussive impulses like claps often creates a form of audio delay effect such that the speech sample is sounded in synchrony with the rhythm of the claps, potentially creating the illusion of a cacophonous, densely layered chorus of speakers.

Furthermore, because the physical configuration of the microphone and PA system in the room permits acoustic feedback, the effects of the Sonic Mirror system can rapidly compound to create lush, complex, and dense sonic textures from a single initial sound source being continuously processed and re-processed. With each cycle of feedback, the system may apply a drastically different audio effect, which can quickly lead to a dramatic and alien transformation of the original sound as a result of the recursive nature of the re-recording process through the acoustics the room, in combination with the successively varying audio effects processing.

When filled with groups of people, the installation was correspondingly active with a large variety of dynamic sonic activity derived from the sounds it

recorded (and re-recorded via feedback). In the absence of people, the room's soundscape was relatively placid, however, because of the Sonic Mirror instrument's design, the room would periodically emit processed versions of previously recorded sounds. This function acted as a means of indicating the presence of the installation to passing viewers, and to signal its interactive functionality. The periodic playback of recorded audio samples also served to create a type of temporal sonic distillation of all the recent activity that had occurred in the space. If the instrument system did not detect a sound over the duration of a minute or more, it would either play a re-processed version of a previously recorded sound or alternatively play through its entire sample library in a matter of seconds to render a basic form of sound scene audification. Even though the room may have been visibly absent of people, ghostly echos of preceding sound events could be heard as the system reflected sounds from its memory. Additionally, while the volume levels of the sound system were adjusted to avoid runaway self-feedback, occasionally the system would re-record a replayed sound during these periods of relative silence that was loud enough to exceed the thresholds of sound detection, and as a result of this acoustic feedback, the instrument system could listen to the sound of its own "voice," re-processes these reflected sounds, and gradually evolve its sonic memory over time.

During the exhibition of this piece I routinely encountered people interacting with it in a variety of ways, but predominantly with their voices. Speaking (and yelling) were the most common forms of interaction, but I was pleased to observe more uninhibited displays of singing, whistling, and bodily

percussion as well. For many, the piece provided a great deal of entertainment. The experience of hearing one's own voice filtered and transposed in time and pitch was a humorous and accessible experience for many, even a baby, who spent the better part of 15-20 minutes continuously laughing and yelling at the responses of the installation to their own voice. Alternatively, others gravitated towards more contemplative observation, and simply sat on the room's floor to listen to the effects of the piece's process unfold over extended periods of time. To my surprise, people repeatedly projected the distinctive personality characteristics of “mocking” and “sarcastic” onto the system. For them, the system's inherent processes of sonic mimicry seemed to indicate its concerted intent to “make fun” of their voices. When audiences reached a critical mass of three or more people, the installation would at times become a beacon of attraction at the exhibition as the cacophony of people's vocalizations and the system's output would lure outsiders into the room out of curiosity.

4.3.2 Recursive Acoustic Convolution (in a room)

Approximately once each hour during the *ultraSHIFT* exhibition I held a short performance demonstrating the system's Recursive Acoustic Convolution mode using a variety of acoustic instruments as sound sources including a melodica, wind chimes, singing bowl, and my voice. This mode of the installation was a slight variant of the operational mode featured in Implementation 3 detailed below. In this implementation, the Sonic Mirror system recursively convolved an arbitrary sound with the acoustics of the room.

In more descriptive terms, this performance process consisted of an initial sound (e.g. a melodica chord or the sound of my speaking voice explaining the process) that would then be automatically recorded by the Sonic Mirror instrument and subsequently played back through the room's PA system. Upon playback, the resulting sound would be simultaneously re-recorded, and this step would be repeated about 11 to 15 times resulting in a total performance duration of about 12 to 15 minutes. Also, with each playback repetition the recording made during the preceding cycle would be played at a reduced playback rate but with the original pitch preserved, creating a basic form of audio time stretching effect.

This process results in an initial sound being repeatedly played through the acoustic space of the room and increasingly stretched in duration, such that with each playback repetition, the resonant frequencies of the room and audio system are increasingly excited. After several repetitions of this process, the original sound is seemingly transformed into a lush, harmonically dense, sustaining, droning texture. During the later stages of the piece, listeners experience an immersive “wall-of-sound” effect as the room seems to sing in response to the initial source sound that has been successively filtered by the acoustics of the room and stretched in time. The sonic results are similar to those achieved in Alvin Lucier's piece, *I Am Sitting in a Room*, because the process is fundamentally identical, however, I found that the introduction of digital audio time stretching with each playback repetition helps accelerate the apparent onset of the process' effects. A more detailed explanation about the influence of Lucier's piece on this implementation of my

work is described in section 3.3.

In contrast to the interactive mode, this mode facilitated deeper levels of listening engagement for many. Listeners commonly reported experiences of serenity and relaxation after experiencing the piece. I should add that the exhibition space of this piece featured a large window with an excellent view of the Monterey Bay, which likely helped induce positive experiences in many listeners. But in consideration of the fact that the sound of this piece is intrinsically shaped by the acoustic environment as a result of the recursive convolution process, I think the integration of both soundscape and landscape in the moments of time occupied by the performance piece created a balanced unity that people seemed to sense in various ways. One listener claimed their experience of the piece was “...surprisingly profound [and] seemed to indicate a presence beyond the sound...” that they didn't anticipate. Admittedly, this kind of anecdotal feedback is highly subjective, but nevertheless it intrigues me to witness the effects that this form of audio feedback can evoke in listeners.

Alternatively, others described the performance experience fostered sensations of curiosity and the desire to better understand the mechanics behind the sonic process at hand. I did attempt to deliver an accessible explanation of the process before each performance, but even for those knowledgeable of music and audio, to conceptualize a technical process in the abstract and experience it aurally are two very different things, so it's understandable how for some this piece could instill a sense of wonder and curiosity about the nature of sound and its interactions within space and time.

4.4 Implementation 3 - Digital Material Resonance Experiment

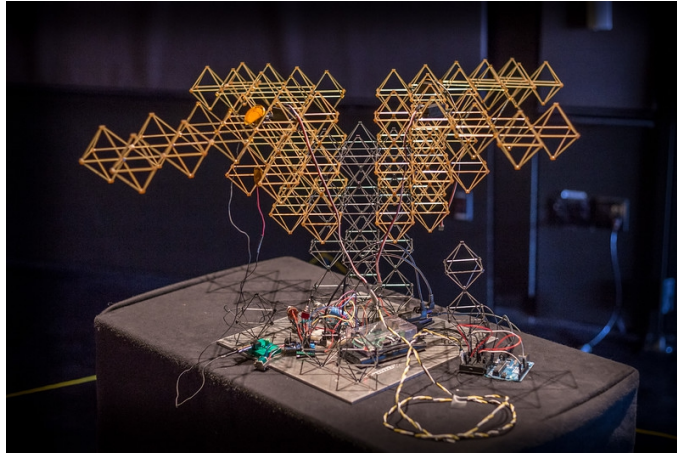


Figure 30: Sonic Mirror Digital Material Resonance Experiment at the *ultraSHIFT* Digital Arts and New Media MFA 2017 exhibition

The third implementation showcased a version of the Sonic Mirror instrument adapted for performing acoustic experiments in a physical digital material structure of an experimental robot made by UCSC's DANSER Lab.

In essence, this implementation is functionally identical to the Recursive Acoustic Convolution performance mode of “Implementation 2” described above, but it utilizes two piezoelectric transducers to play and record sounds through the robot's digital material structure instead of a microphone and speaker. Also, instead of using an arbitrary sound source as the convolution impulse, synthesized impulse tones including white noise and a sawtooth wave sweep were used to maximally excite the resonant frequencies of the robot's structure.

Visually, the piece featured a quadrupedal robot comprised of a lattice network of digital materials. Two piezoelectric disc transducers were coupled with the robot's structure via 3D-printed mounts to aid in the transmission of

sound. The connected audio circuitry was fully exposed and placed beneath the robot on a metal plate alongside a connected Raspberry Pi operating the Sonic Mirror instrument software. One of the robot's ligaments would periodically rotate to demonstrate its movement abilities, which was actuated by an embedded linear servo controlled by an Arduino microcontroller.

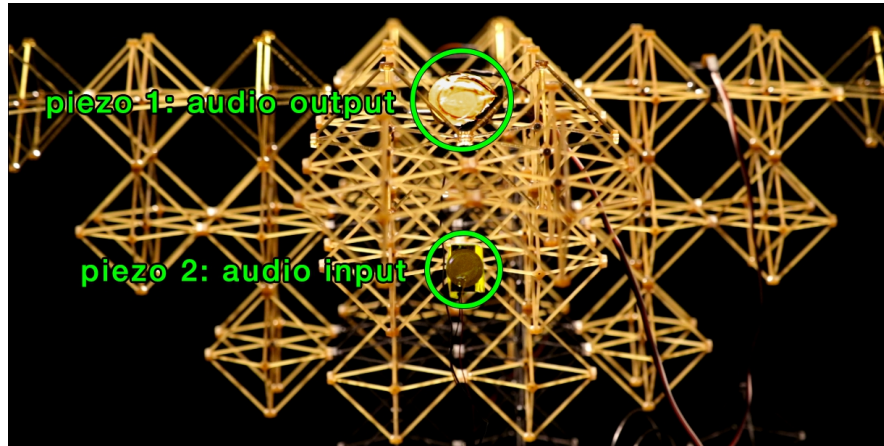


Figure 31: Embedded piezo disc transducers were used to transmit and record sound through the robot's digital material structure

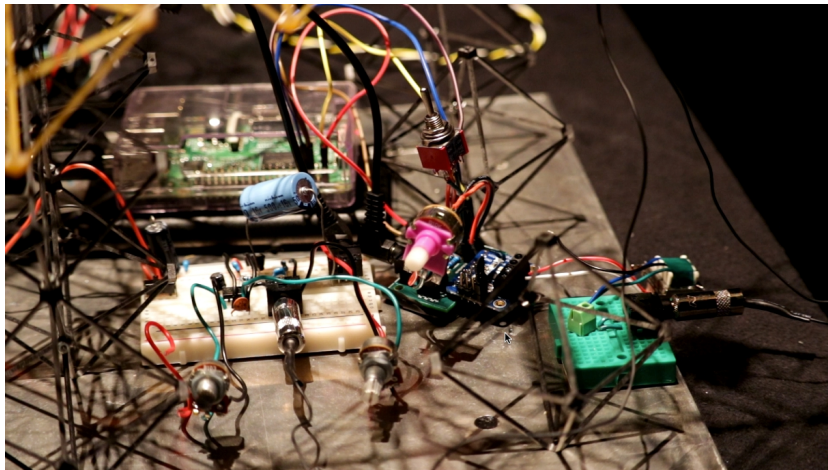


Figure 32: Raspberry Pi and connected audio circuitry used to operate the automated Digital Material Resonance Experiment installation

Viewers could listen with headphones to hear the sonic process of Recursive Acoustic Convolution (RAC) looping inside the robot's structure. At the start of each sequence, a synthesized impulse sound (either five seconds of sustained white noise or a sawtooth wave sweep from 100Hz - 20kHz) is played through the robot's structure, recorded, and the resulting recording played back through the structure while being re-recorded, and so-forth, until this process reveals the resonant frequency of the robot's structure. With each playback repetition the resulting recording is played back at a slightly reduced playback rate (e.g. 70%) with corresponding upwards pitch-shifting to preserve the original pitch of the recorded sound. This effectively produced a basic form of time stretching of the sampled audio, and with each repetition the duration of the resulting sound increased, having the apparent result of accentuating the filtration effect caused by the robot's digital material structure, and allowing listeners more time to hear the effects of the process in action. The observed resonant frequency of the structure in this specific configuration was a high pitch, roughly centered around 1800 Hz (e.g. A6). The duration of an entire cycle of the RAC process with a given initial source sound (either white noise or a sawtooth wave sweep) was limited to last no more than about five minutes.

Listening to the process unfold would reveal the gradual transformation of what could otherwise be considered an initially harsh synthesized tone into a purer, whistling, almost sinusoidal tone centered around the resonant frequency of the system. The subjective sonic quality of the end result was of an almost ethereal tonal quality, sounding like a soft, droning, resonant

whistle. I personally find the sounds produced by this piece to be vaguely reminiscent of steam or liquid flowing through industrial utility pipes.

Common remarks by viewers and listeners revolved around the piece's contemplative nature. In contrast to the other implementations described above, having to experience the sounds of this piece with headphones facilitates a more intimate and individual listening experience. Throughout the exhibition I noticed that for some listeners, this piece evoked a great deal of surprise. For them, the realization that a sound could be occurring within a solid structure but not be audible externally through the air revealed a new dimension to the physics of sound that they had not previously considered. Aside from its educational value, other viewers commented on how the unique sounds being generated within the structure combined with the visual complexity of the robot allowed them to “get lost in the piece,” and experience feelings of tranquility and curiosity such that this was something they “could get lost in for hours.” By far, this was the least interactive, but perhaps one of the more contemplative and subtle implementations of the Sonic Mirror instrument on display at the *ultraSHIFT* exhibition.

More exhaustive descriptions about the technical process, background, and outcomes of the experimental research underlying this implementation of the Sonic Mirror are detailed at length in the following chapter.

5. Art-Science Research: Digital Material Acoustics

5.1 Introduction

This chapter documents the interdisciplinary art-science research collaboration that evolved over the course of this thesis project, which entails the application of the Sonic Mirror instrument system for engineering research. Specifically, the background context of the development and nature of the research, the ensuing experimental process, results, and discussion are described in detail.

5.1.1 Background

During my studies at UC Santa Cruz, I became acquainted with engineering faculty Mircea Teodorescu and the Dynamics, Autonomous Navigation, Surface Engineering, and Robotics (DANSER) Lab. After familiarizing myself with the DANSER Lab's research projects, it soon became apparent that the Sonic Mirror instrument system could be adapted to assist with their ongoing studies of *digital cellular solids*, a new material technology developed by researchers at NASA's Coded Structures Lab, which has potential applications for aerospace engineering.

5.1.2 Digital Materials and Digital Cellular Solids

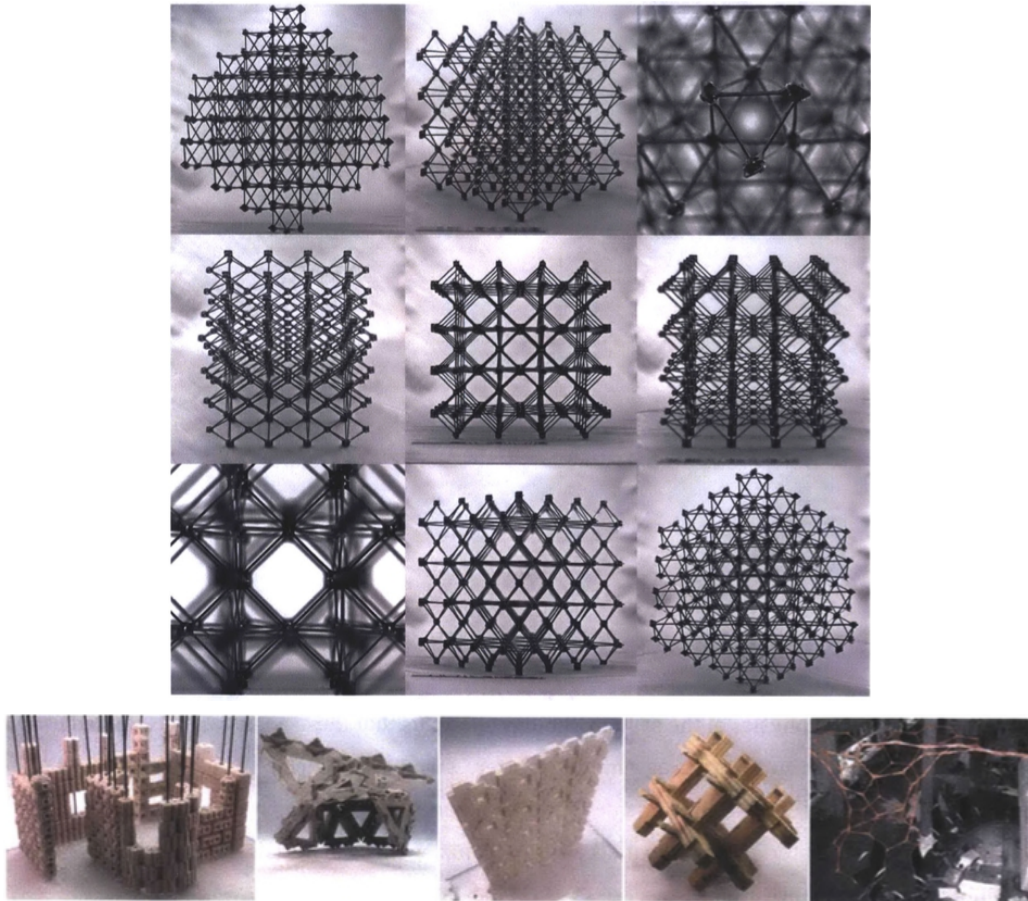


Figure 33: Examples of digital materials (Source: Cheung, 2012, p.18, 59)

Digital materials are comprised of a few types of discrete, compatible, and reconfigurable parts, which assemble to form larger functional constructions. Inspired by biology, digital materials represent a method of programming matter by encoding structural and functional information in the morphology of small sets of building blocks.

Digital cellular solids are a new ultra-light digital material technology developed by scientists from the Coded Structures Lab at NASA's Ames Research Center. Relative to their size and density, when implemented with

fiber composites and arranged into lattice geometries, digital cellular solids are stronger than any previously known ultra-light material.

This technology presents a new scalable workflow for applying fiber composites to engineering problems, and it has many potential applications for robotics and aerospace engineering – including flexible airplane wings and space station construction (Cheung, 2012; Jenett et al, 2016; Jenett et al, 2017).

5.2 Digital Material Resonance Experiments

5.2.1 Background

At the time of this writing, engineers at UC Santa Cruz's DANSER Lab are actively researching the structural properties and applications of digital cellular solids (DCS) by developing experimental robots comprised of DCS in the form of octahedral lattices, specifically referred to as “volumetric pixels” or “voxels” (Jenett, Cellucci, Gregg & Cheung, 2016).

The voxels are manufactured using a conventional injection molding process, which allows them to be made of a variety of materials and exhibit varying structural properties (e.g. stiffness, strength, density, etc.). Three different types of voxels are used by the DANSER Lab for their robotics research: 30% Carbon Reinforced REI (carbon fiber composite), Ultem 2200 (20% glass filled), and Ultem 1000 (unfilled). Each voxel consists of 12 three inch elastic beams joined at six corners, and each corner is designed to interlock with an adjacent voxel's corner using a geometrical interlocking technique secured by a screw (Cramer et al, 2017).

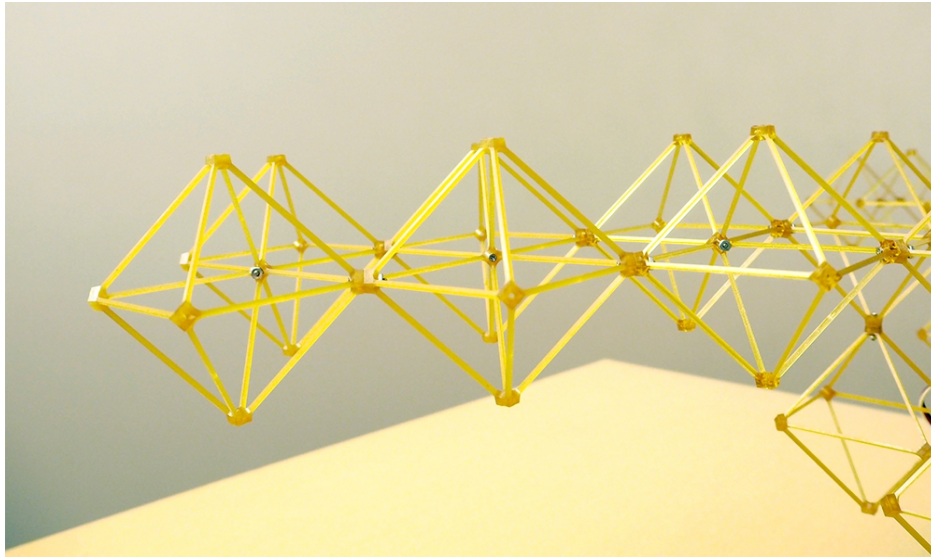


Figure 34: Digital cellular solid voxels used in DANSER Lab research

As of this writing, the newest DCS robot under development by the DANSER Lab is quadrupedal robot. Lab researchers discovered that the forced expansion and contraction of physical displacement between specific adjacent voxel joint nodes within the robot's structure can cause larger-scale net rotation of individual ligaments. Using embedded linear actuator servos, the robot's individual ligaments can be precisely rotated in sequence to create a new form of biomorphic locomotion, allowing the robot to rotate in place and walk. As of this writing, the precise physics causing the complex rotational motion of the quadrupedal robot's ligaments is not yet fully understood, and this is one of the current research foci of the DANSER Lab.

In the course of my exposure to this research, it became evident to me that I may be able to adapt the Sonic Mirror instrument system I was developing to assist with acoustic testing of DCS materials. Primarily, I was interested in recording, audifying, or devising some acoustic measure to help

observe and correlate the physical causes contributing to these structures' complex motions.

While witnessing the development of the DANSER Lab's quadrupedal robot, I was actively developing the Recursive Acoustic Convolution (RAC) functionality (see section 2.3.2) of the Sonic Mirror instrument software. After repeated RAC experiments in a room, the idea emerged to apply the same process in the context of the DANSER Lab DCS research. It was apparent that this process was a viable means of exciting and detecting the resonant frequencies of an acoustic structure, which happened to be a physical feature of DCS that DANSER Lab researchers were in the process of verifying experimentally. Furthermore, we were also interested in examining how the movement and configuration of a digital material robot's structure would effect the structure's resonant frequencies.

At a practical level, developing an automated means of acoustically exciting and detecting the resonant frequencies of any arbitrary structure is a task I felt compelled to adapt the Sonic Mirror system to perform, especially when considering it already had the hardware and software infrastructure in place to perform such an experiment. The desire to implement such functionality was mainly inspired by a knowledge of Alvin Lucier's well-known piece, *I Am Sitting in a Room* (1969). A more detailed discussion of this piece's influence and relation to the broader scope of this thesis research is detailed in section 3.3.

In regards to the engineering research, being able to develop an effective means of analyzing the resonant frequencies of a digital material

structure seemed to present a viable path towards assessing changes in its structural health over time. After realizing the functional implications of Lucier's piece and the RAC process in general, I proceeded to adapt the Sonic Mirror instrument system to perform the RAC process in an automated way to assist with the DANSER Lab's digital material research. In theory, if we could devise a reliable means of measuring the resonant frequency of digital material structures, we could correlate these measurements with conditions of a structure's configuration and health over time.

5.2.2 Experimental Setup

In order to conduct the RAC process experiment and analyze the resonant frequency of digital material structures, two piezoelectric disc transducers are used to record and output audio within a connected structure. Custom designed 3D-printed mounting connectors created by DANSER engineers are used so the piezoelectric transducers can be temporarily mounted and positioned within digital material structures. The mounting connectors also function as an effective waveguide, allowing sound vibrations to pass from digital material structures to the piezoelectric transducers, and vice versa.

Two amplifier circuits are used for the audio input and output stages of the recording and playback system. The specific circuits used (an LM386-based amp for input and a 3.7 Watt Class-D amp for output, see Appendix C for more details) were chosen to better match the impedance of the piezoelectric transducers and maximize the sound pressure level and signal

to noise ratio of the audio input and output.

A computer operating the acoustic testing software program is connected to the audio input and output circuits via a digital audio interface (i.e. an ADC and DAC). The acoustic testing program is an automation of the RAC process operating as a sequenced sub-routine within the larger Sonic Mirror software instrument. Section 2.3.2 describes the RAC process in more detail, but a basic outline of the RAC procedure adapted for these DCS experiments is as follows:

1. Several seconds of synthesized white noise is played into the DCS structure as an impulse source and simultaneously recorded
2. The resulting recording of white noise being played through the DCS structure is played through the DCS structure and re-recorded
3. Step 2 is repeated indefinitely in a recursive fashion such that with each repetition the recording made in the preceding step is played back through the DCS structure.

The structure acts as a filter which, over time, increasingly accentuates the resonant frequencies contained in the original white noise impulse while attenuating all other frequencies. After multiple iterations of this process, the resonant frequencies of the DCS structure are revealed.

So far, this experimental process has been conducted within two different DCS robot designs by the DANSER Lab. The first, a biomorphic “Worm” design, and the second, the quadrupedal robot.

In the Worm robot, two different structural configurations were tested

with the same piezo placement. In 'Configuration 1,' the robot contained only unbroken voxels. In 'Configuration 2,' the robot contained one broken voxel.

In the quadrupedal robot. The two piezo transducers used for audio recording and playback were positioned on one of the robot's ligaments in relative proximity with about 6 inches of vertical displacement from one another.

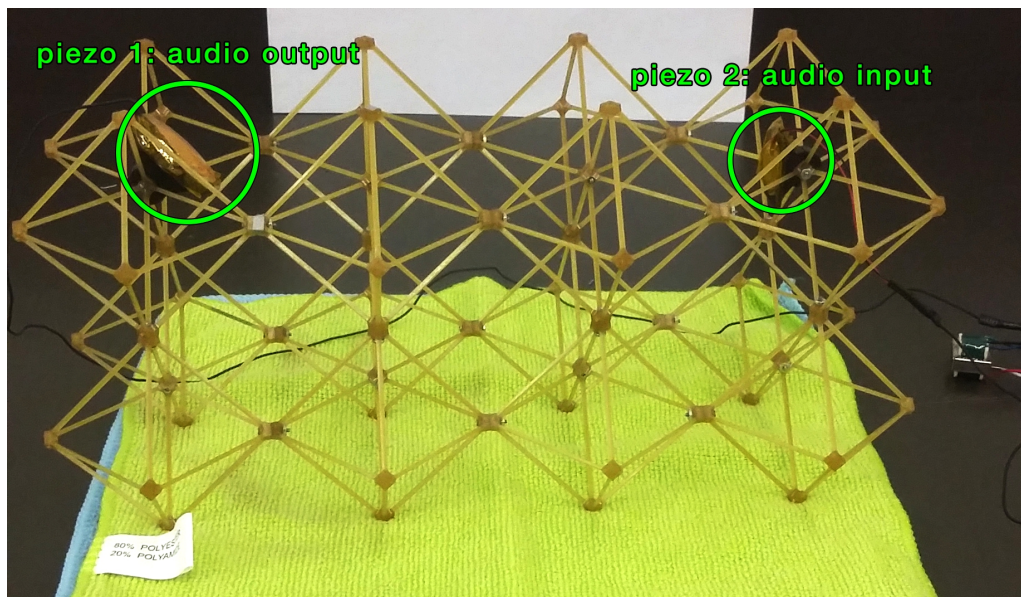


Figure 35: Worm Robot Position 1: Configuration 1 & 2, Unbroken & Broken Voxels

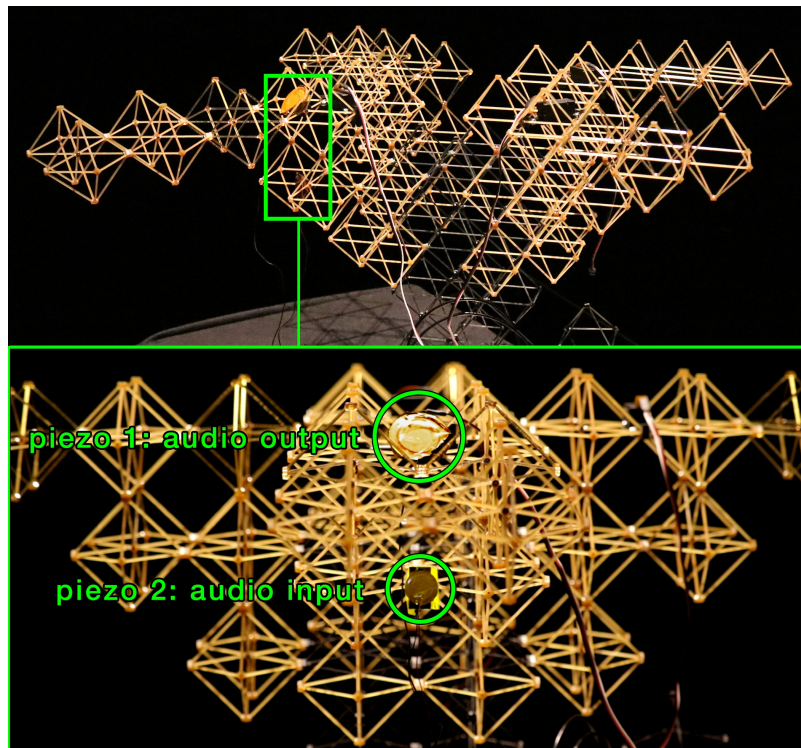


Figure 36: Experimental setup of RAC testing within quadrupedal robot

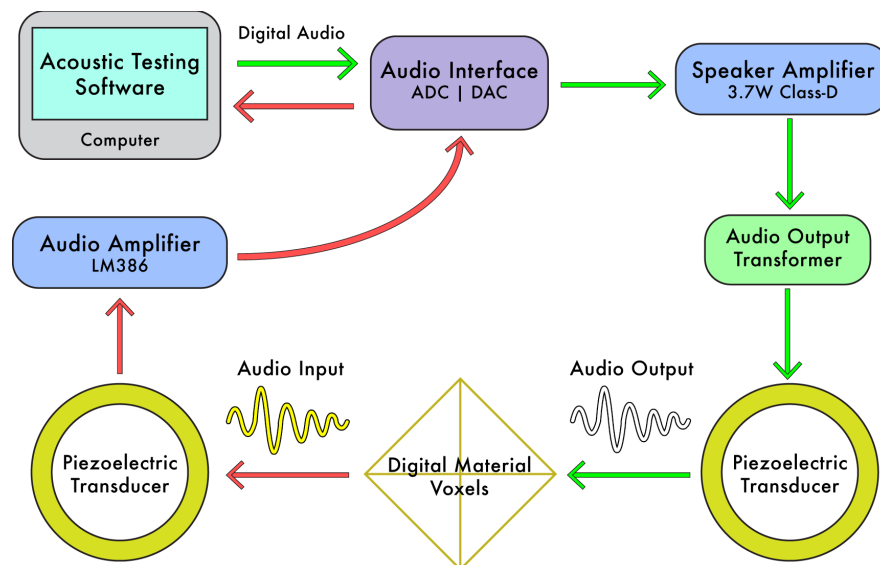
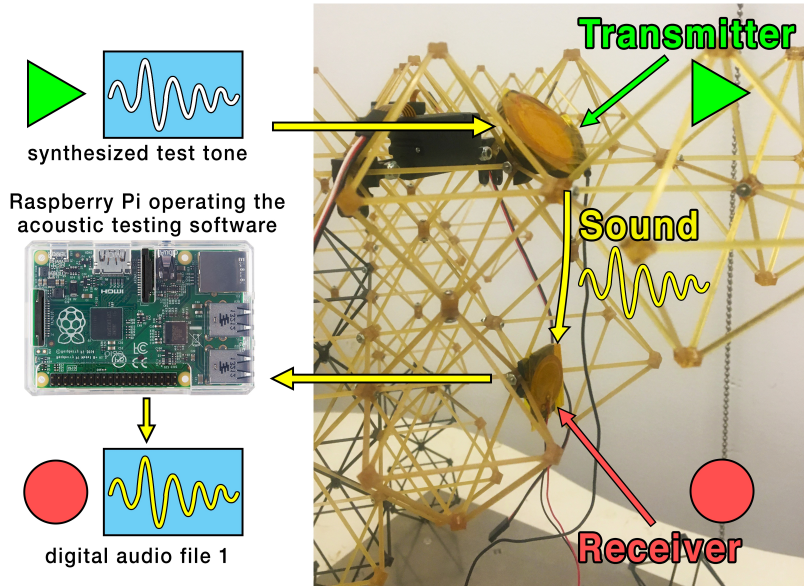


Figure 37: Audio hardware configuration for RAC experiment within DCS structures

Step 1:



Step 2, 3, etc...

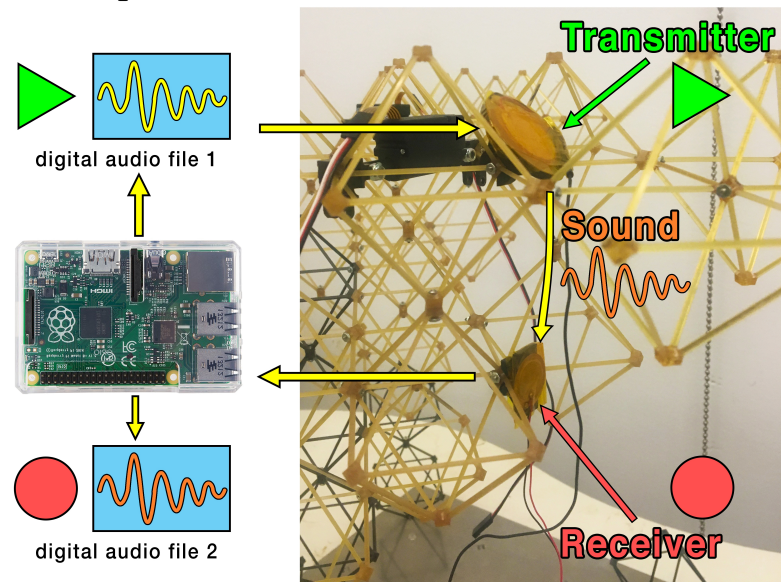


Figure 38: Recursive acoustic convolution process within the DANSER Lab quadrupedal robot's digital material structure

5.2.3 Results and Discussion

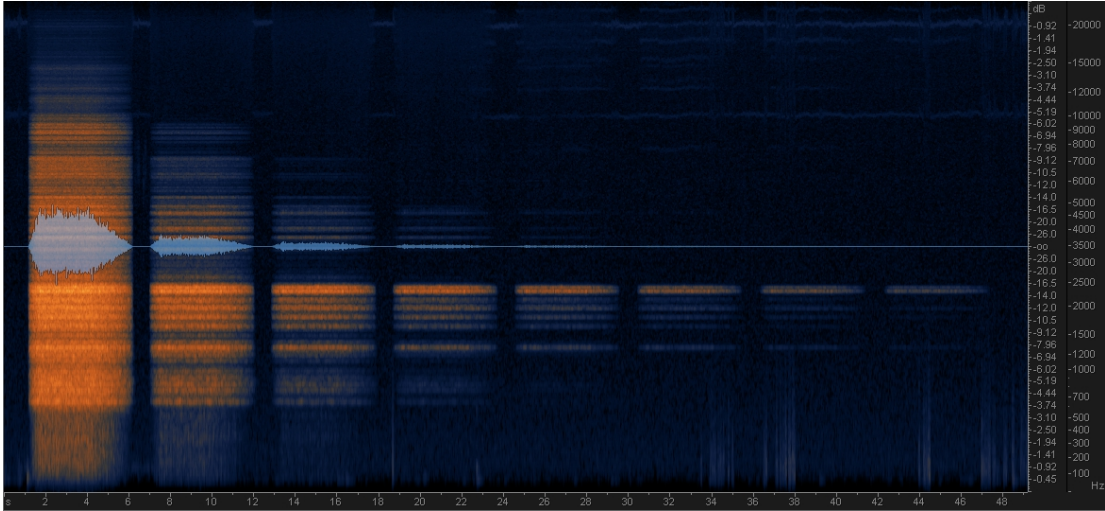


Figure 39: Spectrogram of 8 repetitions of the RAC process within DANSER's Worm Robot – Configuration 1 – Unbroken Voxels

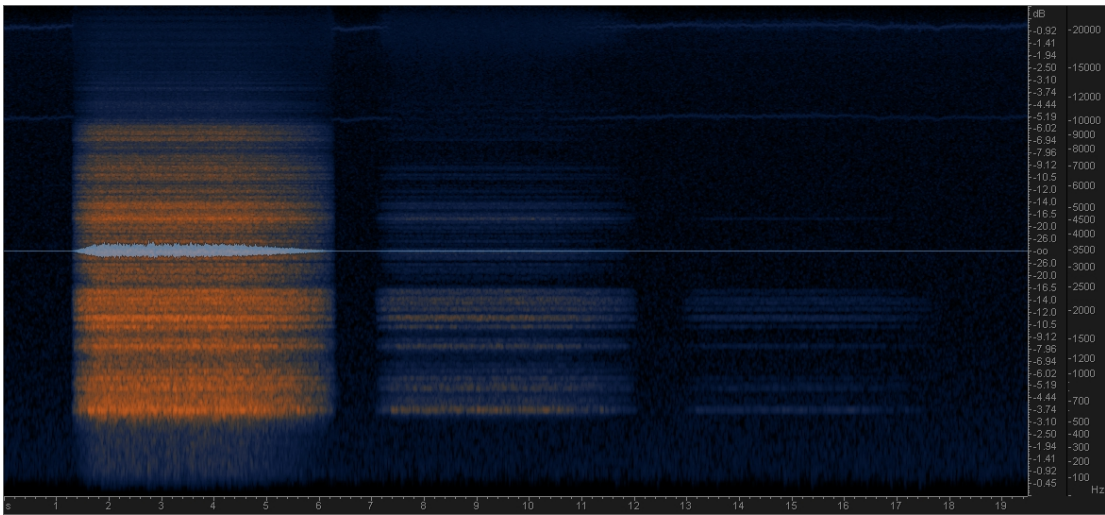


Figure 40: Spectrogram of 3 repetitions of the RAC process within DANSER's Worm Robot – Configuration 2 – Broken Voxel

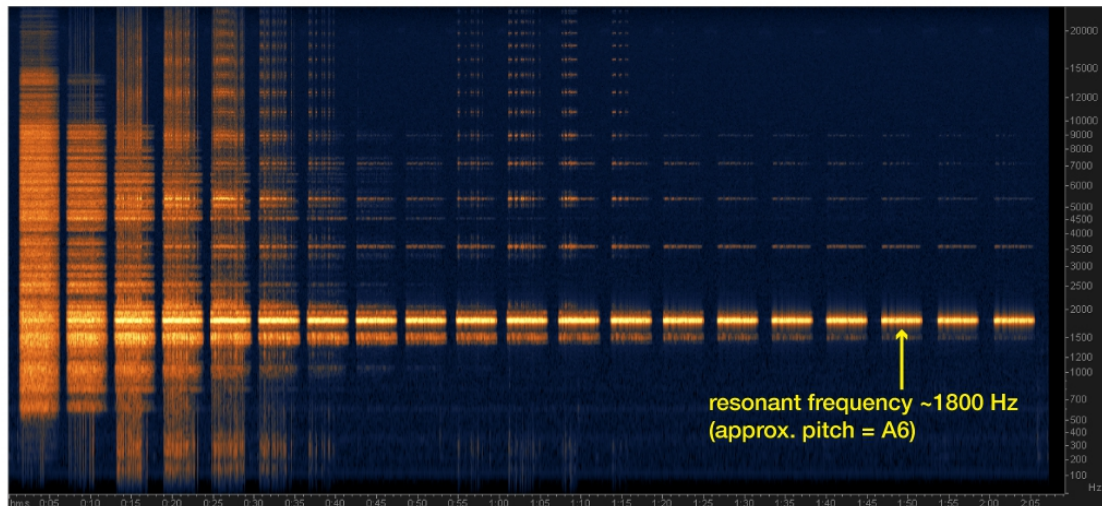


Figure 41: Spectrogram of 20 repetitions of the RAC process within DANSER's quadrupedal voxel robot. Initial sound source: five seconds of sustained white noise

Initial results are encouraging and show that the RAC process may prove to be a viable means of measuring the resonant frequency of DCS structures.

Based on the results of the Worm robot experiments, it's evident that the structural integrity of the DCS voxels has a significant impact on the structure's resonant frequency and ability to propagate sound. In the configuration with only unbroken voxels, the initial white noise impulse sound persists through eight repetitions of the RAC process before signal strength drops below the noise floor, and a defined resonant frequency of approximately 2400 Hz is apparent. Whereas in the configuration with broken voxels, the initial white noise impulse sound persists for only three repetitions of the RAC process before signal strength drops below the noise floor, and no clearly defined resonant frequency is apparent.

In the quadrupedal robot experiment, a clearly defined resonant frequency of approximately 1800 Hz is observed and persists for over 20 repetitions of the RAC process, unabated.

It should be noted that the initial results achieved are derived from a limited amount of experimentation, and much more thorough experimentation needs to be conducted to formulate a more coherent understanding of the precise physics underlying these results. It should also be noted that the resulting measured resonant frequencies are a combined result of the unique characteristics of both the DCS structure and acoustic recording and playback systems (e.g. the frequency response characteristics of the piezo transducers).

But in the meanwhile, even from these limited results, it's evident that the structural configuration and health of digital materials plays a critical role in shaping the resulting measured resonant frequencies. These results are encouraging in the sense that clearly defined patterns in the observed resonant frequency (or the lack thereof) can be detected from a given DCS structural configuration and positioning of piezo transducers. With this basic acoustic testing methodology established, we can now begin the process of conducting more thorough experimentation. After further experimentation and the acquisition of larger amounts of acoustic data, we can begin to extrapolate a more nuanced understanding of the correlation between patterns of observed resonant frequencies of DCS structures and their corresponding configuration and structural health.

5.2.4 Conclusions and Future Research

So far, our research has produced an audio transducer system suited for conducting acoustic studies of digital cellular solids (DCS). Additionally, the Recursive Acoustic Convolution (RAC) process presents one possible means of analyzing the resonant frequencies of DCS structures, however, due to the successive nature of the process, it can take time on the order of a few to tens of seconds to perform an analysis test, which makes it not incredibly time efficient. Conducting more conventional forms of impulse response testing using shorter, more ideal acoustic impulses (e.g. a very short duration broadband pulse) may produce a measurable frequency response of DCS structures that reveals their resonant frequencies in a much more time efficient and accurate fashion. Nevertheless, the same audio transducer system we have devised could conduct these tests. It's worth mentioning that the total combined cost of the audio and computing hardware used to conduct these experiments is very cost effective, totaling less than \$100. The results we have achieved with our testing process and simple piezo transducers present a much clearer and detailed representation of the resonant frequencies of DCS materials in comparison to other attempted measurements performed with more expensive state-of-the-art laser interferometers. By leveraging the low cost of piezoelectric transducers, we could deploy larger arrays of recorders and transmitters to conduct more comprehensive acoustic tests and multi-channel recordings of larger DCS structures. Ultimately, our research has revealed a viable path forwards for future acoustic testing and structural health monitoring of digital materials.

6. Conclusion

Over the course of the past year, I have conducted a variety of research and project development while accomplishing the following:

1. Developed a new software instrument that can operate as an autonomous interactive digital audio sound system for applications in sound art, music, and general acoustic experimentation in different environments
2. Created three unique implementations of the Sonic Mirror instrument:
 - i. Hardware instrument design (5 units and counting)
 - ii. Installation for rooms and other large acoustic spaces
 - iii. Piezo transducer interface for digital materials (and other solid substrates)
3. Adapted the Sonic Mirror instrument system for interdisciplinary research with the DANSER Lab to create an automated acoustic testing system for digital material structures. Our research has produced specialized hardware and a methodology for performing acoustic experiments with digital material structures. It has also revealed avenues for future research in this area.
4. Shown the Sonic Mirror implementations at six different exhibitions, totaling 114+ hours of exhibition time. The amount of unique interactions obtained between people and the Sonic Mirror implementations has exceeded at least 500 - 800+ encounters and produced extensive audio and video documentation and observational feedback of peoples interactions with the systems.

At the inception of this project, I considered it to be dominantly focused on researching applications of embeddable interactive audio systems in creative and scientific contexts. But along the way I have observed recurring patterns which have led me to realize much of my work is asking more fundamental questions about sound through studies of different manifestations of audio feedback and the kinds of emergent phenomenon it can provoke in physical and biological systems. After my collective experience developing and exhibiting these new interactive audio instruments, as well as through a process of artistic and scientific research, I have arrived at the concept of “embedded soundscaping” in an effort to succinctly describe my process and practice, and contextualize it within a larger system of relations to similar works of sound art and music.

In reflecting on my work in relation to other composers and sound artists investigating aspects of audio feedback, certain threads of insight have become clearer in regards to the questions proposed at the outset of this thesis.

Specific forms of audio feedback can indeed reveal features of physical environments and also provoke responses from animals, inclusive of humans. Processes of recursive acoustic convolution have been demonstrated to reveal the resonant frequencies in physical structures. The iconic example of this is Alvin Lucier's *I Am Sitting in a Room*, but I have also augmented and replicated this process to assist with the digital materials research in conjunction with the DANSER Lab.

David Dunn's musical and scientific work has illustrated how basic forms

of environmental audio feedback (e.g. successive recording and playback of field recordings) can manipulate animal behavior to varying degrees. In the case of his original work with the Sonic Mirror, it is suggested that the basic process of recording sounds from an environment and playing them back into said environment can create a vortex of attraction for local wildlife. In this particular case, blue jays and other birds. However, his bark beetle research has demonstrated in a much more dramatic fashion how the audio playback of bark beetle's own bioacoustic signals can provoke them to kill each other and stop reproducing, thus presenting a potential means of purely acoustic pest management. Dunn's original Sonic Mirror instrument also exemplifies the process of embedded soundscaping through the design of an instrument for the express purpose of interacting with the soundscape of an environment.

In a much less dramatic and more humble form of creative investigation, through the exhibition of my Sonic Mirror instruments, I have observed how designed systems of interactive audio feedback can reliably elicit specific reactions in people. Aside from common forms of vocalization and singing, some people had a tendency to project a form agency onto the instrument, assuming that it was intentionally attempting to mock them through its patterns of audio transformation and mimicry.

Similarly, in witnessing peoples responses to other operational modes of the Sonic Mirror, I've observed how different forms of acoustic feedback can evoke other recurring emotional responses. In the case of the Sonic Mirror's implementation of Lucier's (and originally Bose's) process of recursive acoustic convolution, a spectrum of emotional responses ranging from

scientific curiosity to relaxation and quiet contemplation have been repeatedly reported to me by viewers of this piece.

To me, these observations suggest that people share some innate captivation with systems of audio feedback, perhaps motivated by instincts of natural curiosity or play. To varying degrees, this is reflected in the works of Mumma, Basanta, and Cybulski insofar as they're providing a mechanism or environment for people to readily interact with systems of audio feedback.

And upon deeper consideration of these works, my own, and indeed all systems of audio feedback, it's evident that the sound generated by a total system of audio feedback is tightly coupled with the relationships between the specific audio system facilitating the audio feedback, external physical objects and other possible sound sources within a larger enclosing soundscape. Basanta's *A Room Listening to Itself* illustrates this idea perfectly in the sense that everything from the unique shape and size of the speaker cones, the positioning of all the microphones and speakers in the room, potential sounds and physical movements of museum visitors, and the acoustics of the space all sum to create the resulting sound generated by the audio feedback system. With this in mind, using our natural auditory perception, we (and other animals) can at a certain level hear patterns of relationships between all of the distinct constituents of an audio feedback system. And because the sound of feedback encodes such a rich amount of information about physical relationships in our environment (albeit information that is difficult to unpack) perhaps this reveals a clue as to why humans and other animals are seemingly innately attracted to the sounds created by systems of audio

feedback.

And lastly, to broach the final question posed by this thesis, technologies utilized for creative experimentation with audio feedback can in certain circumstances translate to scientific research and applications. This is evidenced by the chronology of Dunn's history of composing pieces of environmental audio feedback and interaction with natural systems, subsequent beetle field recording, and the ensuing application of bark beetle audio playback to disrupt beetle's reproductive habits.

In my own work I've experienced the transfer of the Sonic Mirror instrument from initially artistic applications to scientific research with the DANSER Lab with some tentatively encouraging results. Conversely, Lucier's *I Am Sitting in a Room* reflects how ideas from scientific fields can propagate into artistic fields. And of course, the elaborate production of the E.A.T. collective's Pepsi Pavilion is an iconic representation of the synergistic outcomes that can result from idea and skill transfer between artists and scientists. Significantly, a recurring element between all of these works are various implementations of audio feedback systems.

Just like the behaviors of feedback systems, creative inspiration and influence between fields can be non-linear and circuitous. Artists can sometimes act as agents of idea transfer across fields to promote new scientific developments, or at least inspire new lines of thinking in the pursuit of creative problem solving.

With all of these ideas in mind, the recently formulated concept of embedded soundscaping has become more reified. Equipped with this new

way of describing the process and practice of designing instruments to intentionally interact with environmental soundscapes, I plan to continue developing the Sonic Mirror instrument system to expand the range of environments in which it can be deployed. In parallel, I will open-source all of the hardware documentation I have accrued so that others can build and experiment with their own embedded soundscaping systems. Ultimately, my hope is that others will use such technology to help deepen their awareness and engagement with the sounds in the world around them.

Appendix A: Sonic Mirror Program Code

File – “00 Main.scd”:

```
//Sonic Mirror – 00_Main:

//execute below lines one at a time and then "INITIALIZE" block last

~dir = "/Users/Scott/Dropbox/DocSync/Project Docs/DigitalArt/SonicMirror/sonic-mirror_SC_git/dev/";
//input file path to SM code folder here

(~dir ++ "01_Startup.scd").load;    // Server Options and Startup
(~dir ++ "02_SynthDefs.scd").load; // SynthDefs
(~dir ++ "03_Busses.scd").load;    // Busses
(~dir ++ "04_Functions.scd").load; // Functions and Routines
(~dir ++ "05_InitSM.scd").load;    // Main SM Initialization Code
~init_SonicMirror();               //Launch
```

File – “01 Startup.scd”:

```
//== Server Options and Hardware I/O Setup ==

//ServerOptions.devices; //uncomment and evaluate this to see what audio input/output devices are
//available (if you want to customize your audio I/O configuration)
(
o = Server.local.options;
o.sampleRate = 48000;
o.blockSize = 256; //SC default
//s.latency = 0.05; //0.2 is SC default

o.memSize = 16384;

o.inDevice = "Built-in Microph";
o.outDevice = "Built-in Output";

//o.inDevice = nil; //use system default audio input (change value to manually configure input)
//o.outDevice = nil; //use system default audio output (change value to manually configure output)

s.boot;
);
```

File – “02 SynthDefs.scd”:

```
//Sonic Mirror - 02_SynthDefs:

(
//Record audio input to buffer using audio-rate input Phasor to control position in buffer
SynthDef(bufRec, { //<---NEW -- Added Lo-cut
    lphaseBus, buf, sendBus, loCut = 90, loCutQ = 1, /*peak = 120, peakQ = 1, peakGain = 0,*/
    mul = 1|
    //var sig = SoundIn.ar(bus: 0, mul: mul);
    var sig = BHiPass4.ar(SoundIn.ar(bus: 0, mul: 1), loCut, rq: loCutQ, mul: mul); //pass SoundIn
through HiPass
    //sig = BPeakEQ.ar(sig, freq: peak, rq: peakQ, db: peakGain); //<----consider
    BufWr.ar(sig, buf, InFeedback.ar(phaseBus) );
    //OffsetOut.ar(sendBus, sig); //would this provide better sample accurate scope plotting and
recording?
    Out.ar(sendBus, sig);
}).add;

SynthDef.new(\bufRead_complex1, {
```

```

    lbuf, duration, amp=1, attack = 0.02, release = 0.02, rate = 1, pitch = 1, winSize = 0.1,
timeDisp = 0.003, pShiftMix = 0, out = 0l
    var sig1, sig2, sigOut, frames, dur, atRelDur, env, phasor;
    frames = BufFrames.kr(buf);
    dur = duration.asFloat; //total playback duration in seconds
    atRelDur = attack.asFloat + release.asFloat;
    env = Env.linen(attackTime: attack, sustainTime: (dur - atRelDur), releaseTime: release, level:
1);
    //env = Env.linen(attackTime: 0.003, sustainTime: (dur - 0.006), releaseTime: 0.003, level:
1); //for fast rate playback
    phasor = Phasor.ar(0, BufRateScale.kr(buf) * rate, 0, frames - 1);
    sig1 = BufRd.ar(1, buf, phasor, interpolation: 4); //changed -- interp 4 for faster playback
rates
    //sig1 = BufRd.ar(1, buf, phasor);
    sig2 = PitchShift.ar(sig1, windowSize: winSize, pitchRatio: pitch, pitchDispersion: 0,
timeDispersion: timeDisp);
    //sig2 = PitchShift.ar(sig1, windowSize: 0.1, pitchRatio: pitch, pitchDispersion: 0,
timeDispersion: 0.003);
    //sig2 = PitchShift.ar(sig1, windowSize: 0.35, pitchRatio: pitch, pitchDispersion: 0,
timeDispersion: 0);
    sigOut = SelectX.ar(pShiftMix, [sig1, sig2]) * EnvGen.ar(env, doneAction: 2);
    Out.ar(out, sigOut * amp);
} ).add;

SynthDef.new(\bufRead_noise1, {
    //arg buf, duration, amp=1, freq = 1, out = 0;
    arg buf, duration, amp=1, modFreq = 0.25, loFreq = 1, hiFreq = 3, out = 0;
    var dur, frames, ptr, env, lfo, sig;
    dur = duration.asFloat; //total playback duration in seconds
    frames = BufFrames.kr(buf);
    lfo = SinOsc.ar(modFreq, 3pi/2).range(loFreq, hiFreq);
    ptr = LFDNoise1.ar(lfo).range(0, frames - 1);
    //ptr = LFDNoise1.ar(freq).range(0, frames - 1);
    env = Env.linen(attackTime: 0.02, sustainTime: (dur - 0.04), releaseTime: 0.02, level: 1);
    sig = BufRd.ar(1, buf, ptr) * EnvGen.ar(env, doneAction: 2);
    Out.ar(out, sig * amp);
} ).add;

SynthDef.new(\bufRead_noise3, {
    //arg buf, duration, amp=1, freq = 1, out = 0;
    arg buf, duration, amp=1, modFreq = 0.25, loFreq = 1, hiFreq = 3, out = 0;
    var dur, frames, ptr, env, lfo, sig;
    dur = duration.asFloat; //total playback duration in seconds
    frames = BufFrames.kr(buf);
    lfo = SinOsc.ar(modFreq, 3pi/2).range(loFreq, hiFreq);
    ptr = LFDNoise3.ar(lfo).range(0, frames - 1);
    //ptr = LFDNoise1.ar(freq).range(0, frames - 1);
    env = Env.linen(attackTime: 0.02, sustainTime: (dur - 0.04), releaseTime: 0.02, level: 1);
    sig = BufRd.ar(1, buf, ptr) * EnvGen.ar(env, doneAction: 2);
    Out.ar(out, sig * amp);
} ).add;

SynthDef.new(\bufRead_sineSineMod, {
    arg buf, duration, amp=1, lfoFreq = 0.25, loFreq = 1, hiFreq = 3, out = 0;
    var dur, frames, lfo, ptr, env, sig;
    dur = duration.asFloat; //total playback duration in seconds
    frames = BufFrames.kr(buf);
    lfo = SinOsc.ar(lfoFreq, 3pi/2).range(loFreq, hiFreq);
    ptr = SinOsc.ar(lfo, 3pi/2).range(0, frames - 1);
    env = Env.linen(attackTime: 0.02, sustainTime: (dur - 0.04), releaseTime: 0.02, level: 1);
    sig = BufRd.ar(1, buf, ptr) * EnvGen.ar(env, doneAction: 2);
    Out.ar(out, sig * amp);
} ).add;

SynthDef.new(\bufRead_noiseSineMod, {

```

```

    arg buf, duration, amp=1, lfoFreq = 0.25, loFreq = 1, hiFreq = 3, out = 0;
    var dur, frames, lfo, ptr, env, sig;
    dur = duration.asFloat; //total playback duration in seconds
    frames = BufFrames.kr(buf);
    //lfo = SinOsc.ar(lfoFreq, 3pi/2).range(loFreq, hiFreq);
    lfo = LFDNoise1.ar(lfoFreq).range(loFreq, hiFreq);
    ptr = SinOsc.ar(lfo, 3pi/2).range(0, frames - 1);
    env = Env.linen(attackTime: 0.02, sustainTime: (dur - 0.04), releaseTime: 0.02, level: 1);
    sig = BufRd.ar(1, buf, ptr) * EnvGen.ar(env, doneAction: 2);
    Out.ar(out, sig * amp);
  }).add;

SynthDef.new(\bufRead_sineSineMod_randPhase, { //<--NEW
  arg buf, duration, amp=1, lfoFreq = 0.25, loFreq = 1, hiFreq = 3, attTime = 0.02, relTime =
0.02, out = 0;
  var dur, frames, lfo, ptr, env, sig, randPhase;
  dur = duration.asFloat; //total playback duration in seconds
  frames = BufFrames.kr(buf);
  randPhase = Rand(0, 3pi/2);
  lfo = SinOsc.ar(lfoFreq, phase: randPhase).range(loFreq, hiFreq);
  ptr = SinOsc.ar(lfo, phase: randPhase).range(0, frames - 1);
  //lfo = SinOsc.ar(lfoFreq, Rand(0, 3pi/2)).range(loFreq, hiFreq);
  //ptr = SinOsc.ar(lfo, Rand(0, 3pi/2)).range(0, frames - 1);
  //env = Env.linen(attackTime: 0.02, sustainTime: (dur - 0.04), releaseTime: 0.02, level: 1,
curve: \sine);
  env = Env.linen(attackTime: attTime, sustainTime: (dur - (attTime + relTime)), releaseTime:
relTime, level: 1, curve: \sine);
  sig = BufRd.ar(1, buf, ptr, interpolation: 2) * EnvGen.ar(env, doneAction: 2);
  Out.ar(out, sig * amp);
}).add;

SynthDef.new(\bufRead_noiseSineMod_randPhase, { //<--NEW
  arg buf, duration, lfoFreq = 0.25, loFreq = 1, hiFreq = 3, attTime = 0.02, relTime = 0.02,
amp=1, out = 0;
  var dur, frames, lfo, ptr, env, sig, randPhase;
  dur = duration.asFloat; //total playback duration in seconds
  frames = BufFrames.kr(buf);
  randPhase = Rand(0, 2pi);
  //randPhase = Rand(0, 3pi/2);
  lfo = LFDNoise1.ar(lfoFreq).range(loFreq, hiFreq);
  ptr = SinOsc.ar(lfo, randPhase).range(0, frames - 1);
  env = Env.linen(attackTime: attTime, sustainTime: (dur - (attTime + relTime)), releaseTime:
relTime, level: 1, curve: \sine);
  sig = BufRd.ar(1, buf, ptr) * EnvGen.ar(env, doneAction: 2);
  Out.ar(out, sig * amp);
}).add;

SynthDef.new(\bufRead_noise3_randPhase, {lbuf, duration, modFreq = 0.25, loFreq = 1, hiFreq = 3,
attTime = 0.02, relTime = 0.02, amp=1, out = 0 //<--NEW
  var dur, frames, ptr, env, lfo, sig, randPhase;
  dur = duration.asFloat; //total playback duration in seconds
  frames = BufFrames.kr(buf);
  randPhase = Rand(0, 2pi);
  lfo = SinOsc.ar(modFreq, randPhase).range(loFreq, hiFreq);
  ptr = LFDNoise3.ar(lfo).range(0, frames - 1);
  env = Env.linen(attackTime: attTime, sustainTime: (dur - (attTime + relTime)), releaseTime:
relTime, level: 1);
  sig = BufRd.ar(1, buf, ptr) * EnvGen.ar(env, doneAction: 2);
  Out.ar(out, sig * amp);
}).add;

SynthDef.new(\bufRead_rateLFOsine, { //<-----NEW
  arg buf, duration = 2, amp=1, lfoRate = 0.25, minRate = 0.5, maxRate = 2, attTime = 0.02,
relTime = 0.02, out = 0;
  var frames, ptr, env, sig, phasor, randPhase;

```

```

frames = BufFrames.kr(buf);
randPhase = Rand(0, 3pi/2);
ptr = SinOsc.ar(lfoRate, randPhase).range(minRate, maxRate);
phasor = Phasor.ar(0, BufRateScale.kr(buf) * ptr, 0, frames - 1);
env = Env.linen(attackTime: attTime, sustainTime: duration - (attTime + relTime), releaseTime:
relTime, level: 1, curve: \sine);
sig = BufRd.ar(1, buf, phase: phasor) * EnvGen.ar(env, doneAction: 2);
Out.ar(out, sig * amp);
}).add;

SynthDef.new(\bufRead_noLoop, {
  arg buf, rate = 1, amp=1, out=0;
  var sig, frames, dur, env, phasor;
  frames = BufFrames.kr(buf);
  //dur = frames / BufSampleRate.kr(buf); // in seconds
  dur = BufDur.kr(buf); //in seconds
  env = Env.linen(attackTime: 0.02, sustainTime: (dur - 0.04), releaseTime: 0.02, level: 1);
  //phasor = Phasor.ar(0, BufRateScale.kr(buf) * rate, 0, frames);
  phasor = Phasor.ar(0, BufRateScale.kr(buf) * rate, 0, frames - 1);
  sig = BufRd.ar(1, buf, phasor, loop: 0) * EnvGen.ar(env, doneAction: 2);
  Out.ar(out, sig * amp);
}).add;

SynthDef.new(\bufRead, { //define synth to read from buffer w/ phasor...
  arg amp=1, out=0, phaseBus, buf;
  var sig = BufRd.ar(1, buf, InFeedback.ar(phaseBus), interpolation: 4); //cubic interpolation
  (seems to help preserve detail at faster playback rates +24st)
  sig = sig * amp;
  Out.ar(out, sig);
}).add;

// Phasor indexes through buffer at audio rate
SynthDef(\phasor, {
  //arg out=0, buf, trig = 0, rate = 1, start = 0, end = BufFrames.kr(buf) - 1, resetPos = 0;
  arg out=0, buf, trig = 0, rate = 1, start = 0, end = 1, resetPos = 0; //<---don't remember why I
  used other lines....and why they worked...
  //Out.ar(out, Phasor.ar(trig, BufRateScale.kr(buf) * rate, start * BufFrames.kr(buf), (end *
  BufFrames.kr(buf)) - 1, resetPos ));
  Out.ar(out, Phasor.ar(trig, BufRateScale.kr(buf) * rate, start, end, resetPos ));
}).add;

SynthDef(\a2k_patch, { // Needed to monitor phasor val (don't fully understand why / how this works)
  arg in, out;
  Out.kr(out, A2K.kr(InFeedback.ar(in) )); //is InFeedback necessary?
}).add;

SynthDef.new(\pitchShifter, {
  //arg inBus, outBus, windowSize = 0.1, pitchRatio = 1, pitchDispers = 0, timeDispers = 0.002,
  amp = 1; //v1
  arg inBus, outBus, windowSize = 0.1, pitchRatio = 1, pitchDispers = 0, timeDispers = 0.003,
  amp = 1; //v1.1, FOR TIME STRETCH (larger time Dispers better for time stretching, but worse for
  preserving percussive transients)
  //arg inBus, outBus, windowSize = 0.3, pitchRatio = 1, pitchDispers = 0, timeDispers = 0.001,
  amp = 1; //v2, PRESERVE TRANSIENTS DURING FAST PLAYBACK - good settings for faster
  playback (+24 rate, -24 pitch)
  //arg inBus, outBus, windowSize = 0.35, pitchRatio = 1, pitchDispers = 0, timeDispers = 0, amp
  = 1; //v2.1, FAST PLAYBACK (preserves transients better than v1.1 at fast playback rates) <-----
  //arg inBus, outBus, windowSize = 0.01, pitchRatio = 1, pitchDispers = 0, timeDispers = 0.02,
  amp = 1; //vPiez_1, decent settings for time stretched piezo...
  var sigIn, sigOut;
  sigIn = In.ar(inBus, 1);
  //sigIn = InFeedback.ar(inBus, 1);
  sigOut = PitchShift.ar(sigIn, windowSize, pitchRatio, pitchDispers, timeDispers, amp);
  Out.ar(outBus, sigOut);
}

```

```

}).add;

SynthDef.new(\XFade, {
  lin1, in2, out, amountControlBus, amp = 1 |
  var inArray, mixSignal, sig;
  inArray = [In.ar(in1), In.ar(in2)];
  //inArray = [InFeedback.ar(in1), InFeedback.ar(in2)];
  mixSignal = SelectX.ar(amountControlBus, inArray);
  sig = mixSignal * amp;
  Out.ar(out, sig);
}).add;

SynthDef(\LinInterp_pitch, {
  arg startVal, stopVal, duration, out;
  Out.kr(out, Line.kr(startVal, stopVal, duration, doneAction: 2).midiratio);
}).add;

SynthDef(\LinInterp_rate, {
  arg startVal, stopVal, duration, out, rateMod = 1;
  var outputVal = rateMod * Line.kr(startVal, stopVal, duration, doneAction: 2).midiratio;
  Out.kr(out, outputVal);
}).add;

SynthDef(\LinInterp_num, {
  arg startVal, stopVal, duration, mult = 1, out;
  Out.kr(out, Line.kr(startVal, stopVal, duration, mul: mult, doneAction: 2));
}).add;

SynthDef(\ExpInterp_num, {startVal, stopVal, duration, mult = 1, out |
  Out.kr(out, XLine.kr(startVal, stopVal, duration, mul: mult, doneAction: 2));
}).add;

SynthDef(\asrEnvGen_trig, {lout, attTime = 0.01, susTime = 1, relTime = 2, level = 1, curve = \sinel
  var env, envGen;
  env = Env.linen(attackTime: attTime, sustainTime: susTime, releaseTime: relTime, level: level,
curve: curve);
  envGen = EnvGen.kr(env, doneAction: 2);
  Out.kr(out, envGen);
}).add;

SynthDef(\adsrEnvGen, {lout, gate = 1, attTime = 0.01, susTime = 1, relTime = 2, peakLevel = 1,
susLevel = 1, curve = \sinel
  var env, envGen;
  env = Env.adsr(attackTime: attTime, decayTime: susTime, sustainLevel: susLevel,
releaseTime: relTime, peakLevel: peakLevel, curve: curve);
  envGen = EnvGen.kr(env, gate: gate, doneAction: 2);
  Out.kr(out, envGen);
}).add;

SynthDef(\dEnvGen, {lout, gate = 1, startVal = 0, endVal = 0, peakVal = 1, susVal = 1, attTime = 1,
decTime = 1, susTime = 1, relTime = 2, shape = 3, curve = 0, repeat = 1 | //shape: 3 => S-curve | 5 =>
linear | curve vals only effective when shape = 5 | <-----USE THIS W/ AUTO-LEVEL / ANTI-
FEEDBACK ROUTINES.....
  var levels, durations, env;
  levels = Dseq([startVal, peakVal, susVal, susVal, endVal], repeats: repeat);
  durations = Dseq([attTime, decTime, susTime, relTime], repeats: repeat);
  env = DemandEnvGen.kr(level: levels, dur: durations, shape: shape, curve: curve, gate: gate,
doneAction: 2); //if gate <=0, ugen's sampled and held, if gate >=1, it runs...
  Out.kr(out, env);
}).add;

SynthDef.new(\subMix, {
  lin, amp = 1, outl
  var sig = In.ar(in) * amp;
  //sig = InFeedback.ar(in) * amp;

```

```

        OffsetOut.ar(out, sig);
        //Out.ar(out, sig);
    }).add;

SynthDef(\comp1, {lbus, thresh = (-6.dbamp), ratio = 2, attack = 0.01, release = 0.25, amp = 1| //<----
NEW
    var sigIn, sigOut, ratioConv;
    sigIn = In.ar(bus: bus, numChannels: 1);
    ratioConv = ratio.reciprocal;
    sigOut = Compander.ar(sigIn, sigIn, thresh: thresh, slopeAbove: ratioConv, clampTime: attack,
relaxTime: release, mul: amp);
    ReplaceOut.ar(bus, sigOut);
}).add;

SynthDef.new(\mainOut, { //w/ Lo-cut and Peak EQ
    lin, loCut = 90, loCutQ = 1, amp = 1, peak = 120, peakQ = 1, peakGain = 0|
    var sig;
    sig = BHiPass4.ar(In.ar(in), freq: loCut, rq: loCutQ);
    sig = BPeakEQ.ar(sig, freq: peak, rq: peakQ, db: peakGain);
    sig = Limiter.ar(sig, level: (-0.3.dbamp), dur: 0.01); //<---NEW
    //sig = In.ar(in) * amp;
    //sig = InFeedback.ar(in) * amp;
    //OffsetOut.ar(0, sig);
    Out.ar([0, 1], sig * amp);
}).add;

/* //<-----ADDING ---- Main Out w/ soft Limiter
SynthDef.new(\mainOut, { //w/ Lo-cut and Limiter
    lin, loCutoff = 90, rQ = 1, amp = 1|
    var sig;
    sig = BHiPass4.ar(In.ar(in), loCutoff, rq: rQ, mul: amp);
    sig = Compander.ar(in: sig, control: sig, thresh: 0.95, slopeBelow: 1, slopeAbove: 0.1,
clampTime: 0.01, relaxTime: 0.1); //set threshold to -0.3.dbamp ?
    //OffsetOut.ar(0, sig);
    Out.ar([0, 1], sig);
}).add; */

SynthDef.new(\scopeOut, { //<-----NEW
    //linBus, buf, frames = 4096|
    linBus, buf|
    var sig;
    sig = In.ar(inBus, 1);
    ScopeOut2.ar(sig, buf);
    //ScopeOut2.ar(sig, buf, BufFrames.kr(buf));
    //ScopeOut2.ar(sig, buf, 262144, 524288);
    //ScopeOut2.ar(sig, buf, 2641920, 2641920);
    //ScopeOut2.ar(sig, buf, frames);
}).add;

SynthDef(write2Disk, {arg buf, inBus = 0; //record audio file to disk - mono
    DiskOut.ar(buf, In.ar(inBus, 1));
}).add;

// play audio file from disk w/ fade in-out
SynthDef(readDisk, { arg buf, sampleDur, amp = 1, out;
    var env, envgen, sig;
    env = Env.linen(attackTime: 0.02, sustainTime: (sampleDur - 0.04), releaseTime: 0.02, level:
1);
    envgen = EnvGen.ar(env, doneAction: 2);
    sig = (DiskIn.ar(1, buf) * envgen) * amp;
    Out.ar(out, sig);
}).add;

SynthDef.new(\startupTone_1, {
    var sig, amp, env;

```



```

        env = EnvGen.kr(
            Env.new( [0, 1, 0], [6, 4], [1, -1] ),
            doneAction: 2
        );
        amp = SinOsc.kr( {ExpRand(0.2, 12)} ! 3).range(0.1);
        sig = SinOsc.ar( {ExpRand(300, 1200)} ! 3);
        sig = sig * amp * env; //amp mod
        sig = Mix.ar(sig) * 0.20;
        Out.ar(0, sig);
    }).add;

// UTILITY -----
SynthDef(\avgKR, {
    lininput, output, avgDur = 11
    var sigIn, avg, numSamp;
    sigIn = In.kr(input, 1);
    numSamp = ControlRate.ir * avgDur;
    avg = (RunningSum.kr(sigIn, numSamp)) * (numSamp.reciprocal);
    Out.kr(output, avg);
}).add;

// FEATURE EXTRACTORS -----

SynthDef.new(\Loudness, {
    //arg input, buffer, outBus;
    arg input, outBus;
    var sigIn, fft, loudness;
    sigIn = In.ar(input);
    //fft = FFT(buffer, sigIn, wintype: 1);
    fft = FFT(LocalBuf(1024), sigIn, wintype: 1); //using LocalBuf per HelpFile recommendation,
    1024 frame size recommended for samp rates 44100 and 48000 and 2048 for rates 88200 and 96000
    loudness = Loudness.kr(fft);
    //loudness.poll(trig: 10, label: "Sones");
    Out.kr(outBus, loudness);
}).add;

SynthDef.new(\RMS, {
    arg input, numSamp = 40, outBus;
    var sigIn, rms;
    sigIn = In.ar(input);
    rms = (RunningSum.kr(sigIn.squared, numSamp) / numSamp).sqrt;
    //rms.poll(trig: 10, label: "RMS");
    Out.kr(outBus, rms);
}).add;

SynthDef.new(\RMSdb, {
    arg input, numSamp = 40, outBus;
    var sigIn, rms;
    sigIn = In.ar(input);
    rms = (RunningSum.kr(sigIn.squared, numSamp) / numSamp).sqrt;
    Out.kr(outBus, rms.ampdb);
}).add;

SynthDef.new(\Pitch, {
    arg input, ampThresh = 0.04, median = 7, minFreq = 60, maxFreq = 4000, clar = 0, outBus;
    var sigIn, freq, hasFreq;
    sigIn = In.ar(input);
    # freq, hasFreq = Pitch.kr(sigIn, ampThreshold: ampThresh, median: median, minFreq: minFreq,
    maxFreq: maxFreq, clar: clar);
    //freq = Lag.kr(freq.cpsmidi.round(1).midicps, 0.05);
    Out.kr(outBus, [freq, hasFreq]);
    //freq.poll(trig: 10, label: "Freq");
    //hasFreq.poll(trig: 10, label: "HasFreq");
}).add;

```

```

SynthDef.new(\Pitch2, {
    arg input, ampThresh = 0.04, median = 7, minFreq = 60, maxFreq = 4000, clar = 0, outBus;
    var sigIn, freq, hasFreq, midinote;
    sigIn = In.ar(input);
    # freq, hasFreq = Pitch.kr(sigIn, ampThreshold: ampThresh, median: median, minFreq: minFreq,
maxFreq: maxFreq, clar: clar);
    //midinote = Lag.kr(freq.cpsmidi.round(1), 0.05);
    midinote = freq.cpsmidi.round(1);
    Out.kr(outBus, [freq, hasFreq, midinote]);
    //freq.poll(trig: 10, label: "Freq");
    //hasFreq.poll(trig: 10, label: "HasFreq");
}).add;

SynthDef.new(\MFCC, {
    //arg input, buffer, outBus;
    arg input, outBus;
    var sigIn, fft, array;
    sigIn = In.ar(input);
    //fft = FFT(buffer, sigIn, wintype: 1);
    fft = FFT(LocalBuf(1024), sigIn, wintype: 1);
    array = MFCC.kr(fft); //outputs 13 coefficients by default
    Out.kr(outBus, array);
    //array.poll(trig: 10, label: "MFCCs");
}).add;

SynthDef.new(\specCent, {
    arg input, outBus;
    var sigIn, fft, centroid;
    sigIn = In.ar(input);
    fft = FFT(LocalBuf(2048), sigIn, wintype: 1);
    centroid = SpecCentroid.kr(fft);
    Out.kr(outBus, centroid);
}).add;

SynthDef.new(\specFlat, {
    arg input, outBus;
    var sigIn, fft, flatness, flatdb, flatdbScaled;
    sigIn = In.ar(input);
    fft = FFT(LocalBuf(2048), sigIn, wintype: 1);
    flatness = SpecFlatness.kr(fft);

    //flatdb = 10 * flat.log; //convert flatness to decibels
    //flatdbScaled = LinLin.kr(flatdb, -45, -1.6, 0, 1).max(-10); // Rescale db roughly to 0...1

    Out.kr(outBus, flatness);
}).add;

SynthDef.new(\onsets, {
    arg input, outBus, threshold = 0.5, trigtime = 0.1, odtype = \rcomplex, reltime = 1.0, floor = 0.1;
    var sigIn, fft, onsets, trigger;
    sigIn = In.ar(input);
    fft = FFT(LocalBuf(1024), sigIn, wintype: 1);
    onsets = Onsets.kr(fft, threshold, odtype, reltime, floor);
    trigger = Trig1.kr(onsets, trigtime);
    //trigger.poll(10, label: 'trig'); //uncomment this to monitor trigger val constantly
    Out.kr(outBus, trigger);
}).add;

//-----

SynthDef.new(\monAudioInput, { //not sure if I need this currently....
    arg inChannel = 0, amp = 1, outBus;
    var sig = SoundIn.ar(bus: inChannel, mul: amp); //defaults to first channel of system audio
input
    OffsetOut.ar(outBus, sig);

```

```

}).add;

// IMPULSE SOURCES -----

SynthDef(\susWhite1, {\lattTime = 3, susTime = 3, relTime = 3, amp = 1, outl
    var sig, env;
    env = Env.linex(attackTime: attTime, sustainTime: susTime, releaseTime: relTime);
    sig = WhiteNoise.ar(1);
    sig = sig * EnvGen.ar(env, doneAction: 2);
    Out.ar(out, sig * amp);
}).add;

SynthDef(\impulsePink1, {
    ldecayTime = 2, amp = 0.2, outl
    var sig = Decay.ar(impulse.ar(0), decayTime, PinkNoise.ar(amp));
    DetectSilence.ar(sig, amp: 0.0001, time: 0.5, doneAction: 2);
    Out.ar(out, sig);
}).add;

SynthDef(\impulseWhite1, {
    ldecayTime = 2, amp = 0.2, outl
    var sig = Decay.ar(impulse.ar(0), decayTime, WhiteNoise.ar(amp));
    DetectSilence.ar(sig, amp: 0.0001, time: 0.5, doneAction: 2);
    Out.ar(out, sig);
}).add;

SynthDef(\impulseWhite2, {
    lattackTime = 0.002, releaseTime = 0.4, curve = -4, amp = 0.2, outl
    var env, sig;
    env = Env.perc(attackTime, releaseTime, curve: curve);
    sig = WhiteNoise.ar(1);
    sig = sig * EnvGen.ar(env, doneAction: 2);
    Out.ar(out, sig * amp);
}).add;

SynthDef(\impulseWhite3, {
    lattackTime = 0.01, releaseTime = 0.25, curve = -4, shelf_freq = 2500, shelf_db = -35, rs = 1,
    peak_freq = 160, peakQ = 1, peak_db = -10, lop_freq = 5000, lop_q = 0.4, amp = 0.2, outl
    var env, sig;
    env = Env.perc(attackTime, releaseTime, curve: curve);
    sig = WhiteNoise.ar(1);
    sig = sig * EnvGen.ar(env, doneAction: 2);
    sig = BLPass4.ar(sig, freq: lop_freq, rq: lop_q);
    sig = BHPShelf.ar(sig, freq: shelf_freq, rs: rs, db: shelf_db);
    sig = BPEQ.ar(sig, freq: peak_freq, rq: peakQ, db: peak_db, mul: amp);
    Out.ar(out, sig);
}).add;

SynthDef(\impulseWhiteSin, {
    lattackTime = 0.01, releaseTime = 0.25, curve = -4, peak_freq = 160, peakBW = 0.01, amp =
    0.2, outl
    var env, sin, sig1, sig2, sig3, sig4, sig5, sigOut;
    env = Env.perc(attackTime, releaseTime, curve: curve);
    sin = SinOsc.ar(freq: peak_freq);
    sig1 = WhiteNoise.ar(1);
    sig2 = Resonz.ar(sig1, freq: peak_freq, bwr: peakBW);
    sig3 = Resonz.ar(sig1, freq: (peak_freq * 2), bwr: peakBW);
    sig4 = Resonz.ar(sig1, freq: (peak_freq * 4), bwr: peakBW, mul: 0.75);
    sig5 = Resonz.ar(sig1, freq: (peak_freq * 10), bwr: peakBW, mul: 0.85);
    sigOut = Mix.ar([sin, sig2, sig3, sig4, sig5]);
    //sigOut = Mix.ar([sig2, sig3, sig4]);
    sigOut = sigOut * EnvGen.ar(env, doneAction: 2);
    Out.ar(out, sigOut * amp);
}).add;

```

```

SynthDef(\resPinkNoise, {lout, freq = 500, bw = 0.005, amp = 0.2l
  var sig = Resonz.ar(PinkNoise.ar(1), freq: freq, bwr: bw);
  Out.ar(out, sig * amp);
}).add;

SynthDef(\sinSweep1, {
  lfreq1 = 100, freq2 = 2000, dur = 0.5, amp = 0.2, outl
  var freq, env, sig;
  freq = XLine.kr(start: freq1, end: freq2, dur: dur);
  env = Env.linen(attackTime: 0.02, sustainTime: dur - 0.04, releaseTime: 0.02);
  sig = SinOsc.ar(freq);
  sig = sig * EnvGen.ar(env, doneAction: 2);
  Out.ar(out, sig * amp);
}).add;

SynthDef(\sawSweep1, {
  lfreq1 = 100, freq2 = 2000, dur = 0.5, amp = 0.2, outl
  var freq, env, sig;
  freq = XLine.kr(start: freq1, end: freq2, dur: dur);
  env = Env.linen(attackTime: 0.02, sustainTime: dur - 0.04, releaseTime: 0.02);
  sig = Saw.ar(freq);
  sig = sig * EnvGen.ar(env, doneAction: 2);
  Out.ar(out, sig * amp);
}).add;

SynthDef(\triSweep1, {
  lfreq1 = 100, freq2 = 2000, dur = 0.5, amp = 0.2, outl
  var freq, env, sig;
  freq = XLine.kr(start: freq1, end: freq2, dur: dur);
  env = Env.linen(attackTime: 0.02, sustainTime: dur - 0.04, releaseTime: 0.02);
  sig = LFTri.ar(freq);
  sig = sig * EnvGen.ar(env, doneAction: 2);
  Out.ar(out, sig * amp);
}).add;

SynthDef(\pulseSweep1, {
  lfreq1 = 100, freq2 = 2000, dur = 0.5, width = 0.5, amp = 0.2, outl
  var freq, env, sig;
  freq = XLine.kr(start: freq1, end: freq2, dur: dur);
  env = Env.linen(attackTime: 0.02, sustainTime: dur - 0.04, releaseTime: 0.02);
  sig = Pulse.ar(freq, width);
  sig = sig * EnvGen.ar(env, doneAction: 2);
  Out.ar(out, sig * amp);
}).add;

SynthDef.new(\randSinSwell_1, {lout = 0, amp = 1, loFreq = 300, hiFreq = 1200, attTime = 6, susTime =
2, relTime = 5l
  var sig, ampMod, env, envGen;
  env = Env.linen(attackTime: attTime, sustainTime: susTime, releaseTime: relTime, level: 1,
curve: \sine);
  envGen = EnvGen.kr(env, doneAction: 2);
  ampMod = SinOsc.kr( {ExpRand(0.2, 12)} ! 3).range(0.1);
  sig = SinOsc.ar( {ExpRand(loFreq, hiFreq)} ! 3);
  sig = sig * ampMod * envGen; //amp mod
  sig = Mix.ar(sig) * amp;
  Out.ar(out, sig);
}).add;

SynthDef.new(\randSinSwell_lim_1, {lout = 0, amp = 1, loFreq = 300, hiFreq = 1200, attTime = 6,
susTime = 2, relTime = 5, limitThresh = 0.9l
  var sig, ampMod, env, envGen;
  env = Env.linen(attackTime: attTime, sustainTime: susTime, releaseTime: relTime, level: 1,
curve: \sine);
  envGen = EnvGen.kr(env, doneAction: 2);

```

```

        ampMod = SinOsc.kr( {ExpRand(0.2, 12)} ! 3).range(0.1);
        sig = SinOsc.ar( {ExpRand(loFreq, hiFreq)} ! 3);
        sig = sig * ampMod * envGen; //amp mod
        sig = Limiter.ar(in: sig, level: limitThresh, dur: 0.01);
        sig = Mix.ar(sig) * amp;
        Out.ar(out, sig);
    }).add;

SynthDef(\sineBeats, {
    lfreq1 = 100, freq2 = 108, attTime = 3, susTime = 3, relTime = 3, amp = 0.2, limitThresh = 0.9,
    out = 0!
    var freq, env, sig1, sig2, sigOut;
    env = Env.linen(attackTime: attTime, sustainTime: susTime, releaseTime: relTime, curve:
\sine);
    sig1 = SinOsc.ar(freq1);
    sig2 = SinOsc.ar(freq2);
    sigOut = Mix.ar([sig1, sig2]);
    sigOut = sigOut * EnvGen.ar(env, doneAction: 2);
    sigOut = Limiter.ar(in: sigOut, level: (limitThresh), dur: 0.01);
    Out.ar(out, sigOut * amp);
}).add;

SynthDef(\sineBeats_rand, {
    lfreqMin = 100, freqMax = 500, beatHzMin = 8, beatHzMax = 20, attTimeMin = 3, attTimeMax
= 7, susTimeMin = 3, susTimeMax = 5, relTimeMin = 2.1, relTimeMax = 4, amp = 0.5, limitThresh = 0.9,
    out = 0!
    var beatHz, freq1, freq2, env, sig1, sig2, sigOut;
    env = Env.linen(attackTime: rrand(attTimeMin, attTimeMax).round(0.01), sustainTime:
rrand(susTimeMin, susTimeMax).round(0.01), releaseTime: rrand(relTimeMin,
relTimeMax).round(0.01), curve: \sine);
    beatHz = IRand(beatHzMin, beatHzMax);
    freq1 = Rand(freqMin, freqMax);
    freq2 = freq1 + beatHz; //<---beating frequency always higher than selected freq
    sig1 = SinOsc.ar(freq1);
    sig2 = SinOsc.ar(freq2);
    sigOut = Mix.ar([sig1, sig2]);
    sigOut = sigOut * EnvGen.ar(env, doneAction: 2);
    sigOut = Limiter.ar(in: sigOut, level: (limitThresh), dur: 0.01);
    Out.ar(out, sigOut * amp);
}).add;

// OTHER EFFECTS PROCESSING ---

SynthDef(\partConv_sample, {ltargetBuf, irSpectrum, fftSize = 2048, playRate = 1, loop = 1, ampBus,
ampLimit = 0.4, out!
    var input, sig, currentAmp;
    input = PlayBuf.ar(1, targetBuf, rate: playRate, loop: loop);
    sig = PartConv.ar(input, fftsize: fftSize, irbufnum: irSpectrum);
    sig = Limiter.ar(sig, level: -1.dbamp, dur: 0.01);
    currentAmp = ampBus * ampLimit;
    Out.ar(out, sig * currentAmp);
    DetectSilence.ar(sig, amp: 0.0001, time: 0.2, doneAction: 2);
}).add;

SynthDef(\partConv_sampleLoop, {ltargetBuf, irSpectrum, attTime = 4, susTime = 5, relTime = 6, fftSize
= 2048, playRate = 1, ampBus, ampLimit = 0.4, out = 0!
    var input, env, sig, currentAmp;
    input = PlayBuf.ar(1, targetBuf, rate: playRate, loop: 1);
    env = Env.linen(attackTime: attTime, sustainTime: susTime, releaseTime: relTime, curve:
\sine);
    sig = PartConv.ar(input, fftsize: fftSize, irbufnum: irSpectrum);
    sig = sig * EnvGen.ar(env, doneAction: 2);
    sig = Limiter.ar(sig, level: -0.3.dbamp, dur: 0.01);
    currentAmp = ampBus * ampLimit;
    Out.ar(out, sig * currentAmp);

```

```

}).add;

SynthDef(\partConv_sampleLoop_pShift, {\targetBuf, irSpectrum, attTime = 4, susTime = 5, relTime = 6,
fftSize = 2048, playRate = 1, pitch = 1, winSize = 0.1, pitchDisp = 0, timeDisp = 0.003, pShiftMix = 1,
ampBus, ampLimit = 0.4, out = 0l
    var sig1, sig2, env, sigOut, currentAmp;
    sig1 = PlayBuf.ar(1, targetBuf, rate: playRate, loop: 1);
    sig2 = PitchShift.ar(sig1, windowSize: winSize, pitchRatio: pitch, pitchDispersion: pitchDisp,
timeDispersion: timeDisp);
    sigOut = SelectX.ar(pShiftMix, [sig1, sig2]);
    env = Env.linen(attackTime: attTime, sustainTime: susTime, releaseTime: relTime, curve:
\sine);
    sigOut = PartConv.ar(sigOut, fftsize: fftSize, irbufnum: irSpectrum);
    sigOut = sigOut * EnvGen.ar(env, doneAction: 2);
    sigOut = Limiter.ar(sigOut, level: -0.3.dbamp, dur: 0.01);
    currentAmp = ampBus * ampLimit;
    Out.ar(out, sigOut * currentAmp);
}).add;

);

```

File – “03_Busses.scd”:

```

//Sonic Mirror - 03_Busses:

(
~recInPosBus = Bus.control(s, 1);
~recOutPosBus = Bus.control(s, 1);
~playInPosBus = Bus.control(s, 1);
~playOutPosBus = Bus.control(s, 1);

~rateBus = Bus.control(s, 1);
~pitchBus = Bus.control(s, 1);
~pitchShiftMixValBus = Bus.control(s, 1); //mix % between orig and pitch-shifted signal
~pitchShiftOut = Bus.audio(s, 1);

~inputStreamLevelBus = Bus.control(s, 1); //level of audio output from ~pShiftMix synth (rename to
stream1level?)
~fileOutputLevelBus = Bus.control(s, 1); //level of audio file playback (rename to stream2level or
sampPbackLevelBus?)

~revRateBusMon = Bus.control(s, 1);

~phaseBusRec = Bus.audio(s, 1);
~phaseBusMonRec = Bus.control(s, 1);
~inputLevelBus = Bus.control(s, 1);

~loCutBus_in = Bus.control(s, 1);
~loCutBus_out = Bus.control(s, 1);
~inputSendBus = Bus.audio(s, 1);

~phaseBusPly = Bus.audio(s, 1);
~phaseBusMonPly = Bus.control(s, 1);
~play1LevelBus = Bus.control(s, 1); //to control level of audioPlayback ("playhead1")
~playOut1Bus = Bus.audio(s, 1);

~stream1sub = Bus.audio(s, 1); //<---NEW
~stream2sub = Bus.audio(s, 1); //<---NEW

~outMainLevelBus = Bus.control(s, 1);
~outMainBus = Bus.audio(s, 1);

~outMainPeakFreq = Bus.control(s, 1);
~outMainPeakQ = Bus.control(s, 1);

```

```

~outMainPeakGain = Bus.control(s, 1);

~comp1thresh = Bus.control(s, 1);
~comp1ratio = Bus.control(s, 1);
~comp1attack = Bus.control(s, 1);
~comp1release = Bus.control(s, 1);
~comp1gain = Bus.control(s, 1);

// Busses for Feature Extractor Synths (from SCML) -----

~bus_loudness = Bus.control(s, 1);
~bus_rms = Bus.control(s, 1);
~bus_mfcc = Bus.control(s, 13);
~bus_pitch = Bus.control(s, 2);
~bus_pitch2 = Bus.control(s, 3);
~bus_specCent = Bus.control(s, 1);
~bus_specFlat = Bus.control(s, 1);
~bus_onsets = Bus.control(s, 1);
);

```

File – “04_Functions.scd”:

```

//Sonic Mirror - 04_Functions:

// --> functions and routines

(

//===== FUNCTIONS =====
~getRecPlayOffset = {linSeconds = true, bufferI
    var recPos, playPos, offset;
    recPos = ~phaseBusMonRec.getSynchronous.round(1);
    playPos = ~phaseBusMonPly.getSynchronous.round(1);
    if (recPos >= playPos,
        {offset = recPos - playPos;},
        {offset = (buffer.numFrames - playPos) + recPos;} //if playPos > recPos, calc offset by
wrapping around buffer...
    );
    if (linSeconds == true,
        {(offset/s.sampleRate).round(0.001)}, //return offset in seconds
        {offset;} //if linSeconds == false, return offset in frames
    );
};

~recPlayDelayOffset = {loffset = 0/*, gui = true*/ //offset in seconds <-----NEW
    var recPos, playPos;
    recPos = ~phaseBusMonRec.getSynchronous;
    playPos = recPos - (offset * s.sampleRate).round(1);
    if (playPos < 0, {playPos = (~audioBuf0.numFrames - 1) + playPos;});

    if (~bufPlayer == nil, //if ~bufPlayer ("PLAY") has never been triggered (avoids thrown error)
        {
            ~lastPlayPhasePos = playPos; //update env variable
            /*if (gui == true,
                {
                    ~slider_playPos.value = playPos.linlin(0,
                    (~audioBuf0.numFrames - 1), 0, 1); //update slider pos
                    ~numBox_playPos.value =
                    (playPos/s.sampleRate).round(0.01); //convert frames to seconds and display
                }
            ); */
        },
        { //else if ~bufPlayer ("PLAY") has been triggered actively playing from buffer
            if (~bufPlayer.isRunning == false,

```

```

("PLAY") not active      {~lastPlayPhasePos = playPos;}, //update env variable if ~bufPlayer
phasor position          {~phasor_play.set(\resetPos, playPos, \trig, 1);} //else update play
                          );
                          }
                          );

      playPos; //<---return offset frame value
};

~sumFunc_xin = {lbusArray, sumSynth| //bus val concat function for variable size arg array of feat
busses
    var numCh = 0, busCol, outArray, outBus;
    busArray.do({litem| numCh = numCh + item.numChannels}); //calculate total # bus channels
    outBus = Bus.control(s, numCh);
    busCol = Array.newClear(busArray.size);

    sumSynth.envirPut{//create synth to route outputs from different busses to one sum bus w/
correct # of chans
    SynthDef(sumSynth, {//give synth a unique name to avoid conflicts if multiples of this
func are active at once
        busArray.do({litem, il busCol[i] = ln.kr(item, item.numChannels); }); //input
bus vals to var array
        busCol.do({litem| //concat all bus vals into single out array
            if (item.numChannels == 1,
                {outArray = outArray ++ [item]},
                {outArray = outArray ++ (item.numChannels.collect( {lil
item[i]} )))
            );
        });
        Out.kr(outBus, outArray);
    }).play(addAction: \addToTail);
};

    outBus; //return new bus w/ concat values of all input busses in arg array
};

~get_busSum = {lbus, post = 1|
    var numCh = bus.numChannels;
    bus.get( {arg val;
        if (post == 1, {//monitor feature vals in SC post window
            switch (bus,
                ~busSum_ldns_mfcc, {
                    ("Loudness:" + val[0].round(0.0001)).postln;
                    ("MFCC:" + val[1..13].round(0.0001)).postln;
                    ("-----").postln;
                },
                ~busSum_ldns_sFlat_sCent, {
                    ("Loudness:" + val[0].round(0.0001)).postln;
                    ("Flatness:" + val[1].round(0.0001)).postln;
                    ("Centroid:" + val[2].round(0.0001)).postln;
                    ("-----").postln;
                },
                ~busSum_ldns_pitch, {
                    ("Loudness:" + val[0].round(0.0001)).postln;
                    ("Pitch-freq:" + val[1].round(0.01)).postln;
                    ("Pitch-hasFreq?:" + val[2]).postln;
                    ("Pitch-MIDInote:" + val[3]).postln;
                    ("-----").postln;
                },
                ~busSum_ldns_sFlat_onset, {
                    ("Loudness:" + val[0].round(0.0001)).postln;
                    ("Flatness:" + val[1].round(0.0001)).postln;
                    ("Onset Detected:" + val[2]).postln;
            }
        })
    })
}

```



```

        ("-----").postln;
    },
    /*~bus_rms, {
        ("RMS:" + val.round(0.0001) + "I db:" +
val.ampdb.round(0.01)).postln;
    }, */
    { //all other busses (cases)
        if (numCh > 1,
            {
                numCh.do({arg item; ("Feature_" ++
(item + 1) ++ ":" + val[item].round(0.0001)).postln;});
                ("-----").postln;
                { ("Feature_1:" + val.round(0.0001)).postln; } //else
            }
        );
    }
});

//INSERT SWITCH CASE HERE FOR EACH DIFFERENT SUM BUS
//....W/ NESTED CONDITIONALS SETTING ~currentSound based on extracted
audio...

//push ~currentSound into "sound memory array / buffer" periodically <--put this in
~autoRec routine?
//update ~currentScene based on "sound memory array" periodically <--put this in
~autoRec routine?

});
};

~streamFeatures = {
    lbus, rate = 0.1, post = 1
    Routine ( {
        ~get_busSum.value(bus, post);
        rate.yieldAndReset;
    });
};

//AUTO REC / PLAY HELPER FUNCTIONS -----
~writeSampFromBuf_name = {lbuffer, in, out, inputSec = 1, filecap = false, filePath //expecting in and
out time in seconds
    var inFrame, outFrame;
    if (inputSec == 1, { //in/out times are assumed to be in seconds
        inFrame = (in.asFloat.round(0.001) * buffer.sampleRate);
        outFrame = (out.asFloat.round(0.001) * buffer.sampleRate);
    }, { //if inputSec == 0, in/out times are assumed to be in frames
        inFrame = in.asFloat.round(1);
        outFrame = out.asFloat.round(1);
    });

    if (inFrame < outFrame, { //if sample does not wrap around to beginning of buffer
        var numFrames = (outFrame - inFrame) + 1;
        buffer.write(filePath, "wav", ~sampBitDepth, numFrames: numFrames, startFrame:
inFrame);
        ("WROTE SAMPLE: " ++ filePath + "I dur = " ++
(numFrames/buffer.sampleRate).round(0.01) + "I In = " ++ in.round(0.01) + "I Out = " ++
out.round(0.01)).postln;
        //~fileAudio_writeln_nameNum = ~fileAudio_writeln_nameNum + 1;

```

```

    }, { //if sample wraps around to beginning of buffer
        var numFrames, chunk1frames, chunk2frames, copyBuf;
        chunk1frames = buffer.numFrames - inFrame; //inclusive frame count from inFrame to
last frame in buffer
        chunk2frames = outFrame + 1; //inclusive frame count (b/c buffers start at frame 0)
        numFrames = chunk1frames + chunk2frames;
        copyBuf = Buffer.alloc(s, numFrames, 1);
        buffer.copyData(copyBuf, dstStartAt: 0, srcStartAt: inFrame, numSamples:
chunk1frames);
        buffer.copyData(copyBuf, dstStartAt: (chunk1frames - 1), srcStartAt: 0, numSamples:
chunk2frames);
        copyBuf.write(filePath, "wav", ~sampBitDepth);
        copyBuf.free;
        ("WROTE SAMPLE: " ++ filePath + "I dur = " ++
(numFrames/buffer.sampleRate).round(0.01) + "I In = " ++ in.round(0.01) + "I Out = " ++
out.round(0.01)).postln;
        //~fileAudio_writeln_nameNum = ~fileAudio_writeln_nameNum + 1;
    });

    ~fileAudio_writeln_nameNum = ~fileAudio_writeln_nameNum + 1;
    ~fileAudio_writeln_lastFile2 = (~fileAudio_writeln_directory ++ ~fileAudio_writeln_filename ++
~fileAudio_writeln_nameNum ++ ".wav").standardizePath; //<--- NEW

    ~recSamp_input = false; //from orig (from orig, for use in routines)
    {~button_recln.value = 0}.defer; //update GUI (from orig, for use in routines)

    if (filecap == true,
    {
        if (~fileAudio_writeln_nameNum > (~fileAudio_writeln_maxNum - 1), { //cap
max file count...
            ~fileAudio_writeln_nameNum = 0;
            ~fileAudio_writeln_maxNum_exceeded = true;
        });
    });

    };

    ~writeSampFromBuf = { |buffer, in, out, filecap = false | //expecting in and out time in seconds
        var inFrame, outFrame;
        inFrame = (in.asFloat.round(0.001) * buffer.sampleRate);
        outFrame = (out.asFloat.round(0.001) * buffer.sampleRate);

        if (inFrame < outFrame, { //if sample does not wrap around to beginning of buffer
            var numFrames = (outFrame - inFrame) + 1;
            buffer.write((~fileAudio_writeln_lastPath ++ ~fileAudio_writeln_nameNum +
+".wav").standardizePath, "wav", ~sampBitDepth, numFrames: numFrames, startFrame: inFrame);
            ("WROTE SAMPLE: " ++ ~fileAudio_writeln_lastPath ++
~fileAudio_writeln_nameNum ++ ".wav I dur = " ++ (numFrames/buffer.sampleRate).round(0.01) + "I In
= " ++ in.round(0.01) + "I Out = " ++ out.round(0.01)).postln;
            //~fileAudio_writeln_nameNum = ~fileAudio_writeln_nameNum + 1;
        }, { //if sample wraps around to beginning of buffer
            var numFrames, chunk1frames, chunk2frames, copyBuf;
            chunk1frames = buffer.numFrames - inFrame; //inclusive frame count from inFrame to
last frame in buffer
            chunk2frames = outFrame + 1; //inclusive frame count (b/c buffers start at frame 0)
            numFrames = chunk1frames + chunk2frames;
            copyBuf = Buffer.alloc(s, numFrames, 1);
            buffer.copyData(copyBuf, dstStartAt: 0, srcStartAt: inFrame, numSamples:
chunk1frames);
            buffer.copyData(copyBuf, dstStartAt: (chunk1frames - 1), srcStartAt: 0, numSamples:
chunk2frames);
            copyBuf.write((~fileAudio_writeln_lastPath ++ ~fileAudio_writeln_nameNum +
+".wav").standardizePath, "wav", ~sampBitDepth);
            copyBuf.free;

```

```

        ("WROTE SAMPLE: " ++ ~fileAudio_writeln_lastPath ++
~fileAudio_writeln_nameNum ++ ".wav | dur = " ++ (numFrames/buffer.sampleRate).round(0.01) + " | In
= " ++ in.round(0.01) + " | Out = " ++ out.round(0.01)).postln;
        //~fileAudio_writeln_nameNum = ~fileAudio_writeln_nameNum + 1;
    });

    ~fileAudio_writeln_nameNum = ~fileAudio_writeln_nameNum + 1;

    if (filecap == true,
        {
            if (~fileAudio_writeln_nameNum > (~fileAudio_writeln_maxNum - 1), { //cap
max file count...
                ~fileAudio_writeln_nameNum = 0;
                ~fileAudio_writeln_maxNum_exceeded = true;
            });
        }
    );

~writeSampFromBufCopy_input = {
    var temp;
    ("-----").postln;
    (">>>SAMPLING ~audioBuf0.....<<<").postln;

    if (~recSamp_input_startFrame < ~recSamp_input_stopFrame,
        { //SIMPLE BUF WRITE

            var recFrames = (~recSamp_input_stopFrame -
~recSamp_input_startFrame) + 1; //+1 to include stopFrame in frame count

            ~audioBuf0.write((~fileAudio_writeln_lastPath+
+~fileAudio_writeln_nameNum++ ".wav").standardizePath, "wav", ~sampBitDepth, startFrame:
~recSamp_input_startFrame, numFrames: recFrames); // write sampled buffer to audio file - directly
from original buffer...

            //LOG DURATION OF THIS RECORDED SAMPLE HERE, OR EMBED IN
FILE NAME? <-----

            ("--SIMPLE Write--").postln;
            ("--WROTE Sample: " ++ ~fileAudio_writeln_lastPath+
+~fileAudio_writeln_nameNum++ ".wav").postln;
            ("--sampled from sec: " ++ (~recSamp_input_startFrame /
s.sampleRate).round(0.01) ++ " - " ++ (~recSamp_input_stopFrame / s.sampleRate).round(0.01) ++ "
(dur = " ++ (recFrames/s.sampleRate).round(0.01) ++ " sec)").postln;
        },
        { //else if ~recSamp_input_startFrame > stopFrame, we've wrapped around the buffer
and will need to copy from it in 2 steps before writing an audio file...
            var recFrames, bufChunk1frames, bufChunk2frames;
            bufChunk1frames = ~audioBuf0.numFrames -
~recSamp_input_startFrame; //inclusive frame count from "startFrame" to last frame in buffer
            bufChunk2frames = ~recSamp_input_stopFrame + 1; //inclusive frame count
(b/c buffers start at frame 0)

            recFrames = bufChunk1frames + bufChunk2frames; //total # of frames to
sample

            //LOG DURATION OF THIS RECORDED SAMPLE HERE, OR EMBED IN
FILE NAME? <----

            ~audioBuf_inMainWrite = Buffer.alloc(s, recFrames, 1);

            //copy chunk 1 to new buf from ~recSamp_startFrame to last frame of
~audioBuf0
            ~audioBuf0.copyData(~audioBuf_inMainWrite, dstStartAt: 0, srcStartAt:
~recSamp_input_startFrame, numSamples: bufChunk1frames); //<---if "NO BUF data" errors thrown, try

```

reducing bufChunk1frames by 1, not sure if this method counts samples inclusive or exclusive of start frame

```

//copy chunk 2 to new buf from first frame of ~audioBuf0 to
~recSamp_stopFrame
    ~audioBuf0.copyData(~audioBuf_inMainWrite, dstStartAt:
(bufChunk1frames - 1), srcStartAt: 0, numSamples: bufChunk2frames);

    //write ~audioBuf_inMainWrite to an audio file
    ~audioBuf_inMainWrite.write((~fileAudio_writeln_lastPath+
+~fileAudio_writeln_nameNum++".wav").standardizePath, "wav", ~sampBitDepth);

    ("--WRAP AROUND 2-STEP WRITE--").postln;
    ("--WROTE Sample: " ++ ~fileAudio_writeln_lastPath+
+~fileAudio_writeln_nameNum++".wav").postln;
    ("--WRAP-AROUND--sampled from sec: " ++ (~recSamp_input_startFrame /
s.sampleRate).round(0.01) ++ " - " ++ (~recSamp_input_stopFrame / s.sampleRate).round(0.01) ++ "
(dur = " ++ (recFrames/s.sampleRate).round(0.01) ++ " sec)").postln;
    ~audioBuf_inMainWrite.free;
    }
};

~recSamp_input = false; //EXIT THIS HALF OF ROUTINE, resume listening for sound...
{~button_recln.value = 0}.defer; //update GUI

~fileAudio_writeln_lastFile = (~fileAudio_writeln_lastPath++~fileAudio_writeln_nameNum+
+".wav").standardizePath; //<----- NEW (keeps track of last written file name)

temp = ~fileHistory_rec.shift(1, ~fileAudio_writeln_lastFile); //<---NEW - rec file memory
~fileHistory_rec = temp; //<---NEW

~fileAudio_writeln_nameNum = ~fileAudio_writeln_nameNum + 1; //increment file counter

if (~fileAudio_writeln_nameNum > (~fileAudio_writeln_maxNum - 1), { //cap max file count...
    ~fileAudio_writeln_nameNum = 0;
    ~fileAudio_writeln_maxNum_exceeded = true;
});

(">>>RECORDING COMPLETE<<<").postln;
};

~sample_randAudify = {lbuffer, duration, ampBus, outBus, post = 0|
    var rate, pitch, mix;
    pitch = 1;
    mix = 0;

    if (0.5.coin == true, //50% faster or slower rate
        {//FASTER rate
            var rate1, origDuration;
            rate = rrand(2.0, 24.0).round(0.01);
            if (post == 1, {"==FASTER Rate==" + rate).postln;});
            rate1 = rate; //store semitone transp value for reference in case double play
triggered

            if (0.65.coin == true, {//65% chance pitch shift
                if (post == 1, {"--PITCH TRANSP--".postln;});
                if (0.65.coin == true,
                    {//65% chance pitch is inversion of rate - STRETCH
                        pitch = (rate * -1) - rrand(0.0, 2.0).round(0.01);
                        if (post == 1, {"---STRETCH - rate =" + rate +
"pitch =" + pitch).postln;});
                    },
                    {//35% chance pitch transp doesn't correspond to rate
                        if (rate <= 12,
                            {

```



```

rate2);}.defer(duration); //change rate after first playthrough
{~samp0.set(\rate,
duration = duration + duration2;
});
}, {/~10% chance second playback w/ no additional
transposition duration = duration * 2;
});
});
},
{ //slower rate
rate = rrand(-24.0, -2.0).round(0.01);
if (post == 1, {"==SLOWER Rate==" + rate).postln;});
if (0.65.coin == true, { //65% chance pitch shift
if (post == 1, {"--PITCH TRANSP--".postln;});
if (0.65.coin == true,
{ //65% chance pitch is inversion of rate - STRETCH
pitch = rate.abs + rrand(0.0, 1.0).round(0.01);
if (post == 1, {"---STRETCH - rate =" + rate +
"pitch =" + pitch).postln;});
},
{ //35% chance pitch transp doesn't correspond to rate
if (rate >= -6,
{
if (0.5.coin == true, {pitch =
rrand(-7.0, -1.0).round(0.01);}, {pitch = rrand(1.0, 12.0).round(0.01);});
if (post == 1, {"---Pitch Change
(wide) - rate =" + rate + "pitch =" + pitch).postln;});
},
{ //else if rate < -6
pitch = rrand(1.0,
12.0).round(0.01);
if (post == 1, {"---Pitch Change
(upwards) - rate =" + rate + "pitch =" + pitch).postln;});
}
});
});
mix = 1; //set mix so we hear pitch shifter within synth
});
rate = rate.midiratio;
pitch = pitch.midiratio;
duration = duration / rate; //update duration based on transp rate
if (duration > 30, //if transposed sample longer than 30 seconds, shorten
{
if (post == 1, {});
duration = rrand(15.0, 25.0);
if (post == 1, {
("----> SHORTEN DUR: Orig Dur: " ++
duration).postln;
("---->NEW DUR = " ++ duration).postln;
});
});
});
});
~samp0 = Synth(\bufRead_complex1, [\buf, buffer, \amp, ampBus, \out, outBus, \rate, rate,
\pitch, pitch, \pShiftMix, mix, \duration, duration]);

```

```

        duration;
    };

    ~sample_noiseMangle = { |buffer, duration, ampBus, outBus, post = 0|
        if (duration <= 2.5,
            {
                duration = (duration * 2).round(0.001);
                if (post == 1, {
                    ("DOUBLE SHORT DUR: OrigDur =" + duration + "!").post;
                    ("newDur =" + duration).postln;
                });
            },
            {
                if (duration <= 8,
                    {
                        duration = (duration * 1.25).round(0.001);
                        if (post == 1, {
                            ("1.25X MED: OrigDur =" + duration + ">").post;
                            ("newDur =" + duration).postln;
                        });
                    },
                    {
                        duration = duration.round(0.001);
                        if (post == 1, {"No CHANGE: dur = " ++
duration).postln;});
                    }
                );
            }
        );

        if (0.25.coin == true,
            { //straight noise
                var freq = rrand(0.5, 4.2).round(0.01);
                ~samp0 = Synth(\bufRead_noise1, [\buf, buffer, \duration, duration,
\modFreq, 1, \loFreq, freq, \hiFreq, freq, \amp, ampBus, \out, outBus]);
                if (post == 1, {"-x-x-STRAIGHT NOISE-x-x-".postln; ("--freq = " ++
freq).postln;});
            },
            { //else, modulating noise
                var mod, lo, hi;
                mod = rrand(0.1, 1.0).round(0.1);
                lo = rrand(0.25, 1.0).round(0.01);
                hi = rrand(1.25, 4.2).round(0.01);
                ~samp0 = Synth(\bufRead_noise1, [\buf, buffer, \duration, duration,
\modFreq, mod, \loFreq, lo, \hiFreq, hi, \amp, ampBus, \out, outBus]);
                if (post == 1, {"-x-x-MOD NOISE-x-x-".postln; ("mod = " ++ mod + " lo =" + lo
+ " hi =" + hi).postln;});
            }
        );
        duration;
    };

    ~trigRand_sineSineMod_bufPlay = { |buffer, duration, ampBus, outBus, shortDurThresh = 3, loopDurMin
= 6.0, loopDurMax = 9.0, post = 0| //----NEW
        var randNum, randCase, rand1, rand2, rand3, lfoFreq, loFreq, hiFreq;
        if (post == 1, {"--RAND sineSineMod TRIGGERED--".postln;});
        if (duration <= shortDurThresh,
            { //for short dur sounds
                if (post == 1, {"[sineSineMod] SHORT DUR DETECTED --".postln;});
                randNum = rrand(1, 3);
                randCase = case
                { randNum == 1 } {
                    if (post == 1, {"-[sineSineMod] SLOW tempo".postln;});
                    rand1 = rrand(0.8, 1.2).round(0.001);
                    rand2 = rrand(0.8, 1.2).round(0.001);

```

```

        rand3 = rrand(4.0, 6.0).round(0.001);
        lfoFreq = rand1; loFreq = rand2; hiFreq = rand3;
    }
    {randNum == 2} {
        if (post == 1, {"-[sineSineMod] MED tempo").postln;});
        rand1 = rrand(9.5, 12.0).round(0.001);
        rand2 = rrand(0.8, 1).round(0.001);
        rand3 = rrand(8.0, 10.0).round(0.001);

        if (0.8.coin == true, {
            if (post == 1, {"-[sineSineMod] LESS PHASING (lfoFreq
= hiFreq == rand1)").postln;});
            lfoFreq = rand1; loFreq = rand2; hiFreq = rand1; //more
loopable, cohesive, rhythmic figures
        }, {
            if (post == 1, {"-[sineSineMod] MORE PHASING
(lfoFreq != hiFreq)").postln;});
            lfoFreq = rand1; loFreq = rand2; hiFreq = rand3; //neat
phasing often
        });
    }
    {randNum == 3} {
        if (post == 1, {"-[sineSineMod] FAST tempo").postln;});
        rand1 = rrand(8.0, 11.0).round(0.001);
        rand2 = rrand(0.25, 1).round(0.001);
        rand3 = rrand(10.0, 15.0).round(0.001);
        lfoFreq = rand1; loFreq = rand2; hiFreq = rand3;
    };

    if (post == 1, {"-[sineSineMod] lfoFreq =" + lfoFreq.round(0.001) + "loFreq
=" + loFreq.round(0.001) + "hiFreq =" + hiFreq.round(0.001)).postln;});

    duration = rrand(loopDurMin, loopDurMax).round(0.01);
    if (post == 1, {"---[sineSineMod] LOOP dur =" + duration).postln;});

    //consider introducing slight chance to trigger 2 synths w/ variable offset
<-----

    Synth(\bufRead_sineSineMod_randPhase, [\buf, buffer, \duration, duration,
\lfoFreq, lfoFreq, \loFreq, loFreq, \hiFreq, hiFreq, \attTime, 0.02, \relTime, 0.02, \amp, ampBus, \out,
outBus]);

    if (post == 1, {"----[sineSineMod] TRIGGERED").postln;});

    duration; //return adjusted dur for enclosing routine
},
{//for med-long dur sounds
    if (post == 1, {"[sineSineMod] MED-LONG DUR DETECTED --").postln;});
    randNum = rrand(1.0, 10.0).round(0.01);
    randCase = case
    {randNum <= 7} { //70% of time
        if (post == 1, {"-[sineSineMod] RAND lfoFreq, loFreq, hiFreq
vals").postln;});

        lfoFreq = rrand(0.1, 0.5);
        loFreq = rrand(0.02, 0.04);
        hiFreq = rrand(0.11, 0.35);
    }
    {randNum > 7 && randNum <= 8.5} { //15% of time
        if (post == 1, {"-[sineSineMod] PRESET1 lfoFreq, loFreq, hiFreq
vals").postln;});

        lfoFreq = 0.1; loFreq = 0.02; hiFreq = 0.2;
    }
    {randNum > 8.5} { //15% of time
        if (post == 1, {"-[sineSineMod] PRESET2 lfoFreq, loFreq, hiFreq
vals").postln;});

        lfoFreq = 0.22; loFreq = 0.02; hiFreq = 0.3;
    }
}

```



```

    };

    if (post == 1, {"--[sineSineMod] lfoFreq =" + lfoFreq.round(0.001) + "loFreq
    =" + loFreq.round(0.001) + "hiFreq =" + hiFreq.round(0.001)}.postln;});

    Synth(\bufRead_sineSineMod_randPhase, \buf, buffer, \duration, duration,
    \lfoFreq, lfoFreq, \loFreq, loFreq, \hiFreq, hiFreq, \attTime, 0.5, \relTime, 1, \amp, ampBus, \out,
    outBus);

    if (post == 1, {"----[sineSineMod] TRIGGERED").postln;});

    duration;

    }

);

~trigRand_noiseSineMod_bufPlay = {lbuffer, duration, ampBus, outBus, shortDurThresh = 3,
loopDurMin = 6.0, loopDurMax = 9.0, post = 0! //<---NEW
var randNum, randCase, rand1, rand2, rand3, lfoFreq, loFreq, hiFreq;
if (post == 1, {"--RAND noiseSineMod TRIGGERED--".postln;});
if (duration <= shortDurThresh,
    //for short dur sounds
    if (post == 1, {"[noiseSineMod] SHORT DUR DETECTED --").postln;});
    randNum = rrand(1, 3);
    randCase = case
    {randNum == 1} {
        if (post == 1, {"--[noiseSineMod] SLOW tempo").postln;});
        rand1 = rrand(0.8, 1.2).round(0.001);
        rand2 = rrand(1, 1.2).round(0.001);
        rand3 = rrand(4.5, 8.0).round(0.001);
        lfoFreq = rand1; loFreq = rand2; hiFreq = rand3;
    }
    {randNum == 2} {
        if (post == 1, {"--[noiseSineMod] MED tempo").postln;});
        rand1 = rrand(9.5, 12.0).round(0.001);
        rand2 = rrand(0.8, 1).round(0.001);
        lfoFreq = rand1; loFreq = rand2; hiFreq = rand1; //more loopable,
cohesive, rhythmic figures //<---USE for bit of randomness, but similar to above
    }
    {randNum == 3} {
        if (post == 1, {"--[noiseSineMod] FAST tempo").postln;});
        rand1 = rrand(8.0, 10.0).round(0.001); //<---USE
        rand2 = rrand(0.25, 0.5).round(0.001); //<---USE
        rand3 = rrand(10.0, 13.0).round(0.001); //<---USE
        lfoFreq = rand1; loFreq = rand2; hiFreq = rand3;
    }
    };

    if (post == 1, {"--[noiseSineMod] lfoFreq =" + lfoFreq.round(0.001) + "loFreq
    =" + loFreq.round(0.001) + "hiFreq =" + hiFreq.round(0.001)}.postln;});

    duration = rrand(loopDurMin, loopDurMax).round(0.01);
    if (post == 1, {"---[noiseSineMod] LOOP dur =" + duration}.postln;});

    //consider introducing slight chance to trigger 2 synths w/ variable offset
<-----

    Synth(\bufRead_noiseSineMod_randPhase, \buf, buffer, \duration, duration,
    \lfoFreq, lfoFreq, \loFreq, loFreq, \hiFreq, hiFreq, \attTime, 0.2, \relTime, 0.4, \amp, ampBus, \out,
    outBus);

    if (post == 1, {"----[noiseSineMod] TRIGGERED").postln;});

    duration; //return adjusted dur for enclosing routine
},
//for med-long dur sounds
if (post == 1, {"[noiseSineMod] MED-LONG DUR DETECTED --").postln;});

```

```

        lfoFreq = rrand(0.1, 1).round(0.001);
        loFreq = rrand(0.03, 0.05).round(0.001);
        hiFreq = rrand(0.1, 0.25).round(0.001);

        if (post == 1, {"--[noiseSineMod] lfoFreq =" + lfoFreq.round(0.001) + "loFreq
=" + loFreq.round(0.001) + "hiFreq =" + hiFreq.round(0.001)}.postln;));

        Synth(\bufRead_noiseSineMod_randPhase, [\buf, buffer, \duration, duration,
\lfoFreq, lfoFreq, \loFreq, loFreq, \hiFreq, hiFreq, \attTime, 0.2, \relTime, 0.4, \amp, ampBus, \out,
outBus]);

        if (post == 1, {"----[noiseSineMod] TRIGGERED"}.postln;));

        duration;

    }

};

~trigRand_noise3Mod_bufPlay = {\buffer, duration, ampBus, outBus, post = 0| //works best w/ med-long
dur sounds only... //<---NEW
var lfoFreq, loFreq, hiFreq;
if (post == 1, {"--RAND noise3Mod--"}.postln;));
lfoFreq = rrand(0.01, 0.5).round(0.001);
loFreq = rrand(0.29, 0.56).round(0.001);
hiFreq = rrand(0.67, 1.1).round(0.001);
if (post == 1, {"--[noise3Mod] lfoFreq =" + lfoFreq.round(0.001) + "loFreq =" + loFreq.round(0.001) +
"hiFreq =" + hiFreq.round(0.001)}.postln;));

Synth(\bufRead_noise3_randPhase, [\buf, buffer, \duration, duration, \lfoFreq, lfoFreq, \loFreq,
loFreq, \hiFreq, hiFreq, \attTime, 0.2, \relTime, 0.4, \amp, ampBus, \out, outBus]);
if (post == 1, {"----[noise3Mod] TRIGGERED"}.postln;));
};

~trigRand_bufPlay_rateLFOsine = {\buffer, duration, ampBus, outBus, shortDurThresh = 2.5,
loopDurMin = 2.5, loopDurMax = 8.0, post = 0| //<-----NEW
var randNum, randCase, lfoRate, minRate, maxRate;
if (post == 1, {"--RAND rateLFOsine TRIGGERED--"}.postln;));
if (duration <= shortDurThresh,
    { //for short dur sounds
        if (post == 1, {"[rateLFOsine] SHORT DUR DETECTED --"}.postln;));
        randNum = rrand(1.0, 10.0).round(0.01);
        randCase = case
        {randNum <= 4} { //40%
            if (post == 1, {"-[rateLFOsine] WOBBLE - NARROW"}.postln;));
            lfoRate = rrand(4.0, 15.0).round(0.01);
            minRate = rrand(-14.0, -8.0).round(0.01);
            maxRate = rrand(0.0, 7.0).round(0.01);
        }
        {randNum >4 && randNum <= 8} { //40%
            if (post == 1, {"-[rateLFOsine] WOBBLE - WIDE"}.postln;));
            lfoRate = rrand(0.6, 1.2).round(0.01);
            minRate = rrand(-28.0, -20.0).round(0.01);
            maxRate = rrand(20.0, 26.0).round(0.01);
        }
        {randNum > 8} {
            if (post == 1, {"-[rateLFOsine] WOBBLE - space
invaders"}.postln;));
            lfoRate = rrand(8.0, 16).round(0.01);
            minRate = rrand(-38.0, -31.0).round(0.01);
            maxRate = rrand(-26.0, -20.0).round(0.01);
        }
    };

    if (post == 1, {"--[rateLFOsine] lfoRate =" + lfoRate + "minRate =" + minRate
+ "maxRate =" + maxRate}.postln;));

    duration = rrand(loopDurMin, loopDurMax).round(0.01);

```

```

        if (post == 1, {"--[rateLFOsine] LOOP dur =" + duration).postln;});

        //consider introducing slight chance to trigger 2 synths w/ variable offset
<-----

        minRate = minRate.midratio; maxRate = maxRate.midratio; //convert rates
to ratio floats from half steps

        if (0.3.coin == true, { //30% chance play two synths
            var offset = rrand(0.2, 1.8).round(0.01);
            Synth(\bufRead_rateLFOsine, [\buf, buffer, \amp, ampBus, \out,
outBus, \lfoRate, lfoRate, \minRate, minRate, \maxRate, maxRate, \duration, duration, \attTime, 0.02,
\relTime, 0.6]);
            {Synth(\bufRead_rateLFOsine, [\buf, buffer, \amp, ampBus, \out,
outBus, \lfoRate, lfoRate, \minRate, minRate, \maxRate, maxRate, \duration, duration, \attTime, 0.02,
\relTime, 0.6]}.defer(offset);
            duration = duration + offset + 0.01; //update duration to compensate
for offset
            if (post == 1, {"--[rateLFOsine] 2 SYNTHS TRIG! | offset =" + offset
+ " | newDur =" + duration).postln;});
            }, { //80% chance just play one synth
            if (post == 1, {"--[rateLFOsine] TRIGGERED").postln;});
            Synth(\bufRead_rateLFOsine, [\buf, buffer, \amp, ampBus, \out,
outBus, \lfoRate, lfoRate, \minRate, minRate, \maxRate, maxRate, \duration, duration, \attTime, 0.02,
\relTime, 0.6]);
            });

            duration; //return adjusted dur for enclosing routine
        },
        { //for med-long dur sounds
            if (post == 1, {"[rateLFOsine] MED-LONG DUR DETECTED --").postln;});

            lfoRate = rrand(0.5, 1.2).round(0.01);
            minRate = rrand(-16.0, -8.0).round(0.01);
            maxRate = rrand(5.0, 12.0).round(0.01);

            if (post == 1, {"--[rateLFOsine] lfoRate =" + lfoRate + "minRate =" + minRate
+ "maxRate =" + maxRate).postln;});

            Synth(\bufRead_rateLFOsine, [\buf, buffer, \amp, ampBus, \out, outBus,
\lfoRate, lfoRate, \minRate, minRate, \maxRate, maxRate, \duration, duration, \attTime, 0.02, \relTime,
0.6]);
            if (post == 1, {"--[rateLFOsine] TRIGGERED for:" + duration +
"seconds...").postln;});

            duration;
        }
    );
};

//SOUND SCENE AUDIFICATION FUNCTIONS -----

~sampBuf = { |filePath, fileNum, rate| //helper function creates audio buffer and env var storing its dur
    Buffer.read(s, (filePath ++ fileNum ++ ".wav").standardizePath, action: { |buffer| //load audio file
to buffer
        ("~buf" ++ fileNum ++ "dur =" ++ ((buffer.numFrames/buffer.sampleRate) /
rate)).interpret; //store dur in var
    });
};

~audifyScene1 = { |filePath, ampBus, outBus, fileStart = 0, fileNum = 10, rate = 36, pitch = -36, cycle =
1, overlap = 0.02, status| //<---NOT USING filePath arg?
    var audifyRoutine, thisRate, thisPitch;
    thisRate = rate.asFloat.round(0.01).midratio;
    thisPitch = pitch.asFloat.round(0.01).midratio;

```

```

audifyRoutine = Routine({
    status.envirPut(true); //<---NEW

    for (fileStart, (fileNum - 1), {lil
        ("~buf" ++ i ++ "= ~sampBuf.(~fileAudio_writeIn_lastPath," ++ i ++ "," ++
thisRate ++ ")").interpret;
    });

    //could sum all buffer durations to predict when audification will be complete...

    s.sync; //<----CONSIDER CONDITION.HANG INSTEAD OF S.SYNC, but since
waiting on buffers, maybe fine?

    cycle.do({ //trigger synths (one per sample played)
        for (fileStart, (fileNum - 1), {lil
            var thisBuf, thisDur;
            thisBuf = ("~buf" ++ i).interpret;
            thisDur = ("~buf" ++ i ++ "dur").interpret;
            Synth(\bufRead_complex1, [\buf, thisBuf, \amp, ampBus, \out,
outBus, \rate, thisRate, \pitch, thisPitch, \winSize, 0.35, \timeDisp, 0, \pShiftMix, 1, \duration, thisDur]);
            //thisDur.yield;
            (thisDur - overlap).yield; //slight overlap between samples...
        });
    });

    status.envirPut(false); //<---NEW

    0.2.yield; //slight yield to give enough time to read last buffer and avoid Buffer.read
error

    for (fileStart, (fileNum - 1), {lil //free all buffers
        ("~buf" ++ i ++ ".free").interpret;
    });

    audifyRoutine;//return routine
};

```

//-----AUTO ROOM CONVOLVE FUNCS / ROUTINES -----

```

//~autoConv_bufBasic(impulse: 2, postDel: 3, amp: (-10.dbamp));
~autoConv_bufBasic = {active = 1, impulse = 0, sweepDur = 0.35, amp = (-14.dbamp), postDel = 3l
//not being used, first prototype
    if (active == 1,
        {
            if (~bufRecorder.isRunning,
                {
                    var impCase;
                    //START ~monitor_loudness and ~autoLevel routines...
                    "=====[INIT bufBasic AUTO CONVOLVE]

                    ~recPlayDelayOffset.(offset: 0);
                    impCase = case
                    {impulse == 0} {
                        Synth(\impulseWhite3, [\shelf_freq, 10000, \rs,
1, \shelf_db, -6, \peak_freq, 160, \peakQ, 12, \peak_db, 12, \attackTime, 0.001, \releaseTime, 0.13,
\curve, -4, \amp, amp, \out,
~outMainBus]).moveBefore(~bufRecorder).onFree({~button_playBuf.valueAction_(1)}.defer(postDel));

```

```

    }
    {impulse == 1}{
        Synth(\sinSweep1, [\freq1, 40, \freq2, 20000,
\dur, sweepDur, \amp, amp, \out,
~outMainBus]).moveBefore(~bufRecorder).onFree({{~button_playBuf.valueAction_(1)}.defer(postDel)});
    }
    {impulse == 2}{
        Synth(\sawSweep1, [\freq1, 40, \freq2, 20000,
\dur, sweepDur, \amp, amp, \out,
~outMainBus]).moveBefore(~bufRecorder).onFree({{~button_playBuf.valueAction_(1)}.defer(postDel)});
    };
    },
    {
        "===CAN'T RUN AUTOCONV UNLESS BUF RECORDER
ACTIVE===".postln;
    }
    );
    },
    {
        //STOP ~monitor_loudness and ~autoLevel routines...
    }
    };

~autoConvolve1A = {Iactive, featureBus, outBus, ampBus, filePath, loudThresh = 5, pitchThresh = 0.55,
rout1, rout2, synth1, initFlag, repeatCounter, impulse = 0, freq1 = 100, freq2 = 2000, attTime = 5,
susTime = 3, relTime = 6, impPostDel = 0, preRecPad = 0.03, postRecPad = 0.02, timeStretch = false,
timeStretchRate = 0.9, rate = 0.1, post = 0I
    if (active == 1,
        {
            if (~bufRecorder.isRunning,
                {
                    var rout;

                    initFlag.envirPut(1); //set init state flag
                    repeatCounter.envirPut(0); //reset sample playback

iteration counter

                    rout = Routine({
                        if (initFlag.envirGet == 1,
                            {
                                var impCase;
                                if (post == 1, {"===== [INIT
AUTO CONVOLVE] =====".postln;});

                                initFlag.envirPut(0); //move to
                                post-init sequence of routine after next .yieldAndReset

                                if (post == 1, {"...short pause
                                0.4.yield; //slight pause for

                                impCase = case
                                {impulse == 0} {//SUSTAINED
WHITE NOISE IMPULSE

                                ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous; // <-- log sound start
frame

                                synth1.envirPut(Synth(\susWhite1, [\attTime, attTime, \susTime, susTime, \relTime, relTime,
                                \amp, ampBus, \out, outBus]).moveBefore(~bufRecorder));

                                NodeWatcher.register(synth1.envirGet); //<---NEW

```

```

true;
= 1;}.defer; //update GUI
susWhite1".postln;});
relTime + impPostDel).yieldAndReset;

{impulse == 0.1}

{impulse == 0.1}

//TRANSIENT WHITE NOISE PING IMPULSE
frame ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous; // <-- log sound start

synth1.envirPut(Synth(\impulseWhite3, [\shelf_freq, 10000, \rs, 1, \shelf_db, -6, \peak_freq,
160, \peakQ, 12, \peak_db, 12, \attackTime, 0.001, \releaseTime, 0.13, \curve, -4, \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder));

NodeWatcher.register(synth1.envirGet); //<---NEW

true;
= 1;}.defer; //update GUI
impulseWhite3".postln;});
impPostDel).yieldAndReset;

{impulse == 1}{ //SINEWAVE

CHIRP / SWEEP IMPULSE
frame ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous; // <-- log sound start

synth1.envirPut(Synth(\sinSweep1, [\freq1, freq1, \freq2, freq2, \dur, susTime, \amp,
ampBus, \out, outBus]).moveBefore(~bufRecorder));

NodeWatcher.register(synth1.envirGet); //<---NEW

true;
= 1;}.defer; //update GUI
sinSweep1 | sweepDur =" + susTime).postln;});
impPostDel).yieldAndReset;

{impulse == 1.1}{ //RANDOM

SINEWAVE SWELL IMPULSE
frame ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous; // <-- log sound start

synth1.envirPut(Synth(\randSinSwell_1, [\attTime, attTime, \susTime, susTime, \relTime,
relTime, \loFreq, freq1, \hiFreq, freq2, \amp, ampBus, \out, outBus]).moveBefore(~bufRecorder));

NodeWatcher.register(synth1.envirGet); //<---NEW

true;
= 1;}.defer; //update GUI
randSines".postln;});

```

```

relTime + impPostDel).yieldAndReset;
                                                                    (attTime + susTime +
                                                                    }
                                                                    {impulse == 1.2} { //DUAL
SINES - BINAURAL BEATS
    ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous; // <-- log sound start
frame
    synth1.envirPut(Synth\sineBeats, [freq1, freq1, \freq2, freq2, \attTime, attTime, \susTime,
susTime, \relTime, relTime, \limitThresh, 0.9, \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder));
    NodeWatcher.register(synth1.envirGet);
                                                                    ~recSamp_input =
true;                                                                    {~button_recln.value
= 1;}.defer; //update GUI                                                                    if (post == 1, {"TRIG:
sineBeats".postln;});                                                                    (attTime + susTime +
relTime + impPostDel).yieldAndReset;                                                                    }
                                                                    {impulse == 2} {
    ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous; // <-- log sound start
frame
    synth1.envirPut(Synth\sawSweep1, [freq1, freq1, \freq2, freq2, \dur, susTime, \amp, ampBus,
\out, outBus]).moveBefore(~bufRecorder));
    NodeWatcher.register(synth1.envirGet); //<---NEW
                                                                    ~recSamp_input =
true;                                                                    {~button_recln.value
= 1;}.defer; //update GUI                                                                    if (post == 1, {"TRIG:
sawSweep1 | sweepDur =" + susTime).postln;});                                                                    (susTime +
impPostDel).yieldAndReset;                                                                    }
                                                                    {impulse == 3} { //CREATE
IMPULSE FROM DETECTED LIVE SOUND INPUT                                                                    if (post == 1, {"TRIG:
liveSound.....").postln;});                                                                    //START ~autoRec
                                                                    rate.yieldAndReset;
                                                                    };
                                                                    },
                                                                    { //else if (initFlag.envirGet == 0)
                                                                    var impCase, cond =
Condition.new;
                                                                    //repeating monitoring,
recording, playback                                                                    if (post == 1, {"[AUTO
CONVOLVE] -----".postln;});                                                                    impCase = case
                                                                    {impulse >= 0 && impulse < 3}
                                                                    if (~recSamp_input ==
                                                                    {
{ //<----- IF ANY SYNTH IR USED --- v v v v v v -----
true,
                                                                    {

```

```

(post == 1, {"->recSamp_input = true.....").postln;});
                                                                    if
                                                                    if
(~playSamp == true, { //b/c of implementation, we don't re-enter this part of routine until sample
  playback has finished (post sampleDur.yieldAndReset)
    ~playSamp = false; //<---not being used at the moment, but could be useful?
    {~button_playSamp.value = 0;}.defer; //update GUI
    ~audioBuf_play0.free; //<---cleanup after reset from prev part of routine
    if (post == 1, {"-->~playSamp = true --- cleaning up...").postln;});
                                                                    });
                                                                    //list
                                                                    // or
en_for_sound routine tests for silence in room from ~inputSendBus
should I just test for rms? <-----MAYBE JUST TEST RMS <-----
    featureBus.get({|vall //get featBus - 4 features: [Loudness, Onset, Freq, Pitch Measure]
      if (( (val[0] >= loudThresh) || (val[1] == 1) || (val[3] >= pitchThresh) ),
        { //if thresh exceeded, ~hearSound = true
          if (~hearSound != true, {~hearSound = true;});
          if (post == 1, {
            (">SOUND DETECTED...").postln;
            (">>Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3]
++ "]" ).postln;
          });
        },
        { //if thresh not exceeded, ~hearSound = false
          if (~hearSound != false, {~hearSound = false;});
          if (post == 1, {
            (">NO sound detected...").postln;
            (">>Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3]
++ "]" ).postln;
          });
        }
      );
    cond.unhang;
                                                                    });
    cond.hang;
                                                                    if
(~hearSound == true, //wait for sound to end before writing sample

```



```

    {
        if (post == 1, {"-->Post-hang: ~hearSound = true | ...waiting for sound to
end...").postln;});

        rate.yieldAndReset;

    },

    {//else if ~hearSound == false

        var currentRecFrame = ~phaseBusMonRec.getSynchronous;

        if (post == 1, {"-->Post-hang: ~hearSound = false | ...sound stopped...write
sample").postln;});

        ~recSamp_input_stopFrame = (currentRecFrame + (postRecPad *
s.sampleRate)).round(1);

        if (~recSamp_input_stopFrame > (~audioBuf0.numFrames - 1), {

~audioBuf0.numFrames; ~recSamp_input_stopFrame = ~recSamp_input_stopFrame -

        });

        /*if (~playSamp == true, //consider as a way to thoroughly test for end of reverb tail
possibly caused by impulse responses

            {//if we just finished playing a sample

                ~playSamp = false;

                intraSoundTestDur.yieldAndReset;

            },

            {//

            }

        ); */

        ~writeSampFromBufCopy_input.(); //sets GUI and ~recSamp_input = false (ORIG)

        //~writeSampFromBuf_name.(buffer: ~audioBuf0, in: ~recSamp_input_startFrame,
out: ~recSamp_input_stopFrame, inputSec: 0, filePath: filePath); //<---NEW (custom file naming) NOT
WORKING 100%, MESSSES UP CYCLING OF ROUTINE <-----

        if (post == 1, {"-->slight yield after sampleWrite...").postln;});

        0.2.yield; //<--- slight yield between sampleWrite and playback??

        if (post == 1, {"-->yield complete...").postln;});

        rate.yieldAndReset; //not sure how I feel about doing this w/out a check to confirm
sound has stopped (as in orig ~autoRec_soundTrig routine), but this is simpler and may work for now

    }

};

```

```

    },
    { //if
~recSamp_input == false
var
playDur, currentPlayCount;
if
(post == 1, {"->recSamp_input = FALSE.....").postln;});
~playSamp = true; //not used other than to update GUI, maybe more use though?

T ~recSamp_input_startFrame //SE
~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;
~recSamp_input = true;
{~button_recIn.value = 1;}.defer; //update GUI

AY BACK PREVIOUS RECORDED SAMPLE //PL
currentPlayCount = (repeatCounter.envirGet) + 1; //update repeat counter variable
repeatCounter.envirPut(currentPlayCount);
~audioBuf_play0 = Buffer.read(s, ~fileAudio_writeIn_lastFile, action: {!buffer!
playDur = buffer.numFrames/buffer.sampleRate;

if (timeStretch == true,
{
var pitchOffset;
pitch
playDur = (playDur/timeStretchRate).round(0.001);
pitchOffset = timeStretchRate.reciprocal; //inverse of rate to preserve orig

synth1.envirPut(Synth(\bufRead_complex1, [\buf, buffer, \amp, ampBus,
\out, outBus, \rate, timeStretchRate, \pitch, pitchOffset, \winSize, 0.1, \timeDisp, 0.003, \pShiftMix, 1,
\duration, playDur])); //play time-stretched sample
NodeWatcher.register(synth1.envirGet); //<---NEW
if (post == 1, {"-->PLAY LAST SAMP I TIME STRETCH =" +
timeStretchRate).postln;});
},
{//else if no timeStretching
playDur = playDur.round(0.001);
synth1.envirPut(Synth(\bufRead_noLoop, [\buf, buffer, \amp, ampBus, \out,
outBus])); //play unaltered sample
NodeWatcher.register(synth1.envirGet); //<---NEW
if (post == 1, {"-->PLAY LAST SAMP I no time stretch...").postln;});
}

```

```

    );

    {~button_playSamp.value = 1;}.defer; //update GUI
    (post == 1, {"--->Pre-sync samp Dur = " ++ playDur).postln;});

    s.sync;

    (post == 1, {"--->Post-sync samp Dur = " ++ playDur).postln;});

    playDur.yieldAndReset; //leave this side of routine after reset b/c ~recSamp_input = true
    }
    };
    {impulse == 3} { //<-----
LIVE SOUND IMPULSE ----- !!!!!

    rout2.envirPut(~autoRecPlay_conv0.(featureBus: featureBus, outBus: outBus, ampBus:
ampBus, synth1: synth1, repeatCounter: repeatCounter, loudThresh: loudThresh, pitchThresh:
pitchThresh, preRecPad: preRecPad, postRecPad: postRecPad, impPostDel: impPostDel, timeStretch:
timeStretch, timeStretchRate: timeStretchRate, debounceDur: 0, post: post));

    SystemClock.play(rout2.envirGet); //launch autoRecPlay_conv0 routine
    };
    );
    rout1.envirPut(rout);
    SystemClock.play(rout1.envirGet);
    },
    {
    "===CAN'T RUN AUTOCONV UNLESS BUF RECORDER
and ~listen_sound ACTIVE===".postln;
    }
    );
    },
    { //else if active == 0 (disable)
    rout1.envirGet.stop;
    if (rout2 != nil, {rout2.envirGet.stop;});
    if (synth1.envirGet.isRunning == true, {synth1.envirGet.free;}); //<---NEW
    if (~recSamp_input == true, {
    ~recSamp_input = false;
    {~button_recln.value = 0;}.defer; //update GUI
    });
    if (~playSamp == true, {
    ~playSamp = false; //<---not being used at the moment, but could
be useful?
    {~button_playSamp.value = 0;}.defer; //update GUI
    ~audioBuf_play0.free;
    });
    }
    );
    };

    ~autoRecPlay_conv0 = {featureBus, outBus, ampBus, synth1, repeatCounter, loudThresh = 5,
pitchThresh = 0.55, preRecPad = 0.02, postRecPad = 0.02, impPostDel = 0, rate = 0.1, timeStretch =
false, timeStretchRate = 0.9, debounceDur = 0.5, post = 0|
    ~autoRecPlay_state = 0; //<---reset this before restarting...
    Routine({
    if (~autoRecPlay_state == 0,

```

```

{
    if (~recSamp_input == true,
        {
            var currentTime =
Main.elapsedTime.round(0.001);
            if (currentTime <= ~time_minRec, //<----
                {
                    if (post == 1, {
                        ("-----").postln;

                        ("RECORDING.....").postln;

                        currentTime + "< ~time_minRec:" + ~time_minRec).postln;

                        ("currentTime:" +
                    });
                },
                {
                    featureBus.get({lval //get sum
                        if ((val[0] >=

bus "lop" - 4 features: [Loudness, Onset, Freq, Pitch Measure]
loudThresh) || (val[1] == 1) || (val[3] >= pitchThresh) ),

                    {//=====SOUND DETECTED=====
                        ~hearSound = true;

//C
                }
                if
                (post == 1, {
                    ("-----").postln;
                    ("**Recording... SOUND DETECTED").postln;
                    ("Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3] ++ "]" ).postln;
                });
                if
                (~playSamp == true, { //if we previously played sample, update ~time_recTimeOut
                    ~playSamp = false;
                    ~time_recTimeOut = currentTime + 15;
                });
                //ant
                if
                (i-troll / runaway recording measure... <---- KEEP HERE???
                    (currentTime >= ~time_recTimeOut, //terminate recording if we've exceeded recTimeOut
                        {
                            if (post == 1, {"===MAX RECORD DUR EXCEEDED - STOP
RECORDING===").postln;});
                            ~recSamp_input_stopFrame = ~lastRecPhasePos; //<---or ~busMonRec.getSyn???
                            ~writeSampFromBufCopy_input.(); //sets GUI and ~recSamp_input = false
                            ~time_lastRec_stop = currentTime; //<---
                            //~hearSound = false; <---commented out b/c we have been hearing a sound....
                            ~autoRecPlay_state = 1; //advance state to trigger ~autoPlay before next ~autoRec
                        }
                    )
                )
            )
        }
    )
}

```

```

    }
    },
    );

    { //=====IF NO SOUND DETECTED...=====
    if
    (~hearSound == false,
    {
        if (currentTime >= ~time_listenTimeOut, //<---
            { //Sound Stop Confirmed - WRITE AUDIO SAMPLE (sound was stopped
since last iteration) --- ALSO BEGIN PREV AUDIO SAMPLE PLAY BACK -----
                if (post == 1, {
                    ("-----+++++++-----").postln;
                    ("**Recording... SOUND STOP CONFIRMED").postln;
                    ("Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " +
+ val[3] ++ "]" ).postln;
                });

                //watch out if falling through from playingSamp below, might not be
setting ~recSamp_input_stopFrame <-----
                if (~playSamp == true, { //might/always be falling through? <-----
NEW
                    ~recSamp_input_stopFrame =
~phaseBusMonRec.getSynchronous; //<---NEW
                    ~writeSampFromBufCopy_input.value(); //sets GUI and
~recSamp_input = false
                },
                { //else if ~playSamp == false (if we haven't played sample back yet)
                    ~writeSampFromBufCopy_input.value(); //sets GUI and
~recSamp_input = false
                });

                if (post == 1, {
                    ("~autoRecPlay_state = 1").postln;
                });

                ~autoRecPlay_state = 1; //change state to trigger playback

                //-----older plan-----, still relevant for ~autoRec_soundTrigAuto
                //Analyze written audio file for features (avg loudness, pitchiness,
MFCCs, etc)

```

```

//Add sound file's extracted feature summary to a ~soundMemory
buffer for future analysis and behavior triggering based on past history of sound events...

        },
        { //if currentTime < ~time_listenTimeOut....sound may not have stopped
yet....
            if (post == 1, {
                ("----").postln;
                ("**Testing for sound stop...** - Current Time:" +
currentTime).postln;
                ("---Terminate Test at:" + ~time_listenTimeOut).postln;
            });
        }
    );
},
{ //else if ~hearSound == true (test to make sure sound has stopped but continue recording)
    //~time_listenTimeOut = ~listenDur_intraSound + Main.elapsedTime.round(0.001);
    ~time_listenTimeOut = ~listenDur_intraSound + currentTime; //<---make
~listenDur_intraSound into a local variable for this func routine....
    if (post == 1, {
        ("-----").postln;
        ("**Recording...SOUND STOP? ... Testing until: " +
~time_listenTimeOut).postln;
        ("Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3] ++
"]").postln;
    });

    //GET recHead phasor position and log "STOP SAMPLE TIME"
    ~phaseBusMonRec.get({lval //<---switch to .getSynchronous ----- ???
~recSamp_input_stopFrame = (val + (postRecPad *
s.sampleRate)).round(1);
    if (~recSamp_input_stopFrame > (~audioBuf0.numFrames - 1), { //safe
driving
        ~recSamp_input_stopFrame = ~recSamp_input_stopFrame -
~audioBuf0.numFrames; //wrap around buffer and log sound stop frame
    });
    if (post == 1, { ("--Sound Stop (in get) = " ++ (~recSamp_input_stopFrame /
s.sampleRate)).postln; });
    //~time_lastRec_stop = Main.elapsedTime.round(0.001); //<--- NEW - log
sound rec stop time

```



```

(**HEAR SOUND**").postln;
("Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3] ++
"]").postln;

});

if (~hearSound == false, //TEST SOUND START...
{
~phaseBusMonRec.get({lval //log recHead pos //<-----ORIG
~recSamp_input_startFrame = (val - (preRecPad *
s.sampleRate)).round(1);
if (~recSamp_input_startFrame < 0, { //if startFrame is
negative...
~recSamp_input_startFrame =
~audioBuf0.numFrames + ~recSamp_input_startFrame; //wrap around buffer and log sound start frame
});

if (post == 1, {
("-----").postln;
("--> Sound START?: " ++
(~recSamp_input_startFrame/s.sampleRate)).postln;
});

~hearSound = true; //<--- moved inside get
});
},
{//else ~hearSound == true
var currentTime = Main.elapsedTime.round(0.001);
~time_minRec = currentTime + ~minRecDur; //minimum time to
stop recording
~time_lastRec_start = currentTime - rate; //<---NEW - subtract rate
b/c sound detected in previous cycle of routine
~time_recTimeOut = ~time_lastRec_start + ~maxRecDur; //<---NEW
- prevent runaway recordings and trolls
if (post == 1, {
(" - - - ").postln;
("--> Sound START CONFIRMED:" +
(~recSamp_input_startFrame/s.sampleRate)).postln;
("--REC Sample Start").postln;
});
});

```



```

    }
    );
    }); */
}

);

}

});

//commenting out condition
check for ~autoRecPlay_state that prevents simultaneous rec/play
/*},
{ //else if (~playSamp
//("----Can't
== true)
record when playSamp == true || autoRecPlay_state != 0").println;
0.25.yieldAndReset;

}); */
}

);
}

);
},
{ //else if ~autoRecPlay_state = 1 =====PLAYBACK =====
var playDur, bufFrames, currentTime, currentPlayCount;
//-----v-v-v-v-v-v-v-NEW -v-v-v-v-v-v-----

impPostDel.yield; //<----slight pause if desired between recording
and playing back sounds

//SET ~recSamp_input_startFrame
~recSamp_input_startFrame =
~phaseBusMonRec.getSynchronous; //keeps rec/play phase locked, but causes excessive build-up of
silence at beginning of time-stretched samples
~recSamp_input = true; //<---helps keep rec/play phase locked
{~button_recln.value = 1;}.defer; //update GUI

/*
//---v-v--integrate this to avoid multiple samples playing back after
lots of time stretching??---v---v--
if (~playSamp == true, //if we're already playing a sample but
attempt to play another (maybe b/c ~autoRec doesn't hear a quiet / very time-stretched sample w/ large
silent gap...), then deal w/ it

{
//quickly fade out Sample Playback bus
//free ongoing playBack synth
//free active buffer....
//quickly fade in Sample Playback bus

//1.yield;

}

);
*/

//=====PLAY PREV RECORDED SAMPLE!!!
~playSamp = true;

```

```

counter variable      currentPlayCount = (repeatCounter.envirGet) + 1; //update repeat
                      repeatCounter.envirPut(currentPlayCount);

action: {!buffer!    ~audioBuf_play0 = Buffer.read(s, ~fileAudio_writeln_lastFile,
                      bufFrames = buffer.numFrames;
                      playDur = bufFrames/buffer.sampleRate;
                      if (timeStretch == true,
                          {
                              var pitchOffset, durCase, attack,
release;
                              if (post == 1, {"~~~TIME STRETCH:
orig playDur =" + playDur + "! rate =" + timeStretchRate).postln;});
                              playDur =
(playDur/timeStretchRate).round(0.001);
                              if (post == 1, {"~~~TIME STRETCH:
stretched playDur =" + playDur + "! rate =" + timeStretchRate).postln;});
                              if (post == 1, {"---currentPlayCount =" +
currentPlayCount + "!").post;});

                              durCase = case
                              {playDur <= 6} {
                                  if (currentPlayCount <= 3,
{attack = 0.02; release = 0.02;}, {attack = 0.02; release = 0.1;});
                                  }
                              {playDur > 6 && playDur <= 20} {
                                  var countCase;
                                  countCase = case
                                  {currentPlayCount <= 3} {attack
= 0.02; release = 0.2;}
                                  {currentPlayCount > 3 &&
currentPlayCount <= 6} {attack = 0.04; release = 0.4;} //message
                                  {currentPlayCount > 6} {attack
= 0.3; release = 0.4;} //message
                                  //if (currentPlayCount <= 3,
{attack = 0.02; release = 0.2;}, {attack = 0.5; release = 0.4;});
                                  }
                              {playDur > 20} {
                                  var countCase;
                                  countCase = case
                                  {currentPlayCount <= 3} {attack
= 0.02; release = 0.2;}
                                  {currentPlayCount > 3 &&
currentPlayCount <= 6} {attack = 0.5; release = 1.1;}
                                  {currentPlayCount > 6} {attack
= 1.3; release = 3;}
                                  //if (currentPlayCount <= 3,
{attack = 0.02; release = 0.2;}, {attack = 1.3; release = 3;});
                                  };
                              if (post == 1, {"---attack =" + attack + "!
release =" + release).postln;});
                              /*{playDur <= 6} {attack = 0.02; release
= 0.02;}
                              {playDur > 6 && playDur <= 20} {attack
= 0.02; release = 2.5;}
                              {playDur > 20} {attack = 0.02; release =
3.7;}; */

```

```

= 0.02;} //orig
= 1; release = 2.5;} //orig
3.7;}; */ //orig

/*{playDur <= 6} {attack = 0.02; release
{playDur > 6 && playDur <= 20} {attack
{playDur > 20} {attack = 1.3; release =

if (playDur >= ~audioBuf0_dur, {
    playDur = ~audioBuf0_dur -
    if (post == 1, {"~~~TIME
    STRETCH: --BUF LENGTH EXCEEDED I newDur =" + playDur).postln;});
    }); //avoid creating a playDur longer than
the total audio buffer (crude for now, will figure out something better to do later.....)

pitchOffset = timeStretchRate.reciprocal;
//inverse of rate to preserve orig pitch
//Synth(\bufRead_complex1, [\buf,
buffer, \amp, ampBus, \out, outBus, \rate, timeStretchRate, \pitch, pitchOffset, \winSize, 0.1, \timeDisp,
0.003, \pShiftMix, 1, \duration, playDur]); //play time-stretched sample

synth1.envirPut(Synth(\bufRead_complex1, [\buf, buffer, \amp, ampBus, \out, outBus, \rate,
timeStretchRate, \pitch, pitchOffset, \winSize, 0.1, \timeDisp, 0.003, \pShiftMix, 1, \duration, playDur,
\attack, attack, \release, release]).onFree({~button_playSamp.value = 0}.defer;)); //THIS instead
<-----?

NodeWatcher.register(synth1.envirGet); //<---NEW
TIME STRETCH =" + timeStretchRate).postln;});
    },
    //else if no timeStretching
    playDur = playDur.round(0.001);
    //Synth(\bufRead_noLoop, [\buf,
buffer, \amp, ampBus, \out, outBus]); //play unaltered sample

synth1.envirPut(Synth(\bufRead_noLoop, [\buf, buffer, \amp, ampBus, \out,
outBus]).onFree({~button_playSamp.value = 0}.defer;));

NodeWatcher.register(synth1.envirGet); //<---NEW
no time stretch...").postln;});
    if (post == 1, {"-->PLAY LAST SAMP I
    }
    );
    });
    {~button_playSamp.value = 1;}.defer; //update GUI
    if (post == 1, {"-->Pre-sync samp Dur = " ++ playDur).postln;});
    s.sync;
    if (post == 1, {"-->Post-sync samp Dur = " ++ playDur).postln;});

    //~time_minRec = currentTime + ~minRecDur; //minimum time to
stop recording (DON'T THINK I need this here in this version....)
    //~time_lastRec_start = currentTime - rate; //<---NEW - subtract rate
b/c sound detected in previous cycle of routine (DON'T THINK I need this here in this version....)
    //~time_recTimeOut = ~time_lastRec_start + ~maxRecDur; //<--
NEW - prevent runaway recordings and trolls (DON'T THINK I need this here in this version....)

    //~time_listenTimeOut = ~listenDur_intraSound + currentTime;
//<----this if 0.1.yieldAndReset; //<--

    /*if (buffFrames > ~audioBuf0.numFrames, { //WHAT SHOULD I DO
ONCE SAMPLED AUDIO IS LONGER OR APPROACHING TOTAL LENGTH OF MAIN
BUFFER??????? <-----

```

```

    }); */
    ~autoRecPlay_state = 0;
    currentTime = Main.elapsedTime.round(0.001);
    ~time_listenTimeOut = currentTime + playDur; //<----use if sticking
w/ below playDur.yieldAndReset <-
    playDur.yieldAndReset; //this or rate.yieldAndReset, latter maybe
more interesting b/c it's only relying on the sound detection (ML) alg....but this might risk killing the
process?
    }
    );
    rate.asFloat.yieldAndReset;
});
};

~autoConvolve2_repLim_voxels = {lactive, featureBus, outBus, ampBus, filePath, loudThresh = 5,
pitchThresh = 0.55, rout1, rout2, synth1, initFlag, repeatCounter, repeatLimit = 8, impulse = 0, freq1 =
100, freq2 = 2000, attTime = 5, susTime = 3, relTime = 6, impPostDel = 0, preRecPad = 0.03,
postRecPad = 0.02, timeStretch = false, timeStretchRate = 0.9, rate = 0.1, post = 0}
    if (active == 1,
        {
            if (~bufRecorder.isRunning,
                {
                    var rout;

                    initFlag.envirPut(1); //set init state flag
                    repeatCounter.envirPut(0); //reset sample playback

iteration counter

                    rout = Routine({
                        if (initFlag.envirGet == 1,
                            {
                                var impCase;
                                if (post == 1, {"=====[INIT

AUTO CONVOLVE]====="}.postln;});

                                initFlag.envirPut(0); //move to

                                post-init sequence of routine after next .yieldAndReset

                                //if (post == 1, {"...short pause

                                for debounce..."}.postln;});

                                //0.4.yield; //slight pause for

                                button debounce...?

                                impCase = case
                                {impulse == 0} {//SUSTAINED

WHITE NOISE IMPULSE

                                ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous; // <-- log sound start
frame

                                synth1.envirPut(Synth(\susWhite1, [attTime, attTime, \susTime, susTime, \relTime, relTime,
\amp, ampBus, \out, outBus]).moveBefore(~bufRecorder));

                                NodeWatcher.register(synth1.envirGet); //<---NEW

                                ~recSamp_input =

                                true;

                                {~button_recln.value

                                = 1;}.defer; //update GUI

                                if (post == 1, {"TRIG:

                                susWhite1".postln;});

                                (attTime + susTime +

                                relTime + impPostDel).yieldAndReset;

                                }
}

```

```

{impulse == 0.1}

//TRANSIENT WHITE NOISE PING IMPULSE

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

synth1.envirPut(Synth(impulseWhite3, [shelf_freq, 10000, \rs, 1, \shelf_db, -6, \peak_freq,
160, \peakQ, 12, \peak_db, 12, \attackTime, 0.001, \releaseTime, 0.13, \curve, -4, \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder));

NodeWatcher.register(synth1.envirGet); //<---NEW

~recSamp_input =
true;
{~button_recln.value
= 1;}.defer; //update GUI
impulseWhite3".postln;});
if (post == 1, {"TRIG:
(0.14 +
impPostDel).yieldAndReset;
}
{impulse == 1}{ //SINEWAVE

CHIRP / SWEEP IMPULSE

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

synth1.envirPut(Synth(\sinSweep1, [freq1, freq1, \freq2, freq2, \dur, susTime, \amp,
ampBus, \out, outBus]).moveBefore(~bufRecorder));

NodeWatcher.register(synth1.envirGet); //<---NEW

~recSamp_input =
true;
{~button_recln.value
= 1;}.defer; //update GUI
sinSweep1 | sweepDur =" + susTime).postln;});
if (post == 1, {"TRIG:
(susTime +
impPostDel).yieldAndReset;
}
{impulse == 1.1}{ //RANDOM

SINEWAVE SWELL IMPULSE

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

//synth1.envirPut(Synt
h(\randSinSwell_1, [attTime, attTime, \susTime, susTime, \relTime, relTime, \loFreq, freq1, \hiFreq,
freq2, \amp, ampBus, \out, outBus]).moveBefore(~bufRecorder)); //<--orig

synth1.envirPut(Synth(\randSinSwell_lim_1, [attTime, attTime, \susTime, susTime, \relTime,
relTime, \loFreq, freq1, \hiFreq, freq2, \limitThresh, (-1.dbamp), \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder)); //<---NEW

NodeWatcher.register(synth1.envirGet); //<---NEW

~recSamp_input =
true;
{~button_recln.value
= 1;}.defer; //update GUI
randSines".postln;});
if (post == 1, {"TRIG:
(attTime + susTime +
relTime + impPostDel).yieldAndReset;
}
{impulse == 1.2}{ //DUAL

SINES - BINAURAL BEATS

```

```

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

synth1.envirPut(Synth(\impulseWhite3, [\shelf_freq, 10000, \rs, 1, \shelf_db, -6, \peak_freq,
160, \peakQ, 12, \peak_db, 12, \attackTime, 0.001, \releaseTime, 0.13, \curve, -4, \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder));

```

```

NodeWatcher.register(synth1.envirGet); //<---NEW
true;
= 1;};defer; //update GUI
impulseWhite3".postln;});
impPostDel).yieldAndReset;

```

CHIRP / SWEEP IMPULSE

```

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <- log sound start
frame

synth1.envirPut(Synth(\sinSweep1, [freq1, freq1, \freq2, freq2, \dur, susTime, \amp,
ampBus, \out, outBus]).moveBefore(~bufRecorder));

```

```
NodeWatcher.register(synth1.envirGet); //<---NEW
true;
= 1;}.defer; //update GUI
sinSweep1 | sweepDur =" + susTime).postIn;});
impPostDel).yieldAndReset;
```

SINEWAVE SWELL IMPULSE

```

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

//synth1.envirPut(Synt
h(\randSinSwell_1, \attTime, attTime, \susTime, susTime, \relTime, relTime, \oFreq, freq1, \hiFreq,
freq2, \amp. ampBus, \out. outBus)).moveBefore(~bufRecorder)); // <-- orig

```

```
synth1.envirPut(Synth(\vandSinSwell_lim_1, [\attTime, attTime, \susTime, susTime, \relTime,
relTime, \loFreq, freq1, \hiFreq, freq2, \limitThresh, (-1.dbamp), \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder)); //<--NEW
```

```

NodeWatcher.register(synth1.envirGet); //<---NEW
true;
= 1;}.defer; //update GUI
randSines".postIn;});
relTime + impPostDel).yieldAndReset;
~recSamp_input =
{~button_recln.value
if (post == 1, {"TRIG:
(attTime + susTime +

```

SINES - BINAURAL BEATS

```
}
{impulse == 1.2} {//DUAL
```

```

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame
//synth1.envirPut(Synt
h(\sineBeats, [freq1, freq1, \freq2, freq2, \attTime, attTime, \susTime, susTime, \relTime, relTime,
\limitThresh, (-1.dbamp), \amp, ampBus, \out, outBus]).moveBefore(~bufRecorder)); //<--ORIG

synth1.envirPut(Synth(\sineBeats_rand, [freqMin, freq1, \freqMax, freq2, \beatHzMin, 4,
\beatHzMax, 12, \attTimeMin, 2.2, \attTimeMax, 3.5, \susTimeMin, 1.25, \susTimeMax, 2.87,
\relTimeMin, 2.1, \relTimeMax, 3.45, \limitThresh, (-0.8.dbamp), \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder)); //<--NEW (HARDCODED FOR NOW, CHANGE LATER.....?)

NodeWatcher.register(synth1.envirGet);

true;
= 1;}.defer; //update GUI
sineBeats".postln;});
relTime + impPostDel).yieldAndReset;

~recSamp_input =
{~button_recln.value
if (post == 1, {"TRIG:
(attTime + susTime +
}
{impulse == 2}{

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame
synth1.envirPut(Synth(\sawSweep1, [freq1, freq1, \freq2, freq2, \dur, susTime, \amp, ampBus,
\out, outBus]).moveBefore(~bufRecorder));

NodeWatcher.register(synth1.envirGet); //<---NEW

true;
= 1;}.defer; //update GUI
sawSweep1 | sweepDur =" + susTime).postln;});
impPostDel).yieldAndReset;

IMPULSE FROM DETECTED LIVE SOUND INPUT
liveSound.....").postln;});
routine (and ~autoPlay?)<-----

~recSamp_input =
{~button_recln.value
if (post == 1, {"TRIG:
(susTime +
}
{impulse == 3} {//CREATE
if (post == 1, {"TRIG:
//START ~autoRec
rate.yieldAndReset;
};
},
{//else if (initFlag.envirGet == 0)
var impCase, cond =

//repeating monitoring,
if (post == 1, {"[AUTO

impCase = case
{impulse >= 0 && impulse < 3}
if (~recSamp_input ==
{
if

Condition.new;

recording, playback
CONVOLVE] -----".postln;});

{ //<----- IF ANY SYNTH IR USED --- v v v v v v -----
true,

(post == 1, {"->recSamp_input = true.....").postln;});

```

```

(~playSamp == true, { //b/c of implementation, we don't re-enter this part of routine until sample
  playback has finished (post sampleDur.yieldAndReset)
  ~playSamp = false; //<---not being used at the moment, but could be useful?
  {~button_playSamp.value = 0;}.defer; //update GUI
  ~audioBuf_play0.free; //<---cleanup after reset from prev part of routine
  if (post == 1, {"-->~playSamp = true --- cleaning up...").postln;});
  });
  //list
  // or
  en_for_sound routine tests for silence in room from ~inputSendBus
  should I just test for rms? <-----MAYBE JUST TEST RMS <-----
  featureBus.get({lval //get featBus - 4 features: [Loudness, Onset, Freq, Pitch Measure]
  if (( (val[0] >= loudThresh) || (val[1] == 1) || (val[3] >= pitchThresh) ),
    { //if thresh exceeded, ~hearSound = true
      if (~hearSound != true, {~hearSound = true;});
      if (post == 1, {
        (">SOUND DETECTED...").postln;
        (">>Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3]
        ++ "J").postln;
      });
    },
    { //if thresh not exceeded, ~hearSound = false
      if (~hearSound != false, {~hearSound = false;});
      if (post == 1, {
        (">NO sound detected...").postln;
        (">>Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3]
        ++ "J").postln;
      });
    }
  );
  cond.unhang;
  });
  cond.hang;
  if
  (~hearSound == true, //wait for sound to end before writing sample
  {

```



```

        if (post == 1, {"-->Post-hang: ~hearSound = true | ...waiting for sound to
end...").postln;});

        rate.yieldAndReset;

    },

    {//else if ~hearSound == false

        var currentRecFrame = ~phaseBusMonRec.getSynchronous;

        if (post == 1, {"-->Post-hang: ~hearSound = false | ...sound stopped...write
sample").postln;});

        //<-----BUT sound may not have fully stopped, OR, servo noise may be starting at
this time...log sound stop position, but keep listening for a short delay to ensure sound has stopped?
<-----

        /*if (~playSamp == true, //consider as a way to thoroughly test for end of reverb tail
possibly caused by impulse responses

            {//if we just finished playing a sample

                ~playSamp = false;

                intraSoundTestDur.yieldAndReset;

            },

            {//

            }

        ); */

        ~recSamp_input_stopFrame = (currentRecFrame + (postRecPad *
s.sampleRate)).round(1);

        if (~recSamp_input_stopFrame > (~audioBuf0.numFrames - 1), {

            ~recSamp_input_stopFrame = ~recSamp_input_stopFrame -
~audioBuf0.numFrames;

        });

        ~writeSampFromBufCopy_input.(); //sets GUI and ~recSamp_input = false (ORIG)

        //~writeSampFromBuf_name(buffer: ~audioBuf0, in: ~recSamp_input_startFrame,
out: ~recSamp_input_stopFrame, inputSec: 0, filePath: filePath); //<----NEW (custom file naming) NOT
WORKING 100%, MESSES UP CYCLING OF ROUTINE <-----

        if (post == 1, {"-->slight yield after sampleWrite...").postln;});

        0.2.yield; //<---- slight yield between sampleWrite and playback??

        if (post == 1, {"-->yield complete...").postln;});

        rate.yieldAndReset; //not sure how I feel about doing this w/out a check to confirm
sound has stopped (as in orig ~autoRec_soundTrig routine), but this is simpler and may work for now

```

```

    }
    );

    },
    { //if
~recSamp_input == false
    var
currentPlayCount;

    currentPlayCount = (repeatCounter.envirGet) + 1; //update repeat counter variable

    if
(currentPlayCount > repeatLimit,

        { //if we have reached repeat limit, stop process, reset to beginning of routine...

            if (post == 1, {"[autoConv_voxels] REPEAT LIMIT EXCEEDED - RESET
CYCLE").postln; ("[autoConv_voxels] --> currentRep =" + currentPlayCount + "repLimit =" +
repeatLimit).postln;});

            initFlag.envirPut(1); //reset init state flag to trigger initialization side of routine...

            repeatCounter.envirPut(0); //reset sample playback iteration counter

            if (post == 1, {"[autoConv_voxels] initFlag state reset to 1 | repeatCounter reset to
0").postln;});

            rate.yieldAndReset;

        },

        { //else if we haven't reached repeat limit, carry on, playback last recorded sample

            var playDur;

            if (post == 1, {"[autoConv_voxels] REPEAT LIMIT NOT EXCEEDED - CONTINUE
PLAY/REC").postln; ("[autoConv_voxels] --> currentRep =" + currentPlayCount + "repLimit =" +
repeatLimit).postln;});

            repeatCounter.envirPut(currentPlayCount);

            if (post == 1, {"->recSamp_input = FALSE.....").postln;});

            //<-----IF repeatCount > 5 or 6, make volume level adjustments....<-----

            ~playSamp = true; //not used other than to update GUI, maybe more use though?

            //SET ~recSamp_input_startFrame
            ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;
            ~recSamp_input = true;
            {~button_recln.value = 1;}.defer; //update GUI

            //PLAY BACK PREVIOUS RECORDED SAMPLE
            ~audioBuf_play0 = Buffer.read(s, ~fileAudio_writeln_lastFile, action: { |buffer|
                playDur = buffer.numFrames/buffer.sampleRate;

```

```

        if (timeStretch == true,
            {
                var pitchOffset;
                playDur = (playDur/timeStretchRate).round(0.001);
                pitchOffset = timeStretchRate.reciprocal; //inverse of rate
to preserve orig pitch

                synth1.envirPut(Synth(\bufRead_complex1, [\buf,
buffer, \amp, ampBus, \out, outBus, \rate, timeStretchRate, \pitch, pitchOffset, \winSize, 0.1, \timeDisp,
0.003, \pShiftMix, 1, \duration, playDur])); //play time-stretched sample

                NodeWatcher.register(synth1.envirGet); //<---NEW
                if (post == 1, {"-->PLAY LAST SAMP I TIME STRETCH ="
+ timeStretchRate).postln;});
            },
            //else if no timeStretching
                playDur = playDur.round(0.001);
                synth1.envirPut(Synth(\bufRead_noLoop, [\buf, buffer,
\amp, ampBus, \out, outBus])); //play unaltered sample
                NodeWatcher.register(synth1.envirGet); //<---NEW
                if (post == 1, {"-->PLAY LAST SAMP I no time
stretch...").postln;});
        }
    );

});
{~button_playSamp.value = 1;}.defer; //update GUI
if (post == 1, {"-->Pre-sync samp Dur = " ++ playDur).postln;});

s.sync;

if (post == 1, {"-->Post-sync samp Dur = " ++ playDur).postln;});
playDur.yieldAndReset; //leave this side of routine after reset b/c ~recSamp_input =
true
}

    );
}
}
{impulse == 3} { //<-----

LIVE SOUND IMPULSE ----- !!!!!

    rout2.envirPut(~autoRecPlay_conv0.(featureBus: featureBus, outBus: outBus, ampBus:
ampBus, synth1: synth1, repeatCounter: repeatCounter, loudThresh: loudThresh, pitchThresh:

```



```

~autoConv_repLim_selfTerm ACTIVE - waiting for it to end..."postln;});
                                if (post == 1, {[autoConvSeq]
                                0.5.yieldAndReset;
                                },
                                { //else if ~autoConvActive == false
                                var impCase;
                                impCase = case
                                {impulseFlag.envirGet == 0}

//if previous autoConv impulse was 0 (sus white noise)
                                var prevImpulse = 0,
newImpulse = 1.1;
                                if (post == 1,
{("[autoConvSeq] ~autoConv_repLim_selfTerm COMPLETE I prev impulse =" + prevImpulse + "I new
impulse =" + newImpulse).postln;});

                                impulseFlag.envirPut(newImpulse); //update impulse flag

~autoConvolve2_repLim_selfTerminate.(1, activeFlag: \autoConvActive, filePath:
~fileAudio_writeln_lastFile, featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus:
~fileOutputLevelBus.asMap, rout1: \autoC1, rout2: \autoC2, initFlag: \autoC1state, repeatCounter:
\autoCnt1, repeatLimit: 3, selfTerminate: true, synth1: \autoS1, preRecPad: 0, postRecPad: 0, impulse:
1.1, freq1: 300, freq2: 850, impPostDel: 0, timeStretch: true, timeStretchRate: 0.7, post: post);
                                5.yieldAndReset;
                                }
                                {impulseFlag.envirGet == 1.1}{
                                var prevImpulse = 1.1,
newImpulse = 2;
                                if (post == 1,
{("[autoConvSeq] ~autoConv_repLim_selfTerm COMPLETE I prev impulse =" + prevImpulse + "I new
impulse =" + newImpulse).postln;});

                                impulseFlag.envirPut(newImpulse); //update impulse flag

~autoConvolve2_repLim_selfTerminate.(1, activeFlag: \autoConvActive, filePath:
~fileAudio_writeln_lastFile, featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus:
~fileOutputLevelBus.asMap, rout1: \autoC1, rout2: \autoC2, initFlag: \autoC1state, repeatCounter:
\autoCnt1, repeatLimit: 6, selfTerminate: true, synth1: \autoS1, preRecPad: 0, postRecPad: 0, impulse:
2, freq1: 100, freq2: 20000, susTime: 0.5, impPostDel: 0.1, timeStretch: true, timeStretchRate: 0.6, post:
post);
                                2.yieldAndReset;
                                }
                                {impulseFlag.envirGet == 2}{
                                var prevImpulse = 2,
newImpulse = 0;
                                if (post == 1,
{("[autoConvSeq] ~autoConv_repLim_selfTerm COMPLETE I prev impulse =" + prevImpulse + "I new
impulse =" + newImpulse).postln;});

                                impulseFlag.envirPut(newImpulse); //update impulse flag

~autoConvolve2_repLim_selfTerminate.(1, activeFlag: \autoConvActive, filePath:
~fileAudio_writeln_lastFile, featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus:
~fileOutputLevelBus.asMap, rout1: \autoC1, rout2: \autoC2, initFlag: \autoC1state, repeatCounter:
\autoCnt1, repeatLimit: 7, selfTerminate: true, synth1: \autoS1, preRecPad: 0, postRecPad: 0, impulse:
0, attTime: 1.4, susTime: 3.5, relTime: 2.5, impPostDel: 0.2, timeStretch: true, timeStretchRate: 0.75,
post: post); //<---FOR VOXELS _ 0
                                6.yieldAndReset;
                                };
                                }
                                );
                                };
                                );
                                routine.envirPut(rout);
                                SystemClock.play(routine.envirGet);

```

```

    },
    { //else if active == 0
      routine.envirGet.stop;
      ~autoConvolve2_repLim_selfTerminate.(0, rout1: \autoC1, rout2: \autoC2,
synth1: \autoS1);
      if (post == 1, {"[autoConvSeq] STOPPED -----".postln;});
    }
  );
};

~autoConvolve2_repLim_selfTerminate = {active, activeFlag, featureBus, outBus, ampBus, filePath,
loudThresh = 5, pitchThresh = 0.55, rout1, rout2, synth1, initFlag, repeatCounter, repeatLimit = 8,
selfTerminate = true, impulse = 0, freq1 = 100, freq2 = 2000, attTime = 5, susTime = 3, relTime = 6,
impPostDel = 0, preRecPad = 0.03, postRecPad = 0.02, timeStretch = false, timeStretchRate = 0.9, rate
= 0.1, post = 0!
  if (active == 1,
    {
      if (~bufRecorder.isRunning,
        {
          var rout;

          initFlag.envirPut(1); //set init state flag
          repeatCounter.envirPut(0); //reset sample playback
iteration counter
          activeFlag.envirPut(true); //set activity state flag

          rout = Routine({
            if (initFlag.envirGet == 1,
              {
                var impCase;
                if (post == 1, {"===== [INIT
AUTO CONVOLVE] =====".postln;});

                initFlag.envirPut(0); //move to
post-init sequence of routine after next .yieldAndReset

                //if (post == 1, {"...short pause
//0.4.yield; //slight pause for

                impCase = case
                {impulse == 0} { //SUSTAINED
WHITE NOISE IMPULSE

                ~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous; // <-- log sound start
frame

                synth1.envirPut(Synth\susWhite1, [attTime, attTime, \susTime, susTime, \relTime, relTime,
\amp, ampBus, \out, outBus]).moveBefore(~bufRecorder));

                NodeWatcher.register(synth1.envirGet); //<---NEW

                ~recSamp_input =
true;
                {~button_recln.value
= 1;}.defer; //update GUI
                if (post == 1, {"TRIG:
susWhite1".postln;});
                (attTime + susTime +
relTime + impPostDel).yieldAndReset;
                }
                {impulse == 0.1}

                //TRANSIENT WHITE NOISE PING IMPULSE

```

```

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

    synth1.envirPut(Synth(\impulseWhite3, [\shelf_freq, 10000, \rs, 1, \shelf_db, -6, \peak_freq,
160, \peakQ, 12, \peak_db, 12, \attackTime, 0.001, \releaseTime, 0.13, \curve, -4, \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder));

    NodeWatcher.register(synth1.envirGet); //<---NEW

true;
= 1;}.defer; //update GUI
impulseWhite3".postln;});
impPostDel).yieldAndReset;

~recSamp_input =
{~button_recln.value
if (post == 1, {"TRIG:
(0.14 +
}
{impulse == 1}{ //SINEWAVE

CHIRP / SWEEP IMPULSE

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

    synth1.envirPut(Synth(\sinSweep1, [\freq1, freq1, \freq2, freq2, \dur, susTime, \amp,
ampBus, \out, outBus]).moveBefore(~bufRecorder));

    NodeWatcher.register(synth1.envirGet); //<---NEW

true;
= 1;}.defer; //update GUI
sinSweep1 | sweepDur =" + susTime).postln;});
impPostDel).yieldAndReset;

~recSamp_input =
{~button_recln.value
if (post == 1, {"TRIG:
(susTime +
}
{impulse == 1.1}{ //RANDOM

SINEWAVE SWELL IMPULSE

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

    //synth1.envirPut(Synt
h(\randSinSwell_1, [\attTime, attTime, \susTime, susTime, \relTime, relTime, \loFreq, freq1, \hiFreq,
freq2, \amp, ampBus, \out, outBus]).moveBefore(~bufRecorder)); //<--orig

    synth1.envirPut(Synth(\randSinSwell_lim_1, [\attTime, attTime, \susTime, susTime, \relTime,
relTime, \loFreq, freq1, \hiFreq, freq2, \limitThresh, (-1.dbamp), \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder)); //<---NEW

    NodeWatcher.register(synth1.envirGet); //<---NEW

true;
= 1;}.defer; //update GUI
randSines".postln;});
relTime + impPostDel).yieldAndReset;

~recSamp_input =
{~button_recln.value
if (post == 1, {"TRIG:
(attTime + susTime +
}
{impulse == 1.2}{ //DUAL

SINES - BINAURAL BEATS

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

```

```

//synth1.envirPut(Synth
h(\sineBeats, [freq1, freq1, \freq2, freq2, \attTime, attTime, \susTime, susTime, \relTime, relTime,
\limitThresh, (-1.dbamp), \amp, ampBus, \out, outBus]).moveBefore(~bufRecorder)); //<--ORIG

synth1.envirPut(Synth(\sineBeats_rand, [freqMin, freq1, \freqMax, freq2, \beatHzMin, 4,
\beatHzMax, 12, \attTimeMin, 2.2, \attTimeMax, 3.5, \susTimeMin, 1.25, \susTimeMax, 2.87,
\relTimeMin, 2.1, \relTimeMax, 3.45, \limitThresh, (-0.8.dbamp), \amp, ampBus, \out,
outBus]).moveBefore(~bufRecorder)); //<--NEW (HARDCODED FOR NOW, CHANGE LATER.....?)

NodeWatcher.register(synth1.envirGet);

true;
= 1;}.defer; //update GUI
sineBeats".postln;});
relTime + impPostDel).yieldAndReset;

}
{impulse == 2}{

~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;// <-- log sound start
frame

synth1.envirPut(Synth(\sawSweep1, [freq1, freq1, \freq2, freq2, \dur, susTime, \amp, ampBus,
\out, outBus]).moveBefore(~bufRecorder));

NodeWatcher.register(synth1.envirGet); //<---NEW

true;
= 1;}.defer; //update GUI
sawSweep1 | sweepDur =" + susTime).postln;});
impPostDel).yieldAndReset;

IMPULSE FROM DETECTED LIVE SOUND INPUT
liveSound.....").postln;});
routine (and ~autoPlay?)<-----

}
{impulse == 3}{//CREATE
if (post == 1, {"TRIG:
//START ~autoRec
rate.yieldAndReset;
};
},
{//else if (initFlag.envirGet == 0)
var impCase, cond =

//repeating monitoring,
if (post == 1, {"[AUTO

impCase = case
{impulse >= 0 && impulse < 3}
if (~recSamp_input ==

{
if

(post == 1, {"->recSamp_input = true.....").postln;});

```



```

(~playSamp == true, { //b/c of implementation, we don't re-enter this part of routine until sample
  playback has finished (post sampleDur.yieldAndReset)
  ~playSamp = false; //<---not being used at the moment, but could be useful?
  {~button_playSamp.value = 0;}.defer; //update GUI
  ~audioBuf_play0.free; //<---cleanup after reset from prev part of routine
  if (post == 1, {"-->~playSamp = true --- cleaning up...").postln;});
  });
  //list
  en_for_sound routine tests for silence in room from ~inputSendBus
  // or
  should I just test for rms? <-----MAYBE JUST TEST RMS <-----
  featureBus.get({lval //get featBus - 4 features: [Loudness, Onset, Freq, Pitch Measure]
  if (( (val[0] >= loudThresh) || (val[1] == 1) || (val[3] >= pitchThresh) ),
    { //if thresh exceeded, ~hearSound = true
      if (~hearSound != true, {~hearSound = true;});
      if (post == 1, {
        (">SOUND DETECTED...").postln;
        (">>Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3]
++ "J").postln;
      });
    },
    { //if thresh not exceeded, ~hearSound = false
      if (~hearSound != false, {~hearSound = false;});
      if (post == 1, {
        (">NO sound detected...").postln;
        (">>Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3]
++ "J").postln;
      });
    }
  );
  cond.unhang;
  });
  cond.hang;
  if
  (~hearSound == true, //wait for sound to end before writing sample
  {

```

```

end...").postln;});
    if (post == 1, {"-->Post-hang: ~hearSound = true | ...waiting for sound to
        rate.yieldAndReset;
    },
    { //else if ~hearSound == false
        var currentRecFrame = ~phaseBusMonRec.getSynchronous;
        if (post == 1, {"-->Post-hang: ~hearSound = false | ...sound stopped...write
sample").postln;});

        //<-----BUT sound may not have fully stopped, OR, servo noise may be starting at
this time...log sound stop position, but keep listening for a short delay to ensure sound has stopped?
<-----

        /*if (~playSamp == true, //consider as a way to thoroughly test for end of reverb tail
possibly caused by impulse responses

            { //if we just finished playing a sample
                ~playSamp = false;
                intraSoundTestDur.yieldAndReset;
            },
            { //

            }

        ); */

        ~recSamp_input_stopFrame = (currentRecFrame + (postRecPad *
s.sampleRate)).round(1);
        if (~recSamp_input_stopFrame > (~audioBuf0.numFrames - 1), {
            ~recSamp_input_stopFrame = ~recSamp_input_stopFrame -
~audioBuf0.numFrames;
        });

        ~writeSampFromBufCopy_input.(); //sets GUI and ~recSamp_input = false (ORIG)

        //~writeSampFromBuf_name(buffer: ~audioBuf0, in: ~recSamp_input_startFrame,
out: ~recSamp_input_stopFrame, inputSec: 0, filePath: filePath); //<----NEW (custom file naming) NOT
WORKING 100%, MESSES UP CYCLING OF ROUTINE <-----

        if (post == 1, {"-->slight yield after sampleWrite...").postln;});
        0.2.yield; //<---- slight yield between sampleWrite and playback??
        if (post == 1, {"-->yield complete...").postln;});

        rate.yieldAndReset; //not sure how I feel about doing this w/out a check to confirm
sound has stopped (as in orig ~autoRec_soundTrig routine), but this is simpler and may work for now

```

```

    }
    );
    },
    { //if
~recSamp_input == false
    var
currentPlayCount;

    currentPlayCount = (repeatCounter.envirGet) + 1; //increment repeat counter

    if
(currentPlayCount > repeatLimit,
    { //if we have reached repeat limit, repeat or stop process based on selfTerminate var...
        if (selfTerminate == false,
            {
                if (post == 1, {"[autoConv_voxels] -- selfTerminate == false |
continue process...").postln;});
                if (post == 1, {"[autoConv_voxels] REPEAT LIMIT EXCEEDED -
RESET CYCLE").postln; ("[autoConv_voxels] --> currentRep =" + currentPlayCount + "repLimit =" +
repeatLimit).postln;});
                of routine...
                initFlag.envirPut(1); //reset init state flag to trigger initialization side
                counter
                repeatCounter.envirPut(0); //reset sample playback iteration
                if (post == 1, {"[autoConv_voxels] initFlag state reset to 1 |
repeatCounter reset to 0").postln;});
                rate.yieldAndReset;
            },
            { //else if selfTerminate == true
                if (post == 1, {"[autoConv_voxels] -- selfTerminate == true | STOP
process and cleanup...").postln;});
                activeFlag.envirPut(false); //update activity state flag
                if (post == 1, {"[autoConv_voxels] -- TERMINATING ~autoConv |
activeFlag =" + activeFlag.envirGet).postln;});
            }
        );
    },
    { //else if we haven't reached repeat limit, carry on, playback last recorded sample
        var playDur;
        if (post == 1, {"[autoConv_voxels] REPEAT LIMIT NOT EXCEEDED - CONTINUE
PLAY/REC").postln; ("[autoConv_voxels] --> currentRep =" + currentPlayCount + "repLimit =" +
repeatLimit).postln;});

```

```

repeatCounter.envirPut(currentPlayCount); //update repeat counter env variable
if (post == 1, {"-->recSamp_input = FALSE.....").postln;});

//<-----IF repeatCount > 5 or 6, make volume level adjustments....<-----

//SET ~recSamp_input_startFrame
~recSamp_input_startFrame = ~phaseBusMonRec.getSynchronous;
~recSamp_input = true;
{~button_recln.value = 1;}.defer; //update GUI

~playSamp = true; //not used by anything right now....
{~button_playSamp.value = 1;}.defer; //update GUI

//PLAY BACK PREVIOUS RECORDED SAMPLE
~audioBuf_play0 = Buffer.read(s, ~fileAudio_writeIn_lastFile, action: {lbufferl
    playDur = buffer.numFrames/buffer.sampleRate;

    if (timeStretch == true,
        {
            var pitchOffset;
            playDur = (playDur/timeStretchRate).round(0.001);
            pitchOffset = timeStretchRate.reciprocal; //inverse of rate
to preserve orig pitch

            synth1.envirPut(Synth(\bufRead_complex1, [\buf,
buffer, \amp, ampBus, \out, outBus, \rate, timeStretchRate, \pitch, pitchOffset, \winSize, 0.1, \timeDisp,
0.003, \pShiftMix, 1, \duration, playDur])); //play time-stretched sample

            NodeWatcher.register(synth1.envirGet); //<---NEW
            if (post == 1, {"-->PLAY LAST SAMP I TIME STRETCH ="
+ timeStretchRate).postln;});

        },
        {
            //else if no timeStretching
            playDur = playDur.round(0.001);

            synth1.envirPut(Synth(\bufRead_noLoop, [\buf, buffer,
\amp, ampBus, \out, outBus])); //play unaltered sample

            NodeWatcher.register(synth1.envirGet); //<---NEW
            if (post == 1, {"-->PLAY LAST SAMP I no time
stretch...").postln;});

        }
    }

```

```

    );
});

if (post == 1, {"--->Pre-sync samp Dur = " ++ playDur).postln;});

s.sync;

if (post == 1, {"--->Post-sync samp Dur = " ++ playDur).postln;});

playDur.yieldAndReset; //leave this side of routine after reset b/c ~recSamp_input =
true
}

);

});

{impulse == 3} { //<-----

LIVE SOUND IMPULSE ----- !!!!!

    rout2.envirPut(~autoRecPlay_conv0.(featureBus: featureBus, outBus: outBus, ampBus:
ampBus, synth1: synth1, repeatCounter: repeatCounter, loudThresh: loudThresh, pitchThresh:
pitchThresh, preRecPad: preRecPad, postRecPad: postRecPad, impPostDel: impPostDel, timeStretch:
timeStretch, timeStretchRate: timeStretchRate, debounceDur: 0, post: post));

    SystemClock.play(rout2.envirGet); //launch autoRecPlay_conv0 routine
    });

});

    rout1.envirPut(rout);
    SystemClock.play(rout1.envirGet);

    },
    {
        "===CAN'T RUN AUTOCONV UNLESS BUF RECORDER
and ~listen_sound ACTIVE===".postln;
    }
    );
    },
    { //else if active == 0 (disable)
        rout1.envirGet.stop;
        if (rout2 != nil, {rout2.envirGet.stop;});
        if (synth1.envirGet.isRunning == true, {synth1.envirGet.free;}); //<---NEW
        if (~recSamp_input == true, {
            ~recSamp_input = false;
            ~button_recln.value = 0;}.defer; //update GUI
        });
        if (~playSamp == true, {
            ~playSamp = false; //<---not being used at the moment, but could
be useful?
            {~button_playSamp.value = 0;}.defer; //update GUI
            ~audioBuf_play0.free;
        });
    }
    );
};
//-----PARTITIONED CONVOLUTION EFFECT FUNC / ROUTINES -----

```

```

~partConv_samps_autoLoop = {lactive = 1, targetFile, irFile, targetBuf = \pcs_tBuf1, irBuf =
\pcs_irBuf1, fftSize = 2048, routine = \pcs_rout1, synth = \pcs_syn1, ampBus, ampLimit = 0.4, outBus =
0, attTime = 5, susTime = 3, relTime = 6, playRate = 1, pitch = 1, pShiftMix = 0, winSize = 0.1, pitchDisp
= 0, timeDisp = 0.003, post = 0}
  if (active == 1, {//ACTIVATE
    var rout = Routine({
      var irbuffer, bufsize, routineDur;
      irbuffer = Buffer.read(s, irFile); //load IR audio file into buffer
      s.sync;
      bufsize = PartConv.calcBufSize(fftsize: fftSize, irbuffer: irbuffer);
      irBuf.envirPut(Buffer.alloc(s, bufsize, 1));
      irBuf.envirGet.preparePartConv(buf: irbuffer, fftsize: fftSize); //create IR
spectrum and load to separate buffer
      s.sync;
      irbuffer.free; // don't need time domain data buffer anymore, just need
spectral version ("irBuf")

      targetBuf.envirPut(Buffer.read(s, targetFile)); //load target file to buffer

      synth.envirPut(Synth(\partConv_sampleLoop_pShift, [targetBuf,
targetBuf.envirGet, \irSpectrum, irBuf.envirGet, \attTime, attTime, \susTime, susTime, \relTime,
\fftSize, fftSize, \playRate, playRate, \pitch, pitch, \pShiftMix, pShiftMix, \winSize, winSize, \pitchDisp,
pitchDisp, \timeDisp, timeDisp, \ampBus, ampBus, \ampLimit, ampLimit, \out, outBus])); //launch
partConv synth

      NodeWatcher.register(synth.envirGet);

      routineDur = attTime + susTime + relTime;

      if (post == 1, {
        var targetFileName, irFileName;
        targetFileName = ~returnFileNameFromPath(path: targetFile);
        irFileName = ~returnFileNameFromPath(path: irFile);
        ("[pConv_samps_autoLoop] Convolving TARGET:" +
targetFileName + "w/ IR:" + irFileName + "for" + routineDur + "sec...").postln;
        ("[pConv_samps_autoLoop] -- RATE =" +
playRate.ratiomidi.round(0.01) + "I PITCH =" + pitch.ratiomidi.round(0.01) + "I pShiftMix =" +
pShiftMix.round(0.01)).postln;
      });

      (attTime + susTime + relTime + 0.01).yield; //wait for synth to finish playing

      targetBuf.envirGet.free; //cleanup buffers when synth finished playing
      irBuf.envirGet.free;
      if (post == 1, {"[pConv_samps_autoLoop] COMPLETE".postln;});
    });
    routine.envirPut(rout);
    SystemClock.play(routine.envirGet);
  }, {//else DEACTIVATE if active == 0
    if (synth.envirGet.isRunning == true, {
      synth.envirGet.free;
      targetBuf.envirGet.free; //putting in here to avoid needlessly freeing if
routine's already ended and freed this
      irBuf.envirGet.free;
    }); //avoids potential thrown error, but...
    routine.envirGet.stop;
    if (post == 1, {"[pConv_samps_autoLoop] STOPPED"});
  });
};

//---- Loudness / Feature Level Monitoring ----

~updateFeatureMemory = {lactive = 1, featureBus, array, arraySize = 6, routine, rate = 10, post = 0,
postLabel = "-->memArray1:"I //<--NEW
  if (active == 1,
    {

```

```

        var rout;
        array.envirPut(Array.newClear(arraySize));
        rout = Routine({ //update feature memory array at selected rate
            var temp, val = featureBus.getSynchronous.round(0.001); //<----
rounding or no? conditional as arg??
            temp = array.envirGet.shift(1, val); //add latest featBus value to
beginning of array & shift prev vals
            array.envirPut(temp);
            if (post == 1, {
                ("-----").postln;
                (postLabel.asString + array.envirGet + "I total time-span ="
+ (arraySize * rate)).postln;
                ("-----").postln;
            });
            rate.yieldAndReset;
        });
        routine.envirPut(rout);
        SystemClock.play(routine.envirGet); //start memory update routine
    },
    //else if active == 0
    //clean up / delete mem array or no?
    routine.envirGet.stop; //stop memory update routine
    }
);

~avgArray = {array, post = 0, postLabel = "array1-avg:"I //<--NEW
    var size, runningSum, arrayAvg;
    size = array.size;
    runningSum = 0;
    for(0, (size - 1), {lil runningSum = array[i] + runningSum;});
    arrayAvg = runningSum * size.reciprocal;
    if (post == 1, {(postLabel.asString + arrayAvg).postln;});
    arrayAvg;
};

~smoothBusOutputKR = {lactive = 1, inBus, outBus, synth, synthMoveAfter, avgDur = 1I //<---NEW
    if (active == 1,
        {
            var thisOutBus;
            outBus.envirPut(Bus.control(s, 1)); //only one channel control bus for now,
consider arg to modify this in future
            thisOutBus = outBus.envirGet;
            synth.envirPut(Synth(\avgKR, [input, inBus, \output, thisOutBus, \avgDur,
avgDur]).moveAfter(synthMoveAfter)); //create smoothing synth
        },
        //else if active == 0
        synth.envirGet.free; //free synth
        outBus.envirGet.free; //free output bus
    }
);

//---- Misc / Utility ----
~returnFileNameFromPath = {lpathI
    var thisPath, pathArray, fileName;
    thisPath = path.asString;
    pathArray = thisPath.split; //returns array separating each word in path
    fileName = pathArray[pathArray.size - 1]; //return value from last index of array (a.k.a. target
file name)
};

~loadaudioBuf = {lactive = 1, filePath, bufferName, returnDurl //utility function to load audio file to buffer
and cleanup

```

```

        if (active == 1, { //create audio buffer, read in audio file, assign to env var, also calc duration in
sec & return to env var
            bufferName.envirPut(Buffer.read(s, filePath, action: { |buffer|
returnDur.envirPut(buffer.numFrames/buffer.sampleRate;});});
        }, { //else if active == 0, cleanup
            bufferName.envirGet.free;
        });
    };

//===== ROUTINES =====

~initSMpreset_1 = Routine({ //auto-startup routine for RPI <===UPDATE BEFORE NEXT PUSH TO
RPI
    "...starting...".postln;
    2.yield;
    "...test tone...".postln;
    Synth(\startupTone_1);
    10.yield;
    //SET LEVELS
    "...set levels...".postln;
    ~inputStreamLevelBus.set(0.3); //<===UPDATE BEFORE NEXT PUSH TO RPI (convert to
db...)
    ~slider_inFlow1_level.value = (0.3); //<===UPDATE BEFORE NEXT PUSH TO RPI (convert to
db...)
    ~numBox_inFlow1_level.value = (0.3); //<===UPDATE BEFORE NEXT PUSH TO RPI (convert
to db...)
    ~fileOutputLevelBus.set(0.9); //<===UPDATE BEFORE NEXT PUSH TO RPI (convert to db...)
    ~slider_sampOut_level.value = (0.9); //<===UPDATE BEFORE NEXT PUSH TO RPI (convert
to db...)
    ~numBox_sampOut_level.value = (0.9); //<===UPDATE BEFORE NEXT PUSH TO RPI
(convert to db...)
    0.5.yield;
    ~button_recBuf.valueAction = 1;
    "START STREAMING TO BUF0...".postln;
    "...buffering audio for 4 more seconds...".postln;
    4.yield;
    "START BUF0 PLAYBACK".postln;
    ~button_playBuf.valueAction = 1;
    "...waiting a couple more seconds...".postln;
    2.yield;
    "STARTING RT proc behaviors...".postln;
    ~button_Behavior1.valueAction = 1;
    "...waiting a few more seconds...".postln;
    3.yield;
    "STARTING AUTO-REC-PLAY behavior...".postln;
    ~button_autoRecPlay.valueAction = 1;
});
~initSMpreset_2 = Routine({ //auto-startup routine for RPI <===UPDATE BEFORE NEXT PUSH TO
RPI
    "...starting...".postln;
    2.yield;
    "...test tone...".postln;
    Synth(\sineBeats_rand, [\freqMin, 200, \freqMax, 450, \beatHzMin, 8, \beatHzMax, 20,
\attTimeMin, 1, \attTimeMax, 3, \susTimeMin, 3, \susTimeMax, 4, \relTimeMin, 2, \relTimeMax, 3, \amp,
(-6).dbamp, \limitThresh, (-0.3).dbamp, \out, 0]); //play binaural beat startup tone: freq, beats, envelope
randomized
    10.yield;
    //SET LEVELS
    "...set levels...".postln;
    ~inputStreamLevelBus.set((-6).dbamp); //<===UPDATE BEFORE NEXT PUSH TO RPI
(convert to db...)
    ~slider_inFlow1_level.value = 0.75; //<---FIX W/ PROPER MAPPING.....
    ~numBox_inFlow1_level.value = -6; //<===UPDATE BEFORE NEXT PUSH TO RPI (convert
to db...)

```



```

~fileOutputLevelBus.set(1); //<===UPDATE BEFORE NEXT PUSH TO RPI (convert to db...)
~slider_sampOut_level.value = 1; //<===UPDATE BEFORE NEXT PUSH TO RPI (convert to
db...)
~numBox_sampOut_level.value = 0; //<===UPDATE BEFORE NEXT PUSH TO RPI (convert
to db...)
0.5.yield;
~button_recBuf.valueAction = 1;
"START STREAMING TO BUF0...".postln;
"...buffering audio for 4 more seconds...".postln;
4.yield;
"START BUF0 PLAYBACK".postln;
~button_playBuf.valueAction = 1;
"...waiting a couple more seconds...".postln;
2.yield;
"STARTING RT proc behaviors...".postln;
~button_Behavior1.valueAction = 1;
"...waiting a few more seconds...".postln;
3.yield;
"STARTING AUTO-REC-PLAY behavior...".postln;
~button_autoRecPlay.valueAction = 1;
});

~initSMpreset_voxelBot1 = Routine({ //auto-startup routine for RPI <===UPDATE BEFORE NEXT
PUSH TO RPI
"...starting...".postln;
~button_recBuf.valueAction = 1;
"START STREAMING TO BUF0...".postln;
"...waiting a couple seconds...".postln;
2.yield;
//start and set Compressor settings??????
"STARTING autoConvolve proc...".postln;
~button_Behavior3.valueAction = 1;
});

//----- BUS GET / GUI UPDATE ROUTINES -----
~getBus_updateGUI_playPhase = Routine({
~phaseBusMonPly.get({lval
~lastPlayPhasePos = val.round(1);
{
//((val / (s.sampleRate * ~audioBuf0_dur)).postln; //uncomment to view
sliderPlayPos from 0-1 (debugging)
~slider_playPos.value = val / (s.sampleRate * ~audioBuf0_dur); //update
slider position
~numBox_playPos.value = (val.round(1)/s.sampleRate).round(0.01);
//convert frames to seconds and display
}.defer;
});
if (~playPosLatchTrig == true,
{
~playPosLatchTrig = false;
{
~phasor_play.set(\start, ~playInPosBus.asMap); //reset phasor start
to ~playInPos after initializing at ~lastPlayPhasePos (permits resuming playback from last read pos in
buffer)
}.defer(0.15); //<----NOTE TO SELF: in the unlikely event the user stops
playing and then later resumes from within less than 0.15 seconds of the ~playOutPosBus value, this
may cause the play phasor to jump back to its start position initialized at ~lastPlayPhasePos when this
routine starts. Haven't test enough. Low priority....
},
{ //else
~phasor_play.get(\trig, {ltrigval if (trigval > 0, {~phasor_play.set(\trig, 0);}); });
//reset trigger val to 0 if prev triggered
}
});

```

```

    );
    0.1.yieldAndReset;
});

~getBus_updateGUI_recPhase = Routine({
    //(">>>updateRecSliderRoutine: outside get -" + Main.elapsedTime.round(0.01)).postln;
    ~phaseBusMonRec.get({lval
        ~lastRecPhasePos = val.round(1); //log phase pos in env variable <----NEW
        {
            //(">>>updateRecSliderRoutine: inside get -" +
Main.elapsedTime.round(0.01)).postln;
            //(val / (s.sampleRate * ~audioBuf0_dur)).postln; //uncomment to view
sliderPlayPos from 0-1
            ~slider_recPos.value = val / (s.sampleRate * ~audioBuf0_dur); //update
slider position
            ~numBox_recPos.value = (val.round(1)/s.sampleRate).round(0.01) ;
//convert frames to seconds and display
        }.defer;
    });

    if (~recPosLatchTrig == true,
        {
            ~recPosLatchTrig = false;
            {
                ~phasor_rec.set(\start, ~recInPosBus.asMap); //reset phasor start
to ~recInPos after initially starting phasor synth (permits resuming recording from last read pos in buffer)
                //("----reset phasor startPos to recInPosBus:" +
Main.elapsedTime.round(0.01)).postln;
                }.defer(0.15); //<----NOTE TO SELF: in the unlikely event the user stops
recording and then later resumes from within less than 0.15 seconds of the ~recOutPosBus value, this
may cause the rec phasor to jump back to its start position initialized at ~lastRecPhasePos when this
routine starts. Haven't test enough. Low priority....
            },
            { //else
                ~phasor_rec.get(\trig, {l trigval if (trigval > 0, {~phasor_rec.set(\trig,
0);}); }); //reset trigger val to 0 if prev triggered by slider .action (i.e. clicked transport slider)
            }
        );
    0.1.yieldAndReset;
});

~updateGUI_rate = Routine.new ( {
    ~rateBus.get( {
        arg val;
        var thisVal = val.abs; //in case "Reverse" enabled and rate is negative... <-----
AppClock.sched(0.0, { //errors thrown w/out this for some reason... I think can be
replaced by .defer
            //pitchBusVal= ".post; val.ratiomidi.round(0.01).postln;
            ~slider_rate.value_(thisVal.ratiomidi.linlin(-48, 48, 0, 1) );
            ~numBox_rate.value_(thisVal.ratiomidi.round(0.01));
            nil;
        });
    });
    0.1.yieldAndReset;
});

~updateGUI_freq = Routine.new ( {
    ~pitchBus.get( {
        arg val;
        AppClock.sched(0.0, { //this is necessary for some reason...???????
            //pitchBusVal= ".post; val.ratiomidi.round(0.01).postln;
            ~slider_pitch.value_(val.ratiomidi.linlin(-48, 24, 0, 1) );
            ~numBox_pitch.value_(val.ratiomidi.round(0.01));
        });
    });
    0.1.yieldAndReset;
});

```

```

        nil;
    });
    });
    0.1.yieldAndReset;
});

~updateGUI_pitchShiftMix = Routine.new ( {
    ~pitchShiftMixValBus.get( {
        arg val;
        AppClock.sched(0.0, { //this is necessary for some reason...??????
            ~slider_pitchShiftMix.value_(val);
            ~numBox_pitchShiftMix.value_((val * 100).round(1));
            nil;
        });
    });
    0.1.yieldAndReset;
});

~updateGUI_sub1 = Routine.new ( { //--- will eventually optimize this & turn into a function w/ env
    variables and assigned GUI objects as args...
    var level;
    level = ~inputStreamLevelBus.getSynchronous;
    ~slider_inFlow1_level.value_(level);
    //~numBox_inFlow1_level.value_(level);
    ~numBox_inFlow1_level.value_(level.ampdb.round(0.1));
    0.1.yieldAndReset;
});

~updateGUI_sub2 = Routine.new ( { //--- will eventually optimize this & turn into a function w/ env
    variables and assigned GUI objects as args...
    var level;
    level = ~fileOutputLevelBus.getSynchronous;
    ~slider_sampOut_level.value_(level);
    //~numBox_sampOut_level.value_(level);
    ~numBox_sampOut_level.value_(level.ampdb.round(0.1));
    0.1.yieldAndReset;
});

~updateGUI_outMainLevel = Routine.new ( { //--- will eventually optimize this & turn into a function w/
    env variables and assigned GUI objects as args...
    var level;
    level = ~outMainLevelBus.getSynchronous;
    ~slider_outMainLevel.value_(level);
    ~numBox_outMainLevel.value_(level.ampdb.round(0.1));
    0.1.yieldAndReset;
});

~updateGUI_counter_rateT2rand = Routine.new( { //not being used anymore, for old GUI monitoring
    tool
    if (~time2NextRandom_rate <= 0, {~time2NextRandom_rate = 0;});
    ~numBox_counter_rateT2rand.value_(~time2NextRandom_rate);
    ~time2NextRandom_rate = ~time2NextRandom_rate - 0.1;
    0.1.yieldAndReset;
});
// -----

//---- RANDOMIZE ROUTINES ----
~randomRoutine_rate = Routine.new ( { //Automatically randomize playback rate and adjust playHead
    var time2nextRandomize, randNum4transp, newPitch, pitchCase, pitchInterp, interpTime, cond
    = Condition.new; //---

    /*~rateBus.get( { //---OLD
        arg val;
        //(*****currentRate=" + val).post;

```

```

~pitchShiftValPrev_rate = val.ratiomidi;
//("---prevPitch=" + ~pitchShiftValPrev_rate).postln;
}); */
//0.1.yield; //<<<<< not ideal

//~pitchShiftValPrev_rate = ~rateBus.getSynchronous.ratiomidi; //<----NEW (consider this
instead)

~rateBus.get({|vall //<---(NEW) DO THIS IF STICKING W/ GET...safer than using s.sync....
//("*****currentRate=" + val).post;
~pitchShiftValPrev_rate = val.ratiomidi.round(0.01);
//("---prevPitch=" + ~pitchShiftValPrev_rate).postln;
cond.unhang;
//("=== RATE SHIFT - GET - UNHANG").postln;
//("====RATE SHIFT - prevFreq =" + ~pitchShiftValPrev_rate).postln;
});
//("=== RATE SHIFT - PRE-HANG").postln;
cond.hang;

randNum4transp = rrand(1, 10); //based on random # from 1-10, determine type of pitch
transposition
//randNum4transp= ".post; randNum4transp.postln;
pitchCase = case
{randNum4transp <= 4} { //40% chance of lowering pitch
//newPitch = rrand(-7.0, -0.2).round(0.01);
newPitch = rrand(-12.0, -0.2).round(0.01);
//("--newPitch=" + newPitch).post;
}
{randNum4transp >= 5 && randNum4transp <= 8} { //40% chance of no pitch change
newPitch = 0.0; //no transposition...
//("--newPitch=" + newPitch).post;
}
{randNum4transp >= 9} { //20% chance of raising pitch
//newPitch = rrand(1.0, 5.0).round(0.01);
newPitch = rrand(1.0, 12.0).round(0.01);
//("--newPitch=" + newPitch).post;
};

~pitchShiftValTarget_rate = newPitch; //set target pitch to shift to
//("====RATE SHIFT - targetFreq =" + ~pitchShiftValTarget_rate).postln;

if (newPitch != ~pitchShiftValPrev_rate, {
interpTime = rrand(1.2, 7.0);
//("interpTime=" + interpTime).postln;

//interpolate to new rate, is able to reverse during interpolation
pitchInterp = Synth(\LinInterp_rate, \out, ~rateBus, \startVal, ~pitchShiftValPrev_rate,
\stopVal, ~pitchShiftValTarget_rate, \duration, interpTime, \rateMod, ~revRateBusMon.asMap) );

~pitchInterp_rate = true; //flag (currently prevents playRecOffsetAdjust routine from
interrupting pitchInterp)
//("*****pitchInterp=" + ~pitchInterp_rate).postln;
~updateGUI_rate.reset;
AppClock.play(~updateGUI_rate);

(interpTime + 0.1).yield;

~pitchInterp_rate = false;
//("*****pitchInterp=" + ~pitchInterp_rate).postln;
~updateGUI_rate.stop;
time2nextRandomize = rrand(13, 22); //if newPitch != prevPitch
}, {
time2nextRandomize = rrand(5, 10); //reduced possible time2nextRand in case
newPitch == prevPitch
});

```

```

~pitchShiftValPrev_rate = ~pitchShiftValTarget_rate; //<---don't like this, but need until I devise
better solution...

~time2NextRandom_rate = time2nextRandomize; //for populating GUI
//("----TIME2NEXT - RANDOM RATE =" + ~time2NextRandom_rate.round(0.01)).postln;
time2nextRandomize.yieldAndReset;
});

~randomRoutine_freq = Routine.new ( {
    //Automatically randomize pitch w/out time change (granular pitch shift)
    var time2nextRandomize, randNum4transp, newPitch, pitchCase, pitchInterp, interpTime, cond
    = Condition.new;

    //~pitchShiftValPrev_freq = ~pitchBus.getSynchronous.ratiomidi; //<---NEW (consider this
instead)

    ~pitchBus.get( {!val //<---DO THIS IF STICKING W/ GET...safer than using s.sync....
        ~pitchShiftValPrev_freq = val.ratiomidi.round(0.01);
        cond.unhang;
        //("=== PITCH SHIFT - GET - UNHANG").postln;
        //("====PITCH SHIFT - prevFreq =" + ~pitchShiftValPrev_freq).postln;
    });
    //("=== PITCH SHIFT - PRE-HANG").postln;
    cond.hang;

    /*~pitchBus.get( {
        arg val;
        //AppClock.sched(0.0, {
            ~pitchShiftValPrev_freq = val.ratiomidi;
            //("-*-prevFreq=" + ~pitchShiftValPrev_freq).postln;
            //nil;
        });
    } );*/
    //0.1.yield; //<<<<<works, but not ideal way

    randNum4transp = rrand(1.0, 10.0); //based on random # from 1-10, determine type of pitch
transposition
    pitchCase = case
    {randNum4transp <= 5} { //50% chance of raising/lowering pitch randomly
        var thisRandNum = rrand(1, 10);
        if (thisRandNum <= 5, // 50/50 change pitch is raised or lowered
            {newPitch = rrand(-24.0, -0.2).round(0.01);},
            {newPitch = rrand(0.2, 24).round(0.01);} );
    }
    {randNum4transp > 5 && randNum4transp <= 8.5} { //35% chance pitch change in response to
current rate
        newPitch = ~pitchShiftValTarget_rate * -1;
    }
    {randNum4transp > 8.5} { //15% chance of no pitch change
        newPitch = ~pitchShiftValPrev_freq;
    };
    ~pitchShiftValTarget_freq = newPitch; //set target pitch to shift to
    //("====PITCH SHIFT - targetPitch =" + ~pitchShiftValTarget_freq).postln;

    if (newPitch != ~pitchShiftValPrev_freq, { //if newPitch isn't the same as the last...
        var thisDangoRandomNum;
        interpTime = rrand(0.25, 6.0); //===INTERP TIME===
        //("-*-freq_interpTime=" + interpTime).postln;
        pitchInterp = Synth(\LinInterp_pitch, [out, ~pitchBus, \startVal,
~pitchShiftValPrev_freq, \stopVal, ~pitchShiftValTarget_freq, \duration, interpTime] );

        ~pitchInterp_freq = true;
        //("*****pitchInterp=" + ~pitchInterp_rate).postln;

```

```

~updateGUI_freq.reset;
AppClock.play(~updateGUI_freq);

Synth interp?
(interpTime + 0.1).yield; //<----IS IT NECESSARY TO ADD 0.1 to accommodate

~pitchInterp_freq = false;
//("*****pitchInterp=" + ~pitchInterp_rate).postln;
~updateGUI_freq.stop;
//var thisDangoRandomNum = rrand(1, 10);
thisDangoRandomNum = rrand(1.0, 10.0);
if (thisDangoRandomNum <= 4.5, { //45% chance time2next pitch change is SHORT
pitch
    time2nextRandomize = rrand(0.3, 4.0).round(0.1); //if newPitch != the last
    }, { //55% chance time2next pitch change is LONG
        time2nextRandomize = rrand(4, 17);
    });
}, { //ELSE
    time2nextRandomize = rrand(4, 10); //if newPitch == last pitch
});

if (~pitchShiftValTarget_freq == 0, { //if current pShift val == 0, turn pShiftMix to 0%
    var currentMixVal, interpMix;
    currentMixVal = ~pitchShiftMixValBus.getSynchronous.round(0.01);
    interpMix = Synth(LinInterp_num, [out, ~pitchShiftMixValBus, \startVal,
currentMixVal, \stopVal, 0, \duration, 0.25] );
    ~pitchShiftMix_interp = true;
    ~updateGUI_pitchShiftMix.reset;
    AppClock.play(~updateGUI_pitchShiftMix);
    0.35.yield; //+0.1 longer than duration in above interpMix synth, maybe not necessary
to add 0.1...
    ~pitchShiftMix_interp = false;
    ~updateGUI_pitchShiftMix.stop;
    ~pitchShiftMix_prev = 0;
});

~pitchShiftValPrev_freq = ~pitchShiftValTarget_freq; //<---don't like this, but need until I devise
better solution...

~time2NextRandom_freq = time2nextRandomize; //for populating GUI
//("---->TIME 2 NEXT RAND - PITCH SHIFT =" + time2nextRandomize.round(0.01)).postln;
time2nextRandomize.yieldAndReset;
});

~randomRoutine_pitchShiftMix = Routine.new ({
    var time2nextRandomize, currentPitch, randNum, mixCase, interpMix, interpTime;

    /*~pitchBus.get({|vall
    currentPitch = val.ratiomidi.round(0.01); //convert pitch ratio to transposition value in
semitones
    ("---BEFORE SYNC: currentPitch =" + currentPitch).postln;
    }); */
    //("----AFTER SYNC: currentPitch =" + currentPitch).postln;

    //currentPitch = ~pitchBus.getSynchronous.ratiomidi.round(0.01); //<----NEW (WATCH OUT,
DON'T GET FROM SAME BUS FROM TWO ROUTINES AT ONCE!!!!)

    currentPitch = ~pitchShiftValPrev_freq; //<---use this instead of getting from pitchBus

    //("---currentPitch =" + currentPitch).postln;

    if (currentPitch == 0,
        {
            //(">NO PSHIFT - PSHIFT =" + currentPitch).postln;

```

```

    if (~pitchShiftMix_prev > 0, //if PitchShift = 0 but pShiftMix > 0, set pShiftMix
to 0
        {
            var currentMixVal, cond = Condition.new;

            //~pitchShiftMixValBus.get({|vall currentMixVal = val;});
            //0.2.yield; //<----not ideal

            //currentMixVal =
~pitchShiftMixValBus.getSynchronous.round(0.01); //<---NEW (consider)

STICKING W/ GET...
            ~pitchShiftMixValBus.get({|vall //<---DO THIS IF

                currentMixVal = val.round(0.01);
                cond.unhang;
                //("---->pShiftMix - GET - UNHANG").postln;
            });
            //("---->pShiftMix - PRE-HANG").postln;
            cond.hang;

            if (currentMixVal > 0, { //if current pShiftMix val > 0%, set it
to 0%
                //("--PSHIFT == 0 - pShiftMix =" + currentMixVal
+ "SET pShiftMix to 0%").postln;
                interpMix = Synth(LinInterp_num, [out,
~pitchShiftMixValBus, \startVal, currentMixVal, \stopVal, 0, \duration, 0.25] );
                ~pitchShiftMix_interp = true;
                ~updateGUI_pitchShiftMix.reset;
                AppClock.play(~updateGUI_pitchShiftMix);
                0.35.yield; //+0.1 longer than duration in above
interpMix synth, maybe not necessary to add 0.1...
                ~pitchShiftMix_interp = false;
                ~updateGUI_pitchShiftMix.stop;
                ~pitchShiftMix_prev = 0;
            }, {//else if current pShiftMix == 0%, update env variable to
stop querying server for its value...
                //("--PSHIFT == 0 - pShiftMix =" + currentMixVal
+ "SET pShiftMix to 0%").postln;
                ~pitchShiftMix_prev = 0;
            });
            1.yieldAndReset;
        },
        {
            //("--pShiftMixAlready 0%, nothing to do...").postln;
            2.yieldAndReset;
        }
    );
},
{
    var cond = Condition.new; //<---NEW (if using .get method instead of
.getSync)
    //(">PSHIFT DETECTED - PSHIFT =" + currentPitch + "- CHANGING
PSHIFTMIX---").postln;

    //~pitchShiftMixValBus.get({|vall ~pitchShiftMix_prev = val;});
    //0.2.yield; //<----not ideal

    //~pitchShiftMix_prev =
~pitchShiftMixValBus.getSynchronous.round(0.01); //<---NEW (consider)

    ~pitchShiftMixValBus.get({|vall //<---DO THIS IF STICKING W/ GET...
        ~pitchShiftMix_prev = val.round(0.01);
        cond.unhang;
        //("---->pShiftMix - GET - UNHANG").postln;
    });
}

```

```

//("---->pShiftMix - PRE-HANG").postln;
cond.hang;

if (~pitchShiftMix_prev == 0, { //if current pShiftMix val == 0%
    randNum = rrand(1.0, 10.0).round(0.01);
    mixCase = case
    {randNum <= 4} { //set mix to 100%
        interpTime = rrand(0.25, 5.0); //("interpTime=" +
interpTime).postln;
        interpMix = Synth(\LinInterp_num, [out,
~pitchShiftMixValBus, \startVal, ~pitchShiftMix_prev, \stopVal, 1.0, \duration, interpTime] );
        ~pitchShiftMix_interp = true;
        ~updateGUI_pitchShiftMix.reset;
        AppClock.play(~updateGUI_pitchShiftMix);
        (interpTime + 0.1).yield;
        ~pitchShiftMix_interp = false;
        ~updateGUI_pitchShiftMix.stop;
        //pitchShiftMix_prev = 1; //<--- only implement this if its
decided to maintain current mixVal in env variable globally throughout code....would need to update
throughout...
    }
    {randNum > 4 && randNum <= 7} { //set mix to 50%
        interpTime = rrand(0.25, 5.0); //("interpTime=" +
interpTime).postln;
        interpMix = Synth(\LinInterp_num, [out,
~pitchShiftMixValBus, \startVal, ~pitchShiftMix_prev, \stopVal, 0.5, \duration, interpTime] );
        ~pitchShiftMix_interp = true;
        ~updateGUI_pitchShiftMix.reset;
        AppClock.play(~updateGUI_pitchShiftMix);
        (interpTime + 0.1).yield;
        ~pitchShiftMix_interp = false;
        ~updateGUI_pitchShiftMix.stop
    };
    //else if randNum > 7, leave mix at 0%
}, { //else if current pitchShiftMix > 0%
    randNum = rrand(1.0, 10.0).round(0.01);
    mixCase = case
    {randNum <= 4} { //set mix to 100%
        if (~pitchShiftMix_prev != 1, //if pShiftMix isn't already
100%, interp to 100%...
        {
            interpTime = rrand(0.25, 5.0); //
("interpTime=" + interpTime).postln;
            interpMix = Synth(\LinInterp_num, [out,
~pitchShiftMixValBus, \startVal, ~pitchShiftMix_prev, \stopVal, 1.0, \duration, interpTime] );
            ~pitchShiftMix_interp = true;
            ~updateGUI_pitchShiftMix.reset;

            AppClock.play(~updateGUI_pitchShiftMix);
            (interpTime + 0.1).yield;
            ~pitchShiftMix_interp = false;
            ~updateGUI_pitchShiftMix.stop
        },
        //,
        //("">>>>PshiftMix 100% selected. Already at
100%. No interp initiated...".postln; } //else if already 100%
    );
    }
    {randNum > 4 && randNum <= 7} { //set mix to 50%
        if (~pitchShiftMix_prev != 0.5, //if pShiftMix isn't already
50%, interp to 50%
        {
            interpTime = rrand(0.25, 5.0); //
("interpTime=" + interpTime).postln;
            interpMix = Synth(\LinInterp_num, [out,
~pitchShiftMixValBus, \startVal, ~pitchShiftMix_prev, \stopVal, 0.5, \duration, interpTime] );

```



```

~pitchShiftMix_interp = true;
~updateGUI_pitchShiftMix.reset;

AppClock.play(~updateGUI_pitchShiftMix);

(interpTime + 0.1).yield;
~pitchShiftMix_interp = false;
~updateGUI_pitchShiftMix.stop

//,
//{>>>PshiftMix 50% selected. Already at 50%.
No interp initiated...".postln;} //else if already 50%
);
}
{randNum > 7}{ //set mix to 0%
interpTime = rrand(0.25, 5.0); //("interpTime=" +
interpTime).postln;
interpMix = Synth(\LinInterp_num, [out,
~pitchShiftMixValBus, \startVal, ~pitchShiftMix_prev, \stopVal, 0, \duration, interpTime] );
~pitchShiftMix_interp = true;
~updateGUI_pitchShiftMix.reset;
AppClock.play(~updateGUI_pitchShiftMix);
(interpTime + 0.1).yield;
~pitchShiftMix_interp = false;
~updateGUI_pitchShiftMix.stop
};
});

time2nextRandomize = rrand(5, 15).round(0.01);
//time2nextRandomize = rrand(3, 6).round(0.01);
//("----> TIME 2 NEXT - PSHIFTMIX RAND =" +
time2nextRandomize).postln;
~time2NextRandom_pitchShiftMix = time2nextRandomize;
time2nextRandomize.yieldAndReset;
}

);

~randomRoutine_outputLevel = Routine.new ( { //randomly adjust main output level (crude anti-
feedback measure)
var time2nextRandomize, randNum, interpLevel, interpTime;
//("***OUTPUTlevl_randomRoutine***").postln;

~outMainLevelBus.get( { /////method derived from ~randomRoutine_rate and freq....
arg val;
~outMainLevVal_prev = val;
//("--OutputLvlRtn: CURRENTLVL =" + val).postln;
});
0.1.yield; //<<<<<works, but not ideal way...

if (~outMainLevVal_prev < 1.0, { //if outMainLevel is 0 or less than 1, SET LVL BACK TO 1
//("--CASE1-SET LVL TO 1: outLevelPrev=" + ~outMainLevVal_prev).postln;
randNum = rrand(1, 10);
//("randNum1=" + randNum + " | ").post;
if (randNum <= 4, //PICK INTERPTIME BASED ON RAND NUM
{interpTime = rrand(0.25, 0.75); /* ("interpTime=" + interpTime).postln; */ },
{interpTime = rrand(3.25, 6.0); /* ("interpTime=" + interpTime).postln; */ });

interpLevel = Synth(\LinInterp_num, [out, ~outMainLevelBus, \startVal,
~outMainLevVal_prev, \stopVal, 1.0, \duration, interpTime] );
~outMainLevVal_interp = true;
~updateGUI_outMainLevel.reset;
AppClock.play(~updateGUI_outMainLevel);

(interpTime + 0.1).yield; //<----probs not necessary to add 0.1 to accommodate Synth
interp.....

```

```

~outMainLevInterp = false;
~updateGUI_outMainLevel.stop;

randNum = rrand(1, 10);
//("randNum2=" + randNum + " | ").post;
//("---outLevel-randNum=" + randNum).postln;
if (randNum >= 8,
    {time2nextRandomize = rrand(8, 20); //80% of time, long time2nextRand
      //("time2Next=" + time2nextRandomize).postln;
    },
    {time2nextRandomize = rrand(0.25, 3).round(0.01); //20% of time short
time2nextRand
      //("time2Next=" + time2nextRandomize).postln;
    }
);
//("---outLevel-randomTime=" + time2nextRandomize).postln;
//time2nextRandomize.yieldAndReset;
}, { //else if val >= 1
//("--CASE2: outLevelprev=" + ~outMainLevVal_prev).postln;
randNum = rrand(1.0, 10.0);
//("randNum1=" + randNum + " | ").post;
//("---outLevel-randNum=" + randNum).postln;
if (randNum <= 8.8, { //88% of time keep level at 1
    //~outMainLevelBus.set(1);
    //~slider_outMainLevel.value_(1);
    time2nextRandomize = rrand(12, 18);
    //("--NO LEVEL CHANGE...time2next=" + time2nextRandomize).postln;
},
{ //12% of time set level to 0
    interpTime = rrand(0.1, 0.35).round(0.01);
    //("--SET LVL to 0...interpTime=" + interpTime).postln;
    interpLevel = Synth(\LinInterp_num, [out, ~outMainLevelBus, \startVal,
~outMainLevVal_prev, \stopVal, 0.0, \duration, interpTime] );
    ~outMainLevInterp = true;
    ~updateGUI_outMainLevel.reset;
    AppClock.play(~updateGUI_outMainLevel);

    (interpTime + 0.1).yield; //<----IS IT ALWAYS NECESSARY TO ADD 0.1 to
accommodate Synth interp?

    ~outMainLevInterp = false;
    ~updateGUI_outMainLevel.stop;

    /* //ORIG randomization to pick time2nextRandomize
    randNum = rrand(1, 10);
    if (randNum <= 4, {
        time2nextRandomize = rrand(3.0, 5.0).round(0.01);
    }, {
        time2nextRandomize = rrand(5.0, 10.0).round(0.01);
    });
    */
    time2nextRandomize = rrand(2.5, 4.0).round(0.01);
    //("--SET LVL TO 0 | time2nextRand=" + time2nextRandomize).postln;
});
//("---outLevel-randomTime=" + time2nextRandomize).postln;
//time2nextRandomize.yieldAndReset;
});
//("---outLevel-randomTime=" + time2nextRandomize).postln;
time2nextRandomize.yieldAndReset;
});
});

~randomRoutine_reverse = Routine.new ( {
    var time2nextRandomize, thisRandomNum, revState, cond = Condition.new;
    thisRandomNum = rrand(1.0, 10.0).round(0.01);
    //("randRoutine: randNum=" + thisRandomNum).postln;

```

```

~revRateBusMon.get({|vall
    revState = val.round(1);
    cond.unhang;
    //("---->RANDREV - GET - UNHANG").postln;
});
//("---->RANDREV - PRE-HANG").postln;
cond.hang;

if (revState > 0, { //if REVERSE disabled //reverse disabled when ~revRateBusMon == 1
    if (thisRandomNum <= 3.5, { //35% chance REVERSE enabled
        var revRate = ~pitchShiftValPrev_rate.midiratio * -1;
        ~rateBus.set(revRate);
        ~revRateBusMon.set(-1);
        ~button_rev.value_(1);
        thisRandomNum = rrand(1, 10);
        //("***thisRandNum-reroll" + thisRandomNum).postln;
        if (thisRandomNum <= 8,
            {time2nextRandomize = rrand(3.5, 6.0).round(0.01);},
            {time2nextRandomize = rrand(6.0, 10.0).round(0.01);});
        //("--> RAND REV: REV ENABLED").postln;
        //("----> RAND REV: time2NextRand=" + time2nextRandomize).postln;
        time2nextRandomize.yieldAndReset;
    }, { //71% chance REVERSE remains disabled
        time2nextRandomize = rrand(9, 20).round(0.01);
        //time2nextRandomize = rrand(3, 6).round(0.01);
        //("--> RAND REV: REV REMAINS DISABLED").postln;
        //("----> RAND REV: time2NextRand=" + time2nextRandomize).postln;
        time2nextRandomize.yieldAndReset;
    });
}, { //if REVERSE enabled
    if (thisRandomNum < 9, { //80% chance REVERSE disabled
        var currentRate = ~pitchShiftValPrev_rate.midiratio.abs; //abs just to ensure
it's positive rate value
        ~rateBus.set(currentRate);
        ~revRateBusMon.set(1);
        ~button_rev.value_(0);
        time2nextRandomize = rrand(9, 20).round(0.01);
        //time2nextRandomize = rrand(3, 6).round(0.01);
        //("--> RAND REV: REV DISABLED").postln;
        //("----> RAND REV: time2NextRand=" + time2nextRandomize).postln;
        time2nextRandomize.yieldAndReset;
    }, { //20% chance REVERSE remains enabled
        time2nextRandomize = rrand(3.0, 5.0).round(0.01);
        //("--> RAND REV: REV REMAINS ENABLED").postln;
        //("----> RAND REV: time2NextRand=" + time2nextRandomize).postln;
        time2nextRandomize.yieldAndReset;
    });
});
});

//<-----UPDATE this routine to be consistent w/ others (no conditionals based on GUI objects &
safer .get methods)
~randomRoutine_playHeadBehavior = Routine.new ( { //Randomize playHead behavior (Following or
Random)

//CONSIDER AUGMENTING EVENTUALLY to be aware of current rate and pitchShift values (so it can
avoid feedback if randomRate or randomPitch routines not enabled by disabling playRecOffset if this is
the case, for example) AAAANNDD ALSO BE AWARE OF previous and current playBehavior states to
more intelligently govern probability of transitioning to new state (e.g. if previously was at randomState,
increase the chances of reverting back to followPlayhead state upon next state change)

    var time2nextRandomize, thisRandomNum, selectedBehavior, behaviorCase, newPos,
offsetSec, minOffsetSec = 12, maxOffsetSec = 55;

```

```

        thisRandomNum = rand(1.0, 10.0);
        if (((~randPlayPos_flag == false) && (thisRandomNum <= 8)) || (~randPlayPos_flag == true),
{ //80% chance playHead will follow recHead if randomPos not previously active
        ~randPlayPos_flag = false; //reset flag to remember randomPos no longer active after
routine reset
        ~playHeadFollowRecHead.reset;
        AppClock.play(~playHeadFollowRecHead);
        time2nextRandomize = rand(12, 20);
        ("***FOLLOW RECHEAD behavior running for " + time2nextRandomize + "
secs...").postln;
        time2nextRandomize.yieldAndReset;
    }, { //else 25% chance playHead moved to random position...
        //("***RANDOM POSITION***").postln;
        ~randPlayPos_flag = true; //set flag to remember randomPos triggered after this
routine is reset
        ~playHeadFollowRecHead.stop;

        ~phaseBusMonRec.get( {
            arg recPos;
            ~lastRecPhasePos = recPos;
            //("recPos=" + recPos).post;
            //(" | lastRecPhasePos=" + ~lastRecPhasePos).postln;
        });

        ~phaseBusMonPly.get( {
            arg playPos;
            //~lastPlayPhasePos = playPos;
            ~lastPlayPhasePos = playPos;
            //("playPos=" + playPos).post;
            //(" | lastPlayPhasePos=" + ~lastPlayPhasePos).postln;
        });

        0.1.yield; //<----works, but not ideal use Condition.hang/unhang w/ .get or .getSync
instead

        offsetSec = rand(minOffsetSec, maxOffsetSec); //generate random offset value (in
seconds)
        //("offsetSec=" + offsetSec).postln;
        thisRandomNum = rand(1, 2);
        if (thisRandomNum == 1,
            { //50% chance playHead offset backwards in time
                //newPos = rand(0, ~lastPlayPhasePos);

                //("--BACKWARDS--").postln;
                newPos = ~lastPlayPhasePos - (offsetSec * s.sampleRate);
                if (newPos < 0, {newPos = (~audioBuf0.numFrames - 1) +
newPos;}); //wrap around if newPos initially less than 0
                //("lastPlayPhaseFrame=" + ~lastPlayPhasePos + " | lastPlaySec="
+ (~lastPlayPhasePos/s.sampleRate) ).postln;
                //("newPos=" + newPos + " | " + (newPos/s.sampleRate) +
"sec").postln;
            },
            { //50% chance playHead offset forwards in time
                //newPos = rand(~lastPlayPhasePos, ~audioBuf0.numFrames - 1);

                //("--FORWARDS--").postln;
                newPos = ~lastPlayPhasePos + (offsetSec * s.sampleRate);
                if (newPos > (~audioBuf0.numFrames - 1), {newPos = (newPos %
~audioBuf0.numFrames) - 1;}); //wrap around if newPos greater than buffer length

                if ( ((~lastRecPhasePos - newPos) <= (2 * s.sampleRate) ) &&
((~lastRecPhasePos - newPos) >= 0), {
                    //("---AVOIDING FEEDBACK---").postln;
                    //("lastRecHeadPos =" +
(~lastRecPhasePos/s.sampleRate) + " | ").post;

```

```

                                                                    //("newPos WAS: " + (newPos/s.sampleRate).round(0.01)
+ "sec").postln;
                                                                    newPos = ~lastRecPhasePos + (1.5 * s.sampleRate);
//reset newPos to 1.5 seconds AFTER recHead to avoid inducing rapid feedback (only would happen if
rate and pitch shift == 0)
                                                                    if (newPos > (~audioBuf0.numFrames - 1), {newPos =
(newPos % ~audioBuf0.numFrames) - 1;}); //wrap around if newPos greater than buffer length
                                                                    //("newPos IS NOW: " +
(newPos/s.sampleRate).round(0.01) + "sec").postln;
                                                                    }); //to avoid landing too close to recHead and causing rapid audio
feedback
                                                                    //("lastPlayPhaseFrame=" + ~lastPlayPhasePos + " | lastPlaySec="
+ (~lastPlayPhasePos/s.sampleRate) ).postln;
                                                                    //("newPos=" + newPos + " | " + (newPos/s.sampleRate) +
"sec").postln;
                                                                    }
                                                                    );
playHead) ~phasor_play.set(\resetPos, newPos, \trig, 1); //set new position of playPhase (move
time2nextRandomize = rrnd(4.5, 10.0).round(0.01);
("****RANDOM POSITION behavior running for " + time2nextRandomize + "
secs...").postln;
                                                                    time2nextRandomize.yieldAndReset;
                                                                    });

/*
behaviorCase = case //<-----REINCORPORATE
{~playBehavior == 0} { //"follow recHead" behavior
~playHeadFollowRecHead.reset;
AppClock.play(~playHeadFollowRecHead);
if (thisRandomNum <= 7.5, {
time2nextRandomize = rrnd(12.0, 25.0);
("****FOLLOW RECHEAD behavior running next " + time2nextRandomize + "
secs...").postln;
time2nextRandomize.yieldAndReset;
}, {
//////////
});
}
{ } { //"random play" behavior
}
{ } { //"random loop" behavior
}; */
~playHeadFollowRecHead = Routine.new( { //set playhead based on current RecHead-PlayHead offset
AND playback Rate value --- used in ~randomRoutine_playHeadBehavior
//***playRecOffsetIsGoing** ".post;
//("revFlag=" + ~reverseFlag).postln;
if ((~pitchInterp_rate == false) && (~button_rev.value == 0) /*&& (~pitchInterp_freq == false) */,
{ //<---ADDED reverse & pitchInterp detection---DON'T USE GUI OBJECT FOR CONDITION, POLL
CONTROL BUS<-----
var currentTimeOffset, newOffsetPos, newOffsetAmount, pitchCase;

~phaseBusMonRec.get( {
arg recPos;
~lastRecPhasePos = recPos;
//("recPos="+recPos).post;
//(" | lastRecPhasePos=" + ~lastRecPhasePos).postln;
});

~phaseBusMonPly.get( {
arg playPos;

```

```

        //~lastPlayPhasePos = playPos;
        //~lastPlayPhasePos = playPos;
        //~("playPos=" + playPos).post;
        //~(" | lastPlayPhasePos=" + ~lastPlayPhasePos).postln;
    });

    0.1.yield; //<--not sure if needed...not ideal and crude

    if (~lastRecPhasePos < ~lastPlayPhasePos, //measure distance between rec and
play - if rec's wrapped around to start of buffr before play
        {currentTimeOffset = ( (~playOutFrame - ~playInFrame) -
(~lastPlayPhasePos - ~lastRecPhasePos) ) / (s.sampleRate); },
        {currentTimeOffset = (~lastRecPhasePos - ~lastPlayPhasePos)/
(s.sampleRate); } //else measure offset if no wrap around has occurred (will happen most of the time)
    );
    currentTimeOffset = currentTimeOffset.round(0.01);
    //~("!!!currentTimeOffset=" + currentTimeOffset).postln;

    pitchCase = case
    {~pitchShiftValTarget_rate > 0} {
        if ( (currentTimeOffset <= 0.3) || (currentTimeOffset >= 6.5), { //MAINTAIN
fixed PlayHead offset range BEHIND RecHead if ~pitchShiftValTarget_rate > 0 --- not too far behind,
never surpassing
            newOffsetAmount = rrand(2.7, 4.0); //generate random offset
amount (in seconds)
            newOffsetPos = ~lastRecPhasePos - ((newOffsetAmount *
s.sampleRate).round(1)); //deduce new playPhase position based on newOffsetAmount (converted to
frames)

            /*if (newOffsetPos < 0, //orig simple safe driving
            {newOffsetPos = (~audioBuf0.numFrames - 1) +
newOffsetPos;} //wrap around buffer if negative position generated
            ); */

            if ( (~playInFrame == 0) && (newOffsetPos < 0), { //safe driving:
case 1
                newOffsetPos = ~playOutFrame + newOffsetPos; //wrap
around buffer if negative position generated
                if (newOffsetPos < 0, {
                    /*newOffsetPos = 0;*/ //think this could potentially
cause issues if current recPos = 0 too...
                    newOffsetPos = ~lastRecPhasePos + 1000;});
//to avoid never ending loop if result is still < 0
                });

            if ( (~playInFrame > 0) && (newOffsetPos < ~playInFrame), { //safe
driving: case 2
                newOffsetPos = newOffsetPos + (~playOutFrame -
~playInFrame);
                if (newOffsetPos < ~playInFrame, {newOffsetPos =
~lastRecPhasePos + 1000;}); //to avoid never-ending loop
                });

            ~phasor_play.set(\resetPos, newOffsetPos, \trig, 1); //set new
position of playPhase (move playHead)
            });
    }
    {~pitchShiftValTarget_rate <= 0} {
        if ( (currentTimeOffset >= 6.0) || (currentTimeOffset <= 0.3),
        { //MAINTAIN fixed PlayHead offset range BEHIND RecHead if
~pitchShiftValTarget_rate <= 0 --- not too far behind, never surpassing
            newOffsetAmount = rrand(0.5, 2.75); //generate random offset
amount (in seconds)

```

```

newOffsetPos = ~lastRecPhasePos - ((newOffsetAmount *
s.sampleRate).round(1)); //deduce new playPhase position based on newOffsetAmount (convert to
frames)

if ( (~playInFrame == 0) && (newOffsetPos < 0), { //safe driving:
case 1
newOffsetPos = ~playOutFrame + newOffsetPos; //wrap
around buffer if negative position generated
if (newOffsetPos < 0, {
/*newOffsetPos = 0;*/ //think this could potentially
cause issues if current recPos = 0 too...
newOffsetPos = ~lastRecPhasePos + 1000;});
//to avoid never ending loop if result is still < 0
});

if ( (~playInFrame > 0) && (newOffsetPos < ~playInFrame), { //safe
driving: case 2
newOffsetPos = newOffsetPos + (~playOutFrame -
~playInFrame);
if (newOffsetPos < ~playInFrame, {newOffsetPos =
~lastRecPhasePos + 1000;}); //to avoid never-ending loop
});

~phasor_play.set(\resetPos, newOffsetPos, \trig, 1); //set new
position of playPhase (move playHead)
});
});
2.0.yieldAndReset;
});
//-----

//----- AUTO REC / PLAY ROUTINES -----
~autoRec_random = Routine.new( {
if (~recSamp_input == true,
{ //STOP RECORDING / WRITING AUDIO INPUT TO FILE
//var yieldTime = rrand(5, 8).round(0.01);
var yieldTime = rrand(8, 30).round(0.01);
~recSamp_input = false;
{~button_recln.value = 0;}.defer;

~writeIn2Disk.free; //stop recording to disk
~audioBuf_inMainWrite.close; //close the buffer to finish writing the sound
file (won't be readable otherwise)
~audioBuf_inMainWrite.free;
("AutoRecRand -- WROTE Samp-AUDIO-IN_" ++
~fileAudio_writeln_nameNum).postln;
~fileAudio_writeln_nameNum = ~fileAudio_writeln_nameNum + 1;
//increment file name suffix in prep for next rec file
if (~fileAudio_writeln_nameNum > ~fileAudio_writeln_maxNum, //cap max
file count...
{~fileAudio_writeln_nameNum = 0;});
});

("AutoRecRand -- Next Time to RandRec = " ++ yieldTime).postln;
yieldTime.yieldAndReset;
},
//else if ~recSamp_input == false
if (~playSamp == false, {
if (~bufRecorder.isRunning, { //BEGIN RECORDING AUDIO INPUT
TO FILE
var recTime = rrand(~minRecDur,
~maxRecDur).round(0.01);

```

```

~recSamp_input = true;
{~button_recIn.value = 1;}.defer;
~audioBuf_inMainWrite = Buffer.alloc(s, 65536, 1);
//allocate a disk i/o buffer (min rec frame size is 65536)

~audioBuf_inMainWrite.write((~fileAudio_writeln_lastPath++~fileAudio_writeln_nameNum+
+".wav").standardizePath, "wav", ~sampBitDepth, 0, 0, true); // create an output file for this buffer,
leaving it "open" (not recording yet, just created file w/ format header)
~writeln2Disk = Synth.tail(nil, \write2Disk, [\buf,
~audioBuf_inMainWrite, \inBus, ~inputSendBus] ); //record audio from rec playhead (bufRead synth
sends audio input to ~inputSendBus), and write an audio file to hard disk
("AutoRecRand -- WRITING AUDIO-IN SAMP_" ++
~fileAudio_writeln_nameNum).postln;
("AutoRecRand: -- recTime = " ++ recTime).postln;
("---").postln;
recTime.yieldAndReset;
}, { //else if ~bufRecorder not running, don't record anything
"AutoRecRand: -- NO RECORDING - BUF REC NOT
ACTIVE".postln;
8.yieldAndReset;
});
}, { //else if (~playSamp == true)
5.yieldAndReset;
});
}
);
});

~autoPlay_random = Routine.new( { //<-----OLD I LIL BUGGY, almost there.... sample playback buffer
getting lost somehow???
if (~playSamp == true,
{ //if already playing sound file, we can assume it's completed by the time this routine
has stopped yielding based on commands when ~playSamp == false (see below code in 'else' of this
conditional)...
var time2yield, interpSynth;
~playSamp = false;

if (~bufPlayer.isRunning, {
interpSynth = Synth(\LinInterp_num, [startVal, 0.0, \stopVal, 1.0,
\duration, 0.2, \out, ~inputStreamLevelBus]); //turn inStreamLevel back to 1.0 in 0.2 sec (++++~startVal
should probs = variable storing current value of inputStreamLevel++++)
{~slider_inFlow1_level.value = 1;}.defer; //update GUI
{~numBox_inFlow1_level.value = 1;}.defer; //update GUI
//0.2.yield; //not necessary, since sound's stopped by this point...
});

~audioBuf_play0.free;
{~button_playSamp.value = 0;}.defer; //update GUI button

time2yield = rrand(3, 5).round(0.01);
//time2yield = rrand(12, 35); //<-----TIME RANGE - TRIGGER RATE (IN
SEC).....

("AutoPlayRand: -- time2nextPlay= " ++ time2yield ++ " sec").postln;
time2yield.yieldAndReset;
},
{ //if not playing a soundFile, ~playSamp == false
if (~recSamp_input == false, //only play back sounds if not currently
recording...
{
if (~fileAudio_writeln_nameNum >=
~minNumFilesB4Playback, //if min # of files have been written
{ //.....currently triggers random sample every time
routine is played...DEVELOP....

```



```

TRIGGERING SAMPLE PLAYBACK GOES HERE****-----
listening ugen output control busses...

readFileFunc, randFilePick, duration;
duration;

buffer audio, silence it for audio sample playback
level val from ~inputStreamLevelBus and assign to local variable used for \startVal arg in interpSynth
below...
Synth(\linInterp_num, [\startVal, 1.0, \stopVal, 0.0, \duration, 0.2, \out, ~inputStreamLevelBus]);
//silence input stream in 0.2 sec (**startVal should probs = variable storing current value of
inputStreamLevel**)
0.2.yield").postln;
0.2.yield").postln;
0.2.yield").postln;

~bufPlayer.isRunning").postln;

(~fileAudio_writeIn_nameNum - 1)); //pick a random file to play (needs to be "current nameNum counter
val - 1" b/c current counter value doesn't have a file associated w/ it yet due to counter incrementing
after a new file has been written...)

file picking behaviors or mappings here****

function for audio file playback ---- vvvvvv
+randFilePick++".wav").standardizePath;

action: {arg buffer;
buffer.numFrames/s.sampleRate;
buffer.numFrames/buffer.sampleRate;
Synth(\bufRead_noLoop, [\buf, buffer, \amp, ~fileOutputLevelBus.asMap, \out, ~outMainBus]);
0);

(~fileAudio_writeIn_lastPath++randFilePick++".wav").standardizePath).postln;

//update GUI

//<---***CONDITION CHECKING
//---e.g. routines checking machine

//PLAYBACK AUDIO FILE FROM DISK
//var readDiskSynth, interpSynth,
var interpSynth, randFilePick, fileDir,

~playSamp = true; //set flag
if (~bufPlayer.isRunning, { //if playing
//get current signal flow/chain
interpSynth =
{~slider_inFlow1_level.value =
{~numBox_inFlow1_level.value
("~bufPlayer.isRunning -
0.2.yield;
});
("--after checking

randFilePick = rrand(0,
val - 1)); //pick a random file to play (needs to be "current nameNum counter
val - 1" b/c current counter value doesn't have a file associated w/ it yet due to counter incrementing
after a new file has been written...)

//<---***could alternatively insert other

//vvvvvv ---- make this into a reusable
fileDir = (~fileAudio_writeIn_lastPath+

//PLAY SOUND FILE from disk
~audioBuf_play0 = Buffer.read(s, fileDir,

//duration =
duration =
~samp0 =
//NodeWatcher.register(~samp
0);

("PLAYING BACK FILE:" +
(~fileAudio_writeIn_lastPath++randFilePick++".wav").standardizePath).postln;

});
{~button_playSamp.value = 1;}.defer;

```

```

duration).println;                                ("Pre-yield samp Dur = " ++
                                                    0.2.yield; //<---UNFORTUNATE YIELD
needed to update duration variable (probs doesn't need to be that lon, thought short yield times was
causing a "Buffer UGen: buffer data" error, but testing seems to point elsewhere ...
                                                    ("Post-yield samp Dur = " ++
duration).println;
                                                    //duration.yieldAndReset; //<---might not
be necessary to add 0.02 sec, experiment
                                                    (duration +
0.2).round(0.01).yieldAndReset; //<---probs not necessary to add 0.2 sec, experiment
                                                    //----- ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
-----
                                                    },
                                                    {15.yieldAndReset;} //<----- wait if not
enough files have been written yet
                                                    );
                                                    },
                                                    {5.yieldAndReset;} //else if (~recSoundInput == true)
                                                    );
                                                    }
                                                    );
                                                    );
};
);

```

File – “05_InitSM”:

```

//Sonic Mirror - 05_InitSM:

//== CONFIGURE & INITIALIZE SM ==

(
~init_SonicMirror = {

////GLOBAL VARIABLES
~sampBitDepth = "int24"; //alt = "int16" (specify bit depth for all samples recorded by SM)
~plotterEnabled = true; //set to false for RPi (or until I optimize plotter code, it's not efficient now.)

~audioBuf0_dur = 180.0; //<---changing here fine, but changing bufDur w/ GUI causes errors, will fix
bug later...
~audioBuf0 = Buffer.alloc(s, s.sampleRate * ~audioBuf0_dur, 1);

~fileAudio_writeIn_lastPath = "~/Music/SCreos/SncMir/SncMir-In_"; //defines file write path
~fileAudio_writeOut_lastPath = "~/Music/SCreos/SncMir/SncMir-Out_"; //defines file write path
~minNumFilesB4Playback = 1; // file # threshold before file playback routine commences
~fileAudio_writeIn_maxNum = 30; //max # of audio samples written per session (wrap-around &
overwrite limit - inclusive of 0)
~fileAudio_writeOut_maxNum = 30; //max # of audio out samples written per session (wrap-around &
overwrite limit)
~fileAudio_writeIn_maxNum_exceeded = false;

~minRecDur = 1.0; //<-----min sec per recorded sample
~maxRecDur = 30.0; //<-----max sec per recorded sample (shorten or don't use w/ autoRec_random)

~fileAudio_writeOut_nameNum = 0; //counter for writing/reading audio file names
~fileAudio_writeIn_nameNum = 0; //counter for writing/reading audio file name ORIG <--- (moved to top)
~fileAudio_writeIn_lastFile = (~fileAudio_writeIn_lastPath ++ ~fileAudio_writeIn_nameNum ++
".wav").standardizePath; //<--ORIG

~fileAudio_writeIn_lastFileDur; //<-----NEW (not currently being used)
~fileAudio_lastPlay; //<---NEW (not needed if using below array though)
~fileHistory_rec = Array.newClear(~fileAudio_writeIn_maxNum); //filename history of all samples
recorded <--NEW

```

```

~fileHistory_play = Array.newClear(~fileAudio_writeIn_maxNum); //filename history of all samples
played <--NEW

~soundMemory = []; //<-----NEW (init)
//~soundMemory = [{"sound1Name", sound1timeIn, sound1timeOut, sound1dur, sus / transient, mean
loudness, pitchiness, mean pitch}, {"sound2Name", sound2timeIn, sound2timeOut, ....}, {"sound3Name",
...}, {"soundMaxNumName", ...}];

~delayTime = 1.23;
~play2recOffset; //displacement of playhead pos from rec pos (in frames/samples)

~mode_randProcLiveBuf = false; //<---NEW mode flag
~mode_autoRecPlay = false; //<---NEW mode flag

~lastPlayPhasePos = 0; //for resume functionality - stores last pos (frame) of play head before playback
disabled
~lastRecPhasePos = 0; //for resume functionality - stores last pos (frame) of rec head before recording
disabled
~playPosLatchTrig = false;
~recPosLatchTrig = false;
~syncRecPlayOffset = false;

~pitchShiftValPrev_rate = 0;
~pitchShiftValTarget_rate = 0;
~pitchInterp_rate = false;

~pitchShiftValPrev_freq = 0;
~pitchShiftValTarget_freq = 0;
~pitchInterp_freq = false;

~outMainLevVal_prev; //<--
~outMainLevVal_target; //<--not currently being used...could be useful in future though...
~outMainLevInterp = false; //<--not currently being monitored...

~reverseFlag = false; //currently using ~button_rev.value as flag, but is that a good long-term idea????
~randPlayPos_flag = false;

~time2NextRandom_rate = 0;
~time2NextRandom_freq = 0;
~time2NextRandom_pitchShiftMix = 0;

~pitchShiftMix_prev = 0;
~pitchShiftMix_target = 0; //<- not using right now
~pitchShiftMix_prevChangeTime = 0; //<-not being used
~pitchShiftMix_interp = false;

~playInFrame = 0; //<--- consider renaming
~playOutFrame = ~audioBuf0_dur * s.sampleRate - 1; //~audioBuf0.numFrames - 1; (won't work unless
~audioBuf0's already been allocated...)

~recInFrame = 0;
~recOutFrame = ~audioBuf0_dur * s.sampleRate - 1;
//-----

//(RESET BUS VALUES FOR INITIALIZATION OF SKETCH)
~playInPosBus.set(0);
~playOutPosBus.set(~playOutFrame);
~recInPosBus.set(0);
~recOutPosBus.set(~recOutFrame);

~rateBus.set(1);
~pitchBus.set(1);
~pitchShiftMixValBus.set(0); //<---

```

```

~revRateBusMon.set(1); //<---

~inputLevelBus.set(1);
~play1LevelBus.set(1); //<--NO SLIDER CREATED YET!!!!....
~inputStreamLevelBus.set(1); //<---SET TO -6db on INIT? <-----
~fileOutputLevelBus.set(1);
~outMainLevelBus.set(1);

~outMainPeakFreq_init = 1200;
~outMainPeakFreq.set(~outMainPeakFreq_init);
~outMainPeakQ_init = 4;
~outMainPeakQ.set(~outMainPeakQ_init); //technically reciprocal of Q...
~outMainPeakGain_init = -3;
~outMainPeakGain.set(~outMainPeakGain_init); //in db

~loCutVal_in = 110;
~loCutVal_out = 110;
~loCutBus_in.set(~loCutVal_in);
~loCutBus_out.set(~loCutVal_out);

~comp1thresh_init = -6.dbamp;
~comp1thresh.set(~comp1thresh_init); //<---NEW
~comp1ratio_init = 2;
~comp1ratio.set(~comp1ratio_init); //<---NEW
~comp1attack_init = 0.01;
~comp1attack.set(~comp1attack_init); //in seconds <---NEW
~comp1release_init = 0.3;
~comp1release.set(~comp1release_init); //in seconds <---NEW
~comp1gain_init = 1;
~comp1gain.set(~comp1gain_init); //<---NEW

~phaseBusMonPly.set(0);
~phaseBusMonRec.set(0);
//-----

// ----- SET UP SOME INITIAL SYNTHS -----
~mainOut = Synth.tail(s, \mainOut, [in, ~outMainBus, \loCut, ~loCutBus_out.asMap, \loCutQ, 2, \amp,
~outMainLevelBus.asMap, \peak, ~outMainPeakFreq.asMap, \peakQ, ~outMainPeakQ.asMap,
\peakGain, ~outMainPeakGain.asMap]);

~stream1 = Synth(\subMix, [in, ~stream1sub, \out, ~outMainBus]).moveBefore(~mainOut); //<---NEW
~stream2 = Synth(\subMix, [in, ~stream2sub, \out, ~outMainBus]).moveBefore(~mainOut); //<---NEW

//-----

////
Window.closeAll;
//w = Window.new("SonicMirror", Rect.new(250, 330, 780, 405))
w = Window.new("SonicMirror", Rect.new(335, 330, 818, 380), resizable: false)
.front
.alwaysOnTop_(true);

w.onClose_{s.freeAll; Buffer.freeAll(s); CmdPeriod.clearClocks; CmdPeriod.run;}; // clean up when
window is closed

~view_main = CompositeView(w, Rect(3, 3, 819, 380));
~view_MainLevels = CompositeView(~view_main, Rect(625, 26, 135, 147));
~view_comp1 = CompositeView(~view_main, Rect(763, 28, 45, 100));
~view_altLevels = CompositeView(~view_main, Rect(693, 26, 68, 105));
~view_bufConfig = CompositeView(~view_main, Rect(625, 0, 160, 25));
~view_bufPlot = CompositeView(~view_main, Rect(0, 0, 620,
170)).background_(Color.gray).resize_(1);
~view_transport = CompositeView(~view_main, Rect(0, 172, 788, 95));
~view_perfButtons = CompositeView(~view_main, Rect(5, 268, 308, 52));
~view_sampButtons = CompositeView(~view_main, Rect(5, 320, 250, 45));

```

```

~view_procControls1 = CompositeView(~view_main, Rect(317, 268, 460, 65));

//---
~spec_slider_dB = ControlSpec(minval: 0.ampdb, maxval: 1.ampdb, warp: \db, default: 0, units:
"dB"); //<-- NEW

//----- Main In/Out Controls (Levels & Lo-Cut) -----

//Main Input Level --

~slider_inMainLevel = Slider(~view_MainLevels, Rect(8, 2, 15, 90)) //<---NEW
.background_(Color(0.9, 0.0, 0.0, 0.2))
//.value_(1.0)
.value_(~spec_slider_dB.unmap(0)) //<--
.action_({|obj|
    var mapVal, val = obj.value.round(0.001);
    mapVal = ~spec_slider_dB.map(val).round(0.01); //convert linear amp to db scaling
    ~inputLevelBus.set(mapVal.dbamp.round(0.001)); //set level control bus value
    ~numBox_inMainLevel.value_(mapVal); //update numBox
});

~numBox_inMainLevel = NumberBox(~view_MainLevels, Rect(2, 92, 30, 15)) //<---NEW
.value_(0.0)
.clipLo_(-inf)
.clipHi_(0.0)
.decimals_(1)
.scroll_step_(0.1)
.font_(Font("Monaco", 8) )
.align_(\center)
.action_({|obj|
    var mapVal, val = obj.value.round(0.1);
    mapVal = ~spec_slider_db.unmap(val); //convert db num into linear amp scaling
    ~inputLevelBus.set(mapVal.round(0.001));
    ~slider_inMainLevel.value_(mapVal.round(0.01));
});

~label_inMainLevel = StaticText(~view_MainLevels, Rect(10, 103, 12, 20) )
.string_("IN")
.font_(Font("Monaco", 10) )
.align_(\center);

//Main Out Level --
~slider_outMainLevel = Slider(~view_MainLevels, Rect(40, 2, 15, 90) )
.background_(Color(0.0, 0.8, 0.2, 0.01))
//.value_(1.0)
.value_(~spec_slider_dB.unmap(0)) //<--
.action_( {|obj|
    var mapVal, val = obj.value.round(0.001);
    mapVal = ~spec_slider_dB.map(val).round(0.01); //convert linear amp to db scaling
    ~outMainLevelBus.set(mapVal.dbamp.round(0.001)); //set level control bus value
    ~numBox_outMainLevel.value_(mapVal); //update numBox
});

~numBox_outMainLevel = NumberBox(~view_MainLevels, Rect(34, 92, 30, 15))
.value_(0.0)
.clipLo_(-inf)
.clipHi_(0.0)
.decimals_(1)
.scroll_step_(0.01)
.font_(Font("Monaco", 8) )
.align_(\center)
.action_( {|obj|
    var mapVal, val = obj.value.round(0.1);
    mapVal = ~spec_slider_db.unmap(val); //convert db num into linear amp scaling
    ~outMainLevelBus.set(mapVal.round(0.001));
});

```

```

    ~slider_outMainLevel.value_(mapVal.round(0.01));
});

~label_outMainLevel = StaticText(~view_MainLevels, Rect(37, 103, 20, 20) )
.string_("OUT")
.font_(Font("Monaco", 10) )
.align_(\center);

//Main In / Out Lo-Cuts --
~numBox_inMainLoCut = NumberBox(~view_MainLevels, Rect(2, 120, 30, 14))//<---
.value_(~loCutVal_in)
.clipLo_(20)
.clipHi_(20000)
.font_(Font("Monaco", 8) )
.decimals_(1)
.action_( {
    arg obj;
    ~loCutVal_in = obj.value.round(0.01);
    ~loCutBus_in.set(~loCutVal_in);
    //~loCutBus_in.setSynchronous([~loCutVal_in]); //??????? for synchronous control? is this
better?
    ("***LoCutFreq_in=" ++ ~loCutVal_in ++ "Hz").postln;
});

~numBox_outMainLoCut= NumberBox(~view_MainLevels, Rect(32, 120, 30, 14))//<---
.value_(~loCutVal_out)
.clipLo_(20)
.clipHi_(20000)
.font_(Font("Monaco", 8) )
.decimals_(1)
.action_( {
    arg obj;
    ~loCutVal_out = obj.value.round(0.01);
    ~loCutBus_out.set(~loCutVal_out);
    //~loCutBus_out.setSynchronous([~loCutVal_out]); //??????? for synchronous control? is this
better?
    ("***LoCutFreq_out=" ++ ~loCutVal_out ++ "Hz").postln;
});

~label_inoutMainLoCut = StaticText(~view_MainLevels, Rect(8, 130, 48, 20) )//<---
.string_("LoCut-Hz")
.font_(Font("Monaco", 10) )
.align_(\left);

//Master Out Peak EQ --
~label_peakEQ = StaticText(~view_MainLevels, Rect(101, 106, 25, 38) )//<---
.string_("db
Peak
Q")
.font_(Font("Monaco", 10) )
.align_(\left);

~numBox_outMainPeak= NumberBox(~view_MainLevels, Rect(68, 118, 30, 14))//<--- NEW
.value_(~outMainPeakFreq_init)
.clipLo_(40)
.clipHi_(15000)
.font_(Font("Monaco", 8) )
.decimals_(0)
.align_(\center)
.action_({
    arg obj;
    ~outMainPeakFreq.setSynchronous(obj.value.round(1));
});

~numBox_outMainPeakGain= NumberBox(~view_MainLevels, Rect(65, 104, 35, 14))//<--- NEW

```

```

.value_(~outMainPeakGain_init)
.clipLo_(-inf)
.clipHi_(12)
.font_(Font("Monaco", 8) )
.decimals_(2)
.align_(\center)
.action_({
    arg obj;
    ~outMainPeakGain.setSynchronous(obj.value.round(0.01));
});

~numBox_outMainPeakQ = NumberBox(~view_MainLevels, Rect(68, 132, 30, 14))
.value_(~outMainPeakQ_init)
.clipLo_(0.1)
.clipHi_(100)
.font_(Font("Monaco", 8) )
.decimals_(1)
.align_(\center)
.action_( {
    arg obj;
    ~outMainPeakQ.setSynchronous(obj.value.round(0.1));
} );

//Master Out Compressor --
~numBox_comp1thresh = NumberBox(~view_comp1, Rect(1, 1, 30, 14))//<--- NEW
.value_(~comp1thresh_init.ampdb.round(0.1)) //display db
.clipLo_(-inf)
.clipHi_(0)
.font_(Font("Monaco", 8) )
.scroll_step_(0.1)
.decimals_(1)
.align_(\center)
.action_({|obj|
    ~comp1thresh.setSynchronous(obj.value.round(0.1).dbamp); //convert from db to linear amp
before set cont bus
});

~numBox_comp1ratio = NumberBox(~view_comp1, Rect(1, 15, 30, 14))//<--- NEW
.value_(~comp1ratio_init)
.clipLo_(1)
.clipHi_(20)
.font_(Font("Monaco", 8) )
.scroll_step_(0.1)
.decimals_(1)
.align_(\center)
.action_({|obj|
    ~comp1ratio.setSynchronous(obj.value.round(0.1));
});

~numBox_comp1attack = NumberBox(~view_comp1, Rect(1, 29, 30, 14))//<--- NEW
.value_((~comp1attack_init * 1000).round(1)) //displaying ms
.clipLo_(0)
.clipHi_(2000)
.font_(Font("Monaco", 8) )
.scroll_step_(1)
.decimals_(0)
.align_(\center)
.action_({|obj|
    var val = (obj.value/1000).round(0.001); //convert from ms to seconds
    ~comp1attack.setSynchronous(val);
});

~numBox_comp1release = NumberBox(~view_comp1, Rect(1, 43, 30, 14))//<--- NEW
.value_((~comp1release_init * 1000).round(1)) //displaying ms

```

```

.clipLo_(0)
.clipHi_(2000)
.font_(Font("Monaco", 8) )
.scroll_step_(1)
.decimals_(0)
.align_(\center)
.action_({|obj|
    var val = (obj.value/1000).round(0.001); //convert from ms to seconds
    ~comp1release.setSynchronous(val);
});

~numBox_comp1gain = NumberBox(~view_comp1, Rect(1, 57, 30, 14))//<--- NEW
.value_(~comp1gain_init.ampdb.round(0.1)) //display db
.clipLo_(-inf)
.clipHi_(6)
.font_(Font("Monaco", 8) )
.scroll_step_(0.5)
.decimals_(1)
.align_(\center)
.action_({|obj|
    var val = obj.value.round(0.1);
    if (val <= -80, {
        val = -inf;
        ~numBox_comp1gain.value = -inf;
    });
    ~comp1gain.setSynchronous(val.dbamp); //convert from db to linear amp before set cont bus
});

~label_comp1 = StaticText(~view_comp1, Rect(32, 1, 25, 68) )//<---
.string_("TH
:1
AT
RL
dB")
.font_(Font("Monaco", 10) )
.align_(\left);

~button_comp1 = Button(~view_comp1, Rect(0, 72, 33, 17)) //randomize rate and pitchShift over time
.states_( [
    ["COMP", Color.black, Color(0.7, 0.3, 0.7, 0.3)] ,
    ["COMP", Color.white, Color(0.8, 0.3, 0.8)]
] )
.font_(Font("Monaco", 12) )
.action_( {
    arg obj;
    if (
        obj.value == 1,
        {
            ~comp1 = Synth(\comp1, [\bus, ~outMainBus, \thresh,
~comp1thresh.asMap, \ratio, ~comp1ratio.asMap, \attack, ~comp1attack.asMap, \release,
~comp1release.asMap, \amp, ~comp1gain.asMap]).moveBefore(~mainOut);
        },
        {
            ~comp1.free;
        }
    );
} )
.value_(0);

//-----

//---- AUDIO BUFFER -----

~button_audioBuf_0 = Button(~view_bufConfig, Rect(0, 0, 65, 25)) //Activate / Deactivate audio buffer
.states_( [

```



```

["BUF_ACT", Color.black, Color.gray(0.8)] ,
["BUF_ACT", Color.white, Color(0.8,0.8, 0)]
] )
.value_(1)
.font_(Font("Monaco", 14) )
.action_( {
    arg obj;
    if (
        obj.value == 1,
        { //allocate(create) buffer w/ duration = bufferSize seconds
            ~audioBuf0 = Buffer.alloc(s, s.sampleRate * ~audioBuf0_dur, 1);
            ~recOutFrame = ~audioBuf0_dur * s.sampleRate - 1;
            ~recOutPosBus.set(~recOutFrame);
            ~playOutFrame = ~audioBuf0_dur * s.sampleRate - 1;
            ~playOutPosBus.set(~playOutFrame);
        }, //else
        { ~audioBuf0.close; ~audioBuf0.free;} //free(erase) buffer
    );
});

~numBox_audioBuf0_dur = NumberBox(~view_bufConfig, Rect(67, 0, 65, 25))
.value_(~audioBuf0_dur)
.clipLo_(10) //smallest possible length of buffer (in seconds)
.clipHi_(300) //longest possible length of buffer (in seconds)
.font_(Font("Monaco", 14) )
.decimals_(2)
.action_( {
    arg obj;
    ~audioBuf0_dur = obj.value.round(0.01);
    ("***Buffer Length=" + ~audioBuf0_dur + "seconds.").postln;
});

~label_audioBuf0_dur = StaticText(~view_bufConfig, Rect(134, 2, 200, 20) )
.string_("DUR
(sec)")
.font_(Font("Monaco", 8) )
.align_(\left);

//-----

//----- Soundfile Rec / Playback -----

//--put these variables back w/ the rest of env variables above at some point--
~playBehavior = 0; //stores state of playback behavior, not currently being used....

~currentSound = "silence"; //state of current heard sound (0 = silence, 1 = unpitched sound, 2 = pitched
sound...?)
~currentScene = "quiet"; //sound scene index (0 = quiet, 1 = sparse transient sounds, 2 = drone, 3 =
noise.....?)

~playSamp = false; //<--- flag: playing audio file from disk
~playAudify = false; //flag: audify sound routine active... <---NEW
~trigAudify = false; //flag: tracks whether or not audifyScene routine just triggered... <---NEW

//---
~listenDur_noiseFloor = 5.0; //<---not currently being used....
/*~senseNoiseFloor = Routine.new( {
    //sets ~loudnessThresh
}); */

//~hearFeedback = false;
//~fdbackDetect = false;
~input_loud = 0; //consider integer scale: 0 = < min thresh | 1 = > lowThresh, < uprThresh, 2 = >
uprThres
~input_loudSus = false;

```

```

~listen_input_loudness = false; //flag indicating activity of loudness monitoring function (active /
disabled)

~hearSound = false;
~listening = false; //flag indicates audio analysis activity

~recSamp_input = false;
~recSamp_output = false;
~recSamp_input_startFrame = 0;
~recSamp_input_stopFrame = 0;

//----- env variables and functions (routines) for ~autoRec_soundTrigAuto

~loudnessThresh = 5; //in sones - used by ~autoRec_soundTrigAuto routine
~pitchThresh = 0.55; //pitch ratio measure - used by ~autoRec_soundTrigAuto routine

~autoRecPlay_state = 0; //0 == only trigger auto rec, 1 == only trigger auto play

~preRecDur = 0.2; //amount of time to sample from buffer prior to start of ~autoRec
~postRecSnip = -0.3; //amount of time to subtract from total dur of sampled sound (to cut silence at end
of sample after ~autoRec finished and ~listenDur_intraSound pads end...)

~listenDur_intraSound = 0.8; //<----amount of time (sec) to continue listening for continuation of
contiguous sound object between gaps of silence (e.g. separation threshold for dividing sound objects)
ADJUSTING THIS PARAM INCREASES / DECREASES AMOUNT OF SOUND SAMPLE
FRAGMENTATION / SENSITIVITY TO DISCRETE TRANSIENT SOUND EVENTS.....
~listenDur_soundPatience = 10; // time to wait during autoRecMode b4 playing sample if no sound's
heard

~time_listenTimeOut; //time to wait until when testing for cessation of sound event
~time_recTimeOut; //maximum time to record until (to avoid runaway recordings and trolls)
~time_minRec;

~time_lastRec_start;
~time_lastRec_stop;
~time_lastPlay_start;
~time_lastPlay_stop;

~autoRec_rate = 0.1; //finest timing resolution of ~autoRec_soundTrig routine (NOT BEING USED)
//~yieldTime_autoRec = 0.1; //val to be reassigned as routine runs... (OLD
APPROACH....ABANDONING THIS)

~autoRecStartTrig = true; //flag triggered when ~autoRec routines initiated (used for debounce time
when triggered via button press to avoid false sound detection)

//AUTO REC / AUTO PLAY ROUTINES (soundTrigAuto)-----
~autoPlay_soundTrigAuto = {!post = 0! //<----NEW W/ PCONV
  Routine{
    if (~playSamp == true,
      { //if already playing sound file, we can assume it's completed by the time
this routine has stopped yielding based on commands when ~playSamp == false (see below code in
'else' of this conditional)...
        if (~playAudify == true, { //check flag to see if audification in
progress <----NEW
          if (post == 1, {">>>>AUDIFY IN
PROGRESS...<<<<".postln;});
          0.2.yieldAndReset;
        }, { //else if ~playAudify == false
          if (post == 1, {">>>>NO AUDIFICATION IN
PROGRESS...<<<<".postln;});
        }

        // if ~auto_partConvSamplesAutoLoop_StatusFlag == true
      }
    }
  }
  {~partConv_samples_autoLoop.(0...)};

```

```

~playSamp = false;
~time_lastPlay_stop = Main.elapsedTime.round(0.001);

//log sample playback stop time
if (~trigAudify == true, //<---NEW (avoids error caused by
freeing nonexistent ~audioBuf_play0 after ~audifyRout plays...
    {~trigAudify = false;},
    {if (~audioBuf_play0.bufnum != nil,
        {~audioBuf_play0.free;});} //only free buf if it exists <---NEW, similarly, this avoids error caused by
        freeing nonexistent buffer when other routines like autoPConv, etc. have operated instead of regular
        buffer processing effects....
    );

    {~button_playSamp.value = 0;}.defer; //update GUI button
    ~autoRecPlay_state = 0; // (cycle state to trigger ~autoRec
before next ~autoPlay)
    0.2.yieldAndReset;
});
    {
        {if not playing a soundFile or autoRecording...
            if (((~recSamp_input == false) && (~autoRecPlay_state == 1)),
//only play back sounds if not recording...
            {
                //PLAYBACK AUDIO FILE FROM DISK
                var interpSynth, fileDir, file, duration, temp, /*rate,

pitch, mix,*/ currentTime;

                //<---***CONDITION CHECKING TRIGGERING
                //---e.g. routines checking machine listening ugen

                ~playSamp = true; //set flag

                //vvvvvv ---- make this into a reusable function for

                //>-----FILE SELECTION CONDITIONS ---

                //-----vvvvvv-----NEW-----vvvvvv-----////////
                currentTime = Main.elapsedTime.round(0.001);
                if (currentTime <= (~time_lastRec_stop + 5),
                    {if just recorded new sound, play last

recorded sound
                    if (post == 1, {"---SOUND JUST
RECORDED - PLAYING LAST SOUND".postln;});
                    fileDir =

~fileAudio_writeln_lastFile;
                    },
                    {//else if it's been awhile since last

recorded sound
                    var playCase, randNum =
rrand(1.0, 10.0).round(0.01);
                    if (post == 1, {"---LAST SOUND
RECORDED > 5 SECONDS AGO....".postln;});

                    playCase = case
                    //{randNum == 0}{
                    {randNum <= 7}{ //70% of

time, play last recorded file
                    fileDir =

~fileAudio_writeln_lastFile;
                    if (post == 1, {"----
>PLAYING LAST REC FILE (70%)." + ~fileAudio_writeln_lastFile).postln;});

```

```

8.5} { //15% of time play recent rec file
>PLAYING RECENT REC FILE (20%)".post;});

```

```

second to last rec file

```

```

        fileDir = ~fileHistory_rec[1];
(post == 1, {" | second-to-last | ".post;});

```

```

~fileHistory_rec[2];

```

```

third-to-last | ".post;

```

```

third to last rec file

```

```

driving

```

```

~fileAudio_writeIn_lastFile;

```

```

{".....safe-driving...playing last rec file".postIn;});

```

```

+ ~returnFileNameFromPath.(fileDir)).postIn;});

```

```

<= 9} { //5% of time play older or random file....

```

```

>PLAYING OLDER / RAND FILE (5%):").postIn;});

```

```

(~fileAudio_writeIn_maxNum_exceeded == false,

```

```

fileNum;

```

```

(post == 1, {"-----maxNumFiles NOT EXCEEDED-----").postIn;});

```

```

(~fileAudio_writeIn_nameNum >= ~fileHistory_rec.size, //if rec mem is full

```

```

{

```

```

        if (0.5.coin == true,

```

```

                {fileDir = ~fileHistory_rec[~fileHistory_rec.size - 1]}, //select oldest file in rec
mem

```

```

                {fileDir = ~fileHistory_rec[~fileHistory_rec.size - 2]} //or select second-oldest
file in rec mem

```

```

        );

```

```

        if (post == 1, {"----->REC MEM FULL - Playing:" + ~returnFileNameFromPath.
(fileDir)).postIn;});

```

```

    },

```

```

    //else if rec mem is not full

```

```

}
//{randNum == 0} {
//{randNum <= 10} {
{randNum > 7 && randNum <=

```

```

        if (post == 1, {"----

```

```

        if (0.75.coin == true,
        { //play

```

```

                if

```

```

                },
                {fileDir =

```

```

                // " |

```

```

                } //or play

```

```

        );
        if (fileDir == nil, { //safe

```

```

                fileDir =

```

```

                if (post == 1,

```

```

                ));
        if (post == 1, {"FILE:"

```

```

        }
//{randNum == 0} {
{randNum > 8.5 && randNum

```

```

        if (post == 1, {"----

```

```

        if

```

```

        {

```

```

                var

```

```

                if

```

```

                if

```

```

        fileDir = ~fileHistory_rec[~fileAudio_writeIn_nameNum - 1]; //select older file...
        if (post == 1, {"---->REC MEM NOT FULL - Playing:" + ~returnFileNameFromPath.
(fileDir)).postln;});
    }
    );
    if
(fileDir == nil, {//safe driving
        fileDir = ~fileAudio_writeIn_lastFile;
        if (post == 1, {".....safe-driving...playing last rec file".postln;});
    });
    },
{ //else if max
    if
(post == 1, {"----maxNumFiles EXCEEDED----").postln;});
    if
(0.5.coin == true,
    {
        //play oldest or second-oldest file in rec mem
        if (0.5.coin == true,
            {
                fileDir = ~fileHistory_rec[~fileHistory_rec.size - 1]; //oldest file in rec
mem
                if (post == 1, {"----> Playing OLDEST file in recMem:" +
~returnFileNameFromPath.(fileDir)).postln;});
            },
            {
                fileDir = ~fileHistory_rec[~fileHistory_rec.size - 2]; //second-oldest
file in rMem
                if (post == 1, {"----> Playing SECOND OLDEST file in recMem:" +
~returnFileNameFromPath.(fileDir)).postln;});
            }
        );
    },
    {
        var randFile, selectFile;
        if (post == 1, {"---->RANDOM FILE SELECTION ---->".postln;});
        randFile = rrand(0, (~fileAudio_writeIn_maxNum - 1)).round(1);
        fileDir = (~fileAudio_writeIn_lastPath++randFile++".wav").standardizePath;
        if (post == 1, {"----> PLAYING:" + ~returnFileNameFromPath.(fileDir)).postln;});
        /*selectFile = (~fileAudio_writeIn_lastPath++randFile++".wav").standardizePath;

```



```

        ~audifyRout = ~audifyScene1.(~fileAudio_writeln_lastPath, ampBus:
~fileOutputLevelBus.asMap, outBus: ~outMainBus, fileNum: fileNum, cycle: cycle, rate: rate, pitch:
rate.neg, status: \playAudify);

        ~audifyRout = ~audifyScene1.(~fileAudio_writeln_lastPath, ampBus:
~fileOutputLevelBus.asMap, outBus: ~stream2sub, fileNum: fileNum, cycle: cycle, rate: rate, pitch:
rate.neg, status: \playAudify); //<---NEW outBus

        },
        { //else if we
have maximum # of samples saved...
rate
= rrand(28.0, 36.0).round(0.01);

        fileNum = ~fileAudio_writeln_maxNum;

        //~a
        udifyRout = ~audifyScene1.(~fileAudio_writeln_lastPath, ampBus: ~fileOutputLevelBus.asMap, outBus:
~outMainBus, fileNum: fileNum, cycle: cycle, rate: rate, pitch: rate.neg, status: \playAudify);

        ~audifyRout = ~audifyScene1.(~fileAudio_writeln_lastPath, ampBus:
~fileOutputLevelBus.asMap, outBus: ~stream2sub, fileNum: fileNum, cycle: cycle, rate: rate, pitch:
rate.neg, status: \playAudify); //<---NEW outBus

        if
        (post == 1, {"[AUDIFY] -- filesMax Exceeded: ".post; (" rate =" + rate).postIn;});
        }
        );
        if (post == 1,
{"[AUDIFY] -- STARTING... ".post;});
        SystemClock.play(~audifyRout);

        2.yieldAndReset;
        };

        //<-----ADD PCONV

ROUTINE HERE? or just below? 10 - 15 - 25%??

        }
        );
        //-----^-----NEW-----

        ^^^^^^-----////////

        //fileDir = ~fileAudio_writeln_lastFile; //<-----orig
        //OR ADD PCONV HERE AS PART OF LARGER

CONDITIONAL W/ BELOW..... <-----
        if ((0.25.coin == true) && (~fileHistory_rec[1] !=
nil), //25% chance, and if recorded >= 2 samples
        { //PERFORM PCONV
        var target, ir, temp, attTime =
0.5, susTime = 3, relTime = 2.3, dur, rate = 1, pitch = 1, pitchMix = 0, transp, transpCase;

        if (post == 1, {"=== PCONV

SELECTED ===".postIn;});

        target = fileDir; //assign target

and ir file paths

        if (target != ~fileHistory_rec[1],
        {ir =

        //if target ==

        if (0.5.coin

== true, {ir = ~fileHistory_rec[0];}, //50% chance use last rec as ir

        //else pick random file to use for ir

        if (~fileAudio_writeln_maxNum_exceeded == true,

```

```

        {ir = ~fileHistory_rec[rand(2, (~fileAudio_writeIn_maxNum - 1))];},
        {if we haven't written more than maxNum of permitted audio files, we gotta be
        careful to not select an invalid file path....
        ir = ~fileHistory_rec[rand(2, (~fileAudio_writeIn_maxNum - 1))];
        if (ir == nil, {ir = ~fileHistory_rec[0];}); //use last rec as ir, in case no file
        picked (lazy method for now....)
        }
    );

    );
}

    );
}

transp = rand(1.0,
transpCase = case
{transp <= 5} {50% no
    if (post == 1,
    }
    {transp > 5 && transp <= 7.5}
    if (post == 1,
    if (0.5.coin == true,
        {rate =
        {rate =
    );
    if (0.5.coin == true,
        if (0.33.coin
        }
    );
    if (0.7.coin
}

10.0).round(0.01);

transposition
{"[autoPconv] NO TRANSP".postln;});

{//25% slight transposition
{"[autoPconv] slight TRANSP".postln;});

//increase rate
rrand(2.0, 7.0).round(0.01);},
rrand(-2.0, -7.0).round(0.01);} //else decrease rate

{ //50% of time also engage pitch shifting
== true, //adjust pitch
    {pitch = (rate * (-1)).midiratio;}, //33% pitch exact inverse to rate
    {//66% else pitch transp inverse + offset to rate
    if (rate > 0,
        {pitch = rand(-12.0, -2.0).round(0.01).midiratio;},
        {pitch = rand(2.0, 12.0).round(0.01).midiratio;}
    );

    );
}

== true, //adjust pitch mix
    {pitchMix = 1;}, //70% chance, set pitchMix to 100%
    {pitchMix = rand(0.5, 0.75).round(0.01);} //30% chance, set pitchMix 50 - 75%
    );

```



```

    });
}
{transp > 7.5} { //25% extreme
    if (post == 1,
        if (0.5.coin ==
            {rate =
            {rate =
        );
        if (0.5.coin == true,
            if (0.33.coin
                }
            );
            if (0.7.coin
                );
            });
        };
        rate = rate.midiratio;
        susTime = rrand(3.5,
        dur = attTime + susTime +
        ~partConv_samples_autoLoop.
(1, post: post, targetFile: target, irFile: ir, targetBuf: \pcs_tBuf1, irBuf: \pcs_irBuf1, fftSize: 2048,
routine: \pcs_rout1, synth: \pcs_syn1, ampBus: ~fileOutputLevelBus.asMap, ampLimit: 0.3, outBus:
~stream2sub, attTime: attTime, susTime: susTime, relTime: relTime, playRate: rate, pitch: pitch,
pShiftMix: pitchMix, winSize: 0.1, pitchDisp: 0, timeDisp: 0.003);

Main.elapsedTime.round(0.001); //log sample play start time

~fileHistory_play.shift(1, fileDir);

1;}.defer; //update GUI

PLAYING BACK PCONV SAMPLES: target=" + ~returnFileNameFromPath.(target) + "I IR=" +
~returnFileNameFromPath.(ir)).postln;});

```

```

rate = " + rate + " | pitch = " + pitch + " | pitchMix = " + pitchMix + " | dur = " + dur).postln;});
                                                                    if (post == 1, {"[autoPconv]
                                                                    (dur + 0.1).yieldAndReset;
                                                                    },
                                                                    {
//PLAY SOUND FILE FROM DISK....
~audioBuf_play0 =
Buffer.read(s, fileDir, action: {arg buffer;
                                                                    var rand, procSelect;
                                                                    //duration =
buffer.numFrames/s.sampleRate;
                                                                    duration =
buffer.numFrames/buffer.sampleRate; //<--orig
                                                                    //EVENTUALLY
consider selecting proc behaviors based on features of analyzed audio <<<<
                                                                    rand = rrand(1.0,
10.0);
                                                                    procSelect = case
                                                                    //{rand == 0} {
                                                                    {rand <= 6.5} { //65%
chance //<---RESET TO THIS
                                                                    if (post == 1,
                                                                    //duration =
~sample_randAudify.value(buffer: buffer, duration: duration, ampBus: ~fileOutputLevelBus, outBus:
~outMainBus, post: post);
                                                                    duration =
~sample_randAudify.value(buffer: buffer, duration: duration, ampBus: ~fileOutputLevelBus.asMap,
outBus: ~stream2sub, post: post); //<---NEW outBus
                                                                    }
                                                                    //{rand <= 10} {
                                                                    {rand > 6.5} {
                                                                    //{rand > 6.5
&& rand <= 8.7} { //22% chance of modulated playback //<---RESET TO THIS
                                                                    //vvv-----
NEW ----vvvvv
                                                                    if (duration
<= 2, { //if short dur sound detected
                                                                    duration = ~trigRand_bufPlay_rateLFOsine.(buffer: buffer, duration: duration, ampBus:
~fileOutputLevelBus.asMap, outBus: ~stream2sub, shortDurThresh: 2, loopDurMin: 3.0, loopDurMax:
6.0, post: post);
                                                                    //du
ration = duration + 0.1; //some padding to see if missing buffer errors stop....
                                                                    }, { //else if
med - long dur sound > 2 seconds
                                                                    var
randSel, randCase;
                                                                    randSel = rrand(1.0, 10.0).round(0.01);
                                                                    randCase = case
                                                                    {randSel <= 5} { //50% chance
~trigRand_sineSineMod_bufPlay.(buffer: buffer, duration: duration, ampBus:
~fileOutputLevelBus.asMap, outBus: ~stream2sub, shortDurThresh: 2, post: post); //triggers synth
based on sound duration and random probs, also return mod duration when looping short sounds
                                                                    }
                                                                    {randSel > 5 && randSel <= 7} { //20% chance

```



```

{
    if (post == 1, {
        ("-----").postln;
        ("RECORDING.....").postln;
        ("currentTime:" + currentTime + "<
~time_minRec:" + ~time_minRec).postln;
    });
},
{
    featureBus.get({lval //get sum bus "lop" - 4
    if (( (val[0] >= ~loudnessThresh) || (val[1]
    == 1) || (val[3] >= ~pitchThresh) ),
        { //SOUND DETECTED
            ~hearSound = true;
            //CONTINUE
            if (post == 1, {
                ("-----").postln;
                ("***Recording... SOUND DETECTED").postln;
                ("Ldns-Onst-
PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3] ++ "]).postln;
            });
            //anti-troll / runaway
            if (currentTime >=
            {
                if
                (post == 1, { ("===MAX RECORD DUR EXCEEDED - STOP RECORDING===" ).postln;});
                ~recSamp_input_stopFrame = ~lastRecPhasePos;
                ~writeSampFromBufCopy_input.(); //sets GUI and ~recSamp_input = false
                ~time_lastRec_stop = currentTime; //<---
                ~hearSound = false;
                ~autoRecPlay_state = 1; //advance state to trigger ~autoPlay before next ~autoRec
            }
        });
    },
    { //IF NO SOUND DETECTED...
        if (~hearSound ==
        {
            //var
            currentTime = Main.elapsedTime.round(0.001); //<---already got this above
            if
            (currentTime >= ~time_listenTimeOut, //<---
            {
                //Sound Stop Confirmed - WRITE AUDIO SAMPLE (sound was stopped since last
                iteration)
                if (post == 1, {
                    ("-----").postln;
                    ("***Recording... SOUND STOP CONFIRMED").postln;

```

```

        ("Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3] ++
    "]).postln;

    });

    ~writeSampFromBufCopy_input.value(); //sets GUI and ~recSamp_input = false

    ~autoRecPlay_state = 1; //advance state to trigger ~autoPlay before next ~autoRec
    if (post == 1, {
        ("~autoRecPlay_state = 1 | ~autoPlay ONLY").postln;
        ("---.....---").postln;
    });

    //Analyze written audio file for features (avg loudness, pitchiness, MFCCs, etc)
    //Add sound file's extracted feature summary to a ~soundMemory buffer for future
    analysis and behavior triggering based on past history of sound events...
    },
    { //if currentTime < ~time_listenTimeOut....sound may not have stopped yet....
        if (post == 1, {
            ("----").postln;
            ("**Testing for sound stop...** - Current Time:" + currentTime).postln;
            ("---Terminate Test at:" + ~time_listenTimeOut).postln;
        });
    }

    );
    },
    { //else if
        ~hearSound == true (test to make sure sound has stopped but continue recording)
        me_listenTimeOut = ~listenDur_intraSound + Main.elapsedTime.round(0.001);
        ~time_listenTimeOut = ~listenDur_intraSound + currentTime; //<---
        if
        (post == 1, {
            ("-----").postln;
            ("**Recording...SOUND STOP? ... Testing until: " + ~time_listenTimeOut).postln;
            ("Ldns-Onst-PitchM = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3] ++ "]).postln;
        });
    }
    //G

    ET recHead phasor position and log "STOP SAMPLE TIME"
    ~phaseBusMonRec.get({|vall
    ~recSamp_input_stopFrame = (val - (~postRecSnip * s.sampleRate)).round(1);

```



```

    });
    if
    (~hearSound == false, //TEST SOUND START...
    {
        ~phaseBusMonRec.get({|vall //log recHead pos
        ~recSamp_input_startFrame = (val - (~preRecDur *
s.sampleRate)).round(1);
        if (~recSamp_input_startFrame < 0, { //if startFrame is negative...
            ~recSamp_input_startFrame = ~audioBuf0.numFrames +
~recSamp_input_startFrame; //wrap around buffer and log sound start frame
        });
        if (post == 1, {
            ("-----").postln;
            ("--> Sound START?: " ++
( ~recSamp_input_startFrame/s.sampleRate)).postln;
        });
        ~hearSound = true; //<--- moved inside get
    });
    },
    { //else ~hearSound == true
        var currentTime = Main.elapsedTime.round(0.001);
        ~time_minRec = currentTime + ~minRecDur; //minimum time to stop recording

        ~time_lastRec_start = currentTime - rate; //<---NEW - subtract rate b/c sound
detected in previous cycle of routine
        ~time_recTimeOut = ~time_lastRec_start + ~maxRecDur; //<---NEW - prevent
runaway recordings and trolls

        if (post == 1, {
            ("- - - -").postln;
            ("--> Sound START CONFIRMED:" +
( ~recSamp_input_startFrame/s.sampleRate)).postln;
            ("--REC Sample Start").postln;
        });
        ~recSamp_input = true; //causes us to LEAVE THIS SIDE OF ROUTINE
        {~button_recln.value = 1;}.defer;
    }
    );

```

```

    },
    { //if NO
        SOUND DETECTED .....
        if
        (post == 1, {
            ("--- NO SOUND ---").println;
            ("Ldns-Onst-pitMsr = [" ++ val[0] ++ ", " ++ val[1] ++ ", " ++ val[3] ++ "]").println;
        });
        (~hearSound == true,
        {
            ~hearSound = false;
            if (post == 1, {"--> ...false alarm?...").println;});
        },
        { //else if ~hearSound == false
            if (post == 1, {"-->...silence...").println;});

            if (~fileHistory_play[0] != nil, { //if recorded & played @ least 1 sound
                var currentTime = Main.elapsedTime.round(0.001);
                if (currentTime >= (~time_lastPlay_stop + ~listenDur_soundPatience),
                    {
                        ~autoRecPlay_state = 1;
                        //adjust ~listenDur_soundPatience based on timespan
                        since ~time_lastRec_stop; ? <-----CONSIDER DOING THIS -----<<<<<<<<<
                        ~listenDur_soundPatience = rrand(6.0,
                        20.0).round(0.01); //temporary
                        //~listenDur_soundPatience = 3; //for testing
                    }
                );
            });
        }
    );
}

playSamp == true || autoRecPlay_state != 0".println;

0.25.yieldAndReset;

};

{ //else if (~playSamp == true)
    { ("---Can't record when
    0.25.yieldAndReset;
    }
    };
}
);
}
);
}

```



```

        );
        rate.asFloat.yieldAndReset;
    });
};

/*
s.plotTree;

//RMSdb >= -10 to -6 == strong clipping and apparent feedback <-----
//RMSdb >= -15 to -12 == light clipping...

~listen_loudness.(active: 1,
    input: ~inputSendBus,
    outBus: \bus_inputdb,
    featSynth: \rms1,
    loudFlag: \inputSendLoud,
    loudSusFlag: \inputSendLoudSus,
    lowThresh: -35, uprThresh: -15, threshDur: 1,
    //lowThresh: -45, uprThresh: -25, threshDur: 1,
    lastThreshTime: \inputSendThreshTime,
    monRoutine: \inputSendMon,
    rate: 0.5,
    post: 1
);

~smoothBusOutputKR.(active: 1, inBus: ~bus_inputdb, outBus: \smoothBus_inputdb, synth:
\smooth_inputdb, synthMoveAfter: ~rms1, avgDur: 6);

~smoothBus_inputdb.getSynchronous;

~updateFeatureMemory.(active: 1, featureBus: ~smoothBus_inputdb, array: \mem_inputdb, arraySize:
10, routine: \updateMem_inputdb, rate: 6, post: 1, postLabel: "==> MEM_inputdb:");

~updateFeatureMemory.(active: 0, routine: \updateMem_inputdb);
~smoothBusOutputKR.(active: 0, outBus: \smoothBus_inputdb, synth: \smooth_inputdb);
(
~listen_loudness.(active: 0,
    outBus: \bus_inputdb,
    featSynth: \rms1,
    loudFlag: \inputSendLoud,
    loudSusFlag: \inputSendLoudSus,
    monRoutine: \inputSendMon
);
)
~inputSendLoudSus
~inputSendLoud

~listen_loudness_post.(active: 1, input: ~inputSendBus, outBus: \busSum_loud, sumSynth:
\sumSyn_loud, post: 1);
~listen_loudness_post.(active: 0, outBus: \busSum_loud, sumSynth: \sumSyn_loud);
*/

~listen_loudness_post = { lactive = 1, input, outBus, sumSynth, rate = 0.1, post = 1 | //for testing
    if (active == 1, { //start feature extraction...
        if (~bufRecorder.isRunning, {
            //LOUDNESS monitor
            l = Synth(\Loudness, [input, input, \outBus,
~bus_loudness]).moveAfter(~bufRecorder);
            //RMS
            r = Synth(\RMSdb, [input, input, \outBus, ~bus_rms]).moveAfter(l);

            outBus.envirPut(~sumFunc_xin.([~bus_loudness, ~bus_rms],
sumSynth)); //assign bus and resulting summing synth to new env vars

            if (post == 1, { //only run this routine to monitor features w/ post window...

```

```

SCML could live if needed
rate, post: post);

//====THIS IS WHERE ~transmit feature via OSC routine from
~listenPost = ~streamFeatures.value(bus: outBus.envirGet, rate:
SystemClock.play(~listenPost);
});
"====LISTENING NOW====".postln;
}, {
"====CAN'T LISTEN UNLESS BUF RECORDER ENABLED====".postln;
});
}, { //if active == 0, stop feature extraction and clean up...
~listenPost.stop; ~listenPost.free; //this cleanup only needed if post==1
sumSynth.envirGet.free;
outBus.envirGet.free;
l.free;
r.free;
"====STOPPED LISTENING====".postln;
});
};

~listen_loudness = {lactive = 1, input, outBus, featSynth, loudFlag, loudSusFlag, lowThresh = -12,
uprThresh = -6, threshDur = 0.4, lastThreshTime, rate = 0.1, post = 0, monRoutine!
if (active == 1, { //start detecting feedback
if (~bufRecorder.isRunning, {
var thisOutBus;

//RMS monitor
outBus.envirPut(Bus.control(s, 1));
thisOutBus = outBus.envirGet;

//lastThreshTime.envirPut(nil); //<---not necessary

featSynth.envirPut(Synth(\RMSdb, [input, input, \outBus,
thisOutBus])).moveAfter(~bufRecorder));
NodeWatcher.register(featSynth.envirGet);
//RMSdb >= -10 to -6 == strong clipping and apparent feedback <----
//RMSdb >= -15 to -12 == light clipping...

//if (post == 1, { //only run this routine to monitor features w/ post window...
CAUTION: can't .get from single bus using two different routines....will eventually cause one routine's
get requests to fail, that's why this is commented out for now, since below monRoutine also has debug
post feature...
~listenPost = ~streamFeatures.value(bus: thisOutBus, rate: rate,
post: post);
SystemClock.play(~listenPost);
}); */

loudFlag.envirPut(0); //loudFlag indicates instantaneous loudness level: 0 =
no loudness, 1 = lower loudness threshold exceeded, 2 = upper loudness threshold exceeded
loudSusFlag.envirPut(false); //loudSusFlag indicates presence of ongoing
loud sound / feedback (e.g. if loudFlag = 1, but loudSusFlag = false, no on-going loudness has been
confirmed by monitor_loudness routine)

//START LOUDNESS MONITOR ROUTINE
monRoutine.envirPut(~monitor_loudness.(input: thisOutBus, loudFlag:
loudFlag, loudSusFlag: loudSusFlag, lowThresh: lowThresh, uprThresh: uprThresh, threshDur:
threshDur, lastThreshTime: lastThreshTime, rate: rate, post: post));
SystemClock.play(monRoutine.envirGet);

"====LOUDNESS/FEEDBACK DETECTION ACTIVE====".postln;
}, {
"====CAN'T DETECT LOUDNESS/FEEDBACK UNLESS BUF RECORDER
ACTIVE====".postln;
});
}, { //if active == 0, stop feature extraction and clean up...

```

```

//if (~listenPost != nil, {~listenPost.stop; ~listenPost.free;}); //this cleanup only needed
if post==1

//STOP LOUDNESS/FEEDBACK MONITOR ROUTINE
monRoutine.envirGet.stop;
monRoutine.envirGet.free; //<---this doesn't free routine....set to nil?
outBus.envirGet.free; //free RMS feature extractor output bus
featSynth.envirGet.free; //free RMS feature extractor synth
loudFlag.envirPut(0);
loudSusFlag.envirPut(false);

"====LOUDNESS/FEEDBACK DETECTION DISABLED====".postln;
});
};

~monitor_loudness = {input, loudFlag, loudSusFlag, lowThresh = -12, uprThresh = -6, threshDur = 0.4,
lastThreshTime, rate = 0.1, post = 0} //<---REVISE THIS: change threshDur and lastThreshTime to
onsetDur and lastOnsetTime I add releaseDur or loudTrigHysteresis I maybe add boolean args to
trigger autolevel / anti-feedback routines as a quick and easy method, but would rather avoid in long
term and have autolevel / anti-feedback routines stand-alone and continuously check
~monitor_loudness flags continuously...
Routine({
var x, cond = Condition.new; //<---NEW
if (post == 1, {"[MON LOUDNESS]-----".postln;});
input.get({|vall
x = case
{(val >= lowThresh) && (val < uprThresh)} {//if current db is between lower
and upper loudness thresh
if (post == 1, {"-->LOUD SIG > (low thresh): input db =" +
val.round(0.01)).postln;});

if (loudFlag.envirGet == 0,
{
loudFlag.envirPut(1);
if (loudSusFlag.envirGet == false,
{
var currentTime =
Main.elapsedTime.round(0.001);

lastThreshTime.envirPut(currentTime); //log time of threshold transition from loudFlag = 0 to 1
if (post == 1, {"---> TRANS
from loudFlag = 0: loudFlag = 1 | LOUDNESS STARTING??? | lastThreshTime =" +
lastThreshTime.envirGet).postln;});
},
{//else if loudSusFlag already == true,
must have been a false/transient quiet during loudness
if (post == 1, {"---> TRANS
from loudFlag = 0: loudFlag = 1 | LOUDNESS ONGOING").postln;});

//<---DO ANYTHING
MORE??? ADJUST ANTI-FEEDBACK ROUTINES????<-----
}
});
{//else if loudFlag already == 1 or 2
if (loudFlag.envirGet == 1,
{
if (loudSusFlag.envirGet ==
false,
{
var
currentTime = Main.elapsedTime.round(0.001);
if
((currentTime - lastThreshTime.envirGet) >= threshDur,

```

```

{

    loudSusFlag.envirPut(true);

    if (post == 1, {"-->LOUD SOUND SUS CONFIRMED! (low thresh): input db =" +
val.round(0.01)).println;});

    //<----TRIGGER ANTI-FEEDBACK ROUTINES - ADJUST LEVELS ??? <-----

    //trigger long-term loudness monitoring timer and corresponding autoActions here instead OR
w/in ~updateFeatureMemory routine?

    },
    {

        if (post == 1, {"-->LOUD SOUND SUS TEST..... (low thresh): input db =" +
val.round(0.01)).println;});

    }

    );

    },
    { //else if loudSusFlag

        if (post == 1,
{"-->LOUD SOUND SUS continues... (low thresh): input db =" + val.round(0.01)).println;});
        //<--
        --INTENSIFY ANTI-FEEDBACK / ADJUST LEVELS MORE ??? <-----
    }

    );

    },
    { //else if loudFlag == 2
        loudFlag.envirPut(1);

        if (loudSusFlag.envirGet ==

false,

        {

            var

            if

            {

                loudSusFlag.envirPut(true);

                if (post == 1, {"-->TRANS from loudFlag = 2: loudFlag = 1 | LOUD SOUND SUS
CONFIRMED! (low thresh): input db =" + val.round(0.01)).println;});

                //<----TRIGGER ANTI-FEEDBACK ROUTINES - ADJUST LEVELS ??? <-----

                },
                {

                    if (post == 1, {"--> TRANS from loudFlag = 2: loudFlag = 1 | LOUD SOUND SUS TEST...(low
thresh): input db =" + val.round(0.01) + "I lastThreshTime =" + lastThreshTime.envirGet).println;});
                    }

                );

            },
            { //else if loudSusFlag

                already == true

                if (post == 1,
{"-->LOUD SOUND SUS diminished from loudFlag = 2, but continues... (low thresh): input db =" +
val.round(0.01)).println;});
                //<----
                ADJUST LEVELS / ANTI-FEEDBACK ROUTINES ??? <-----

            }

        )

    }

}

```

```

);
    }
    );
}
{val >= uprThresh} { //else if current db >= upper loudness threshold
-----<<<<<
    if (loudFlag.envirGet < 2,
        {
            if (post == 1, {"-->VERY LOUD SIG: input db ="
+ val.round(0.01)).postln;});

            if (loudFlag.envirGet == 1,
                {
                    loudFlag.envirPut(2);

                    if (loudSusFlag.envirGet ==
false,
                        {"//we must have
transitioned from loudFlag = 1 to loudFlag = 2 between two consec iterations of routine without
triggering loudSusFlag...therefore, don't reset lastThreshTime, since it was previously set when
loudFlag set to = 1
                        var
currentTime = Main.elapsedTime.round(0.001);
                        if (post == 1,
{"---DIRECT TRANS FROM: loudFlag = 1 to loudFlag = 2 B4 loudSusFlag set---").postln;});
                        if
((currentTime - lastThreshTime.envirGet) >= threshDur, //<--- or >= rate*2 or rate*3
                            {
                                loudSusFlag.envirPut(true);

                                if (post == 1, {"-->LOUD SOUND SUS CONFIRMED! (UPR thresh): input db =" +
val.round(0.01)).postln;});

                                //<-----TRIGGER ANTI-FEEDBACK ROUTINES AND ADJUST LEVELS <-----?
                                    },
                                    {

                                        if (post == 1, {"-->LOUD SOUND SUS TEST.... (UPR thresh): input db =" +
val.round(0.01)).postln;});

                                        //<-----DO NOTHING.. WAIT AND SEE..?... NO ADJUST LEVELS IMMEDIATELY <---
                                            }
                                            );
                                            },
                                            {"//else if loudSusFlag
== true
                                                if (post == 1,
{"--->TRANS FROM: loudFlag = 1 to loudFlag = 2 --- loudSusFlag already = true").postln;});

                                                //<-----
TRIGGER MORE EXTREME ANTI-FEEDBACK ROUTINES / SET LEVEL TO -inf ???
                                                    }
                                                    );
                                                    },
                                                    {"//else if transitioning from loudFlag == 0
loudFlag.envirPut(2);

                                                    if (loudSusFlag.envirGet ==
false,
                                                        {
                                                            var
currentTime = Main.elapsedTime.round(0.001);

                                                            lastThreshTime.envirPut(currentTime);

```

```

if (post == 1,
{"--> TRANS from 0: loudFlag = 2 | lastThreshTime =" + lastThreshTime.envirGet).postln;});
    },
    //else if loudSusFlag
    if (post == 1,
{"FALSE / SHORT QUIET - LOUDNESS CONTINUING...").postln;});

//<----
    TRIGGER MORE EXTREME ANTI-FEEDBACK ROUTINES / SET LEVEL TO -inf ???
    }
    );
    };
    //if loudFlag == 2 (a.k.a. if rmsBusVal already >=
    uprThresh)
    if (loudSusFlag.envirGet == false,
    {
        var currentTime =
        Main.elapsedTime.round(0.001);
        lastThreshTime.envirGet) >= (rate * 2), { //if the routine's cycled twice
            loudSusFlag.envirPut(true);
            if (post == 1, {"--
>VERY LOUD SOUND SUS CONFRMD (upr thresh): input db =" + val.round(0.01)).postln;});
            //<-----TRIGGER
            ANTI-FEEDBACK ROUTINES AND ADJUST LEVELS <----?
            }, {
            if (post == 1, {"--
>VERY LOUD SOUND SUS TEST..... (upr thresh): input db =" + val.round(0.01)).postln;});
            });
            //else if loudSusFlag == true
            if (post == 1, {"-->VERY LOUD
SOUND SUS continues... (upr thresh): input db =" + val.round(0.01)).postln;});
            //<----TRIGGER MORE
            EXTREME ANTI-FEEDBACK ROUTINES / SET LEVEL TO -inf ???
            }
            );
            };
            {
            {val < lowThresh} { //else if current db < lower loudness threshold -----
            if (post == 1, {"-->QUIET SIG < lowThresh: input db =" +
val.round(0.01)).postln;});
            if (loudFlag.envirGet > 0,
            {
                loudFlag.envirPut(0);
                if (loudSusFlag.envirGet == false,
                {
                    if (post == 1,
{"FALSE/TRANSIENT LOUD TRIG - loudSusFlag = false").postln;});
                    },
                    //else if loudSusFlag == true, we're
                    var currentTime =
                    Main.elapsedTime.round(0.001);
                    lastThreshTime.envirPut(currentTime); //log time of transition from loudFlag > 0 to 0

```

```

FROM loudFlag > 0 to loudFlag = 0 | lastThreshTime = " + lastThreshTime.envirGet + " | loudSusFlag =
true | LOUDNESS PREV OCCURRING...").postln;});
    }
    );
    },
    { //else if loudFlag already == 0
      if (loudSusFlag.envirGet == false,
        {
          //do nothing...no loudness or
          if (post == 1, {"--->NO
feedback has been occurring.....
LOUDNESS...cont...").postln;});
        },
        { //else if loudSusFlag == true, test to see
          var currentTime =
            if ((currentTime -
lastThreshTime.envirGet) >= threshDur, //or set to >= (rate*3) ??? <---
              {
                loudSusFlag.envirPut(false);
                {"--->NO LOUDNESS CONFIRMED | loudSusFlag = false...").postln;});
              if (post == 1,
                //TERMINAT
                //RESET
                },
                { //else if (currentTime
                  if (post == 1,
                    {"--->TESTING FOR END OF LOUDNESS...").postln;});
                  }
                );
              }
            );
          }
        );
      cond.unhang; //<----THIS OR .getSync
    });
    cond.hang;
    rate.yieldAndReset;
  });
};

~listen_sound = { lactive = 1, input, outBus, sumSynth, rate = 0.1, post = 0 | //setup/shutdown feat
extractors on a bus
  //lactive = 1, input, rate = 0.1, post = 0 | //orig args...
  if (active == 1, { //start feature extraction...
    if (~bufRecorder.isRunning, {
      //LOUDNESS monitor input = ~inputSendBus
      l = Synth(\Loudness, [input, input, \outBus,
~bus_loudness]).moveAfter(~bufRecorder);
      //PITCH tracker
      p = Synth(\Pitch, [input, input, \outBus, ~bus_pitch, \ampThresh, 0.02, \clar,
1]).moveAfter(l);
      //ONSET detector
      n = Synth(\onsets, [input, input, \outBus, ~bus_onsets, \threshold, 0.5,
\trigtime, 0.25]).moveAfter(l);
      //etc....
    }
  }
};

```

```

//fire off ~senseNoiseFloor routine....?
outBus.envirPut(~sumFunc_xin.([~bus_loudness, ~bus_onsets, ~bus_pitch],
sumSynth)); //assign bus and resulting summing synth to new env vars

if (post == 1, { //only run this routine to monitor features w/ post window...
//===THIS IS WHERE ~transmit feature via OSC routine from
SCML could live if needed
~listenPost = ~streamFeatures.value(bus: outBus.envirGet, rate:
rate, post: post);
SystemClock.play(~listenPost);
});

~hearSound = false;
~listening = true;
"====LISTENING NOW====".postln;
}, {
"====CAN'T LISTEN UNLESS BUF RECORDER ENABLED====".postln;
});
}, { //if active == 0, stop feature extraction and clean up...
if (~listening == true, { //only free synths and cleanup if we were listening in the first
place, otherwise don't do anything and avoid server errors....
if (~listenPost != nil, {~listenPost.stop; ~listenPost.free;}); //this cleanup only
needed if post==1

sumSynth.envirGet.free;
outBus.envirGet.free;
l.free;
p.free;
n.free;
~listening = false;
~hearSound = false;

~bus_onsets.set(0); //reset value in case mouse click triggers onset detector
at termination of routine
//zero out loudness and other feature busses here too???

"====STOPPED LISTENING====".postln;
});
});

~button_autoRecPlay = Button(~view_perfButtons, Rect(113, 25, 25, 25))
//~view_perfButtons, Rect(94, 25, 25, 25)
.states_( [
["^v", Color.black, Color(0.6, 0.9, 0.4, 0.3)] ,
["^v", Color.white, Color(0.8, 1.0, 0.4)]
] )
.font_(Font("Monaco", 18) )
.action_( {
arg obj;
if (
obj.value == 1,
{
~mode_autoRecPlay = true; //<---NEW mode flag
//~listen_sound.value(active: 1, input: ~inputSendBus, post: 0); //start
listening
~listen_sound.value(active: 1, input: ~inputSendBus, outBus: \busSum_lop,
sumSynth: \sumSyn_1, post: 0); //start listening
~autoRecStartTrig = true; //enables debounce when ~autoRecStart
initialized

~autoRecStart = ~autoRec_soundTrigAuto.(featureBus: ~busSum_lop, rate:
0.1, debounceDur: 0.5, post: 0);
SystemClock.play(~autoRecStart);
}
}
}

```



```

//~autoPlay_soundTrigAuto.reset;
//SystemClock.play(~autoPlay_soundTrigAuto);
~autoPlayRout = ~autoPlay_soundTrigAuto.(post: 1);
SystemClock.play(~autoPlayRout);
},
{
~mode_autoRecPlay = false; //<---NEW mode flag
~autoRecStart.stop;
~autoRecStart.free; //doesn't do anything...

//~autoPlay_soundTrigAuto.stop;
~autoPlayRout.stop;
~autoPlayRout.free; //doesn't do anything...

if (~recSamp_input == true, { //pertains to ~autoRec_soundTrig1
    if (~hearSound == true, { //if actively "recording" sound, stop
recording and write sample file
        ~phaseBusMonRec.get({lval
            ~recSamp_input_stopFrame = val.round(1);
            ("--Sound Stop at " ++
(~recSamp_input_stopFrame / s.sampleRate)).postln;
            ~writeSampFromBufCopy_input.value(); //writes
sample, sets ~recSamp = false, updates GUI
            ~autoRecPlay_state = 0; //<----- NEW
        });
    });
}, { //if ~recSamp_input == false
    ~button_recln.value = 0; //<---not needed?
    ~autoRecPlay_state = 0; //<----- NEW
});

~listen_sound.(0, outBus: \busSum_lop, sumSynth: \sumSyn_1); //stop
listening

if (~playSamp == true, {
    ~samp0.free;
    ~audioBuf_play0.close; //<---not needed...?
    ~audioBuf_play0.free;
    ~playSamp = false;
    ~button_playSamp.value = 0;
    ~autoRecPlay_state = 0; //<----- NEW

    if (~pcs_syn1 != nil, { //end pConv routine if active (checking for its
synth)....<-----NEW
        if (~pcs_syn1.isRunning, {
            ~partConv_samples_autoLoop.(0, post: 1,
targetBuf: \pcs_tBuf1, irBuf: \pcs_irBuf1, routine: \pcs_rout1, synth: \pcs_syn1);
        });
    });
});

}

);
})
.value_(0);

~button_recln = Button(~view_sampButtons, Rect(0, 0, 45, 20))
.states_( [
    ["RECIN", Color.black, Color(0.0, 0.5, 0, 0.3)] ,
    ["RECIN", Color.white, Color(0.0, 0.5, 0)]
])
.font_(Font("Monaco", 11) )
.action_( {
    arg obj;

```

```

        if (
            obj.value == 1,
            { //BEGIN RECORDING / WRITING AUDIO INPUT TO FILE
                ~audioBuf_inMainWrite = Buffer.alloc(s, 65536, 1); //allocate a disk i/o buffer
                (min rec frame size is 65536)
                ~audioBuf_inMainWrite.write((~fileAudio_writeIn_lastPath+
                +~fileAudio_writeIn_nameNum+++".wav").standardizePath, "wav", ~sampBitDepth, 0, 0, true); // create
                an output file for this buffer, leaving it "open" (not recording yet, just created file w/ format header)
                ~writeIn2Disk = Synth.tail(nil, \write2Disk, [\buf, ~audioBuf_inMainWrite,
                \inBus, ~inputSendBus] ); //record audio from rec playhead (bufRead synth sends audio input to
                ~inputSendBus), and write an audio file to hard disk
                ~recSamp_input = true; //<--- not sure if this flag's needed right now...
            },
            { //STOP RECORDING / WRITING AUDIO INPUT TO FILE
                ~writeIn2Disk.free; //stop recording to disk
                ~audioBuf_inMainWrite.close; //close the buffer to finish writing the sound
                file (won't be readable otherwise)
                ~audioBuf_inMainWrite.free;
                ~fileAudio_writeIn_nameNum = ~fileAudio_writeIn_nameNum + 1;
                //increment file name suffix in prep for next rec file
                if (~fileAudio_writeIn_nameNum > ~fileAudio_writeIn_maxNum, { //cap max
                file count...
                    ~fileAudio_writeIn_nameNum = 0;
                });
                ~recSamp_input = false; //<--- not sure if this flag's needed right now...
            });
    });

    ~button_recOut = Button(~view_sampButtons, Rect(0, 22, 45, 20))
    .states_( [
        ["RECOUT", Color.black, Color(0.95, 0, 0, 0.3)],
        ["RECOUT", Color.white, Color(0.95, 0, 0)]
    ])
    .font_(Font("Monaco", 11))
    .action_( {
        arg obj;
        if (
            obj.value == 1,
            { //BEGIN RECORDING / WRITING MAIN AUDIO OUTPUT TO FILE
                ~audioBuf_outMainWrite = Buffer.alloc(s, 65536, 1); //allocate a disk i/o
                buffer (min rec frame size is 65536)
                ~audioBuf_outMainWrite.write((~fileAudio_writeOut_lastPath+
                +~fileAudio_writeOut_nameNum+++".wav").standardizePath, "wav", ~sampBitDepth, 0, 0, true); // create
                an output file for this buffer, leaving it "open" (not recording yet, just created file w/ format header)
                ~write2Disk = Synth.tail(nil, \write2Disk, [\buf, ~audioBuf_outMainWrite,
                \inBus, 0] ); //record audio sent to bus 0 to buffer writing to audio file
                ~recSamp_output = true; //<--- not sure if this flag's needed right now...
            },
            { //STOP RECORDING / WRITING MAIN AUDIO OUTPUT TO FILE
                ~write2Disk.free; //stop recording to disk
                ~audioBuf_outMainWrite.close; //close the buffer to finish writing the sound
                file (won't be readable otherwise)
                ~audioBuf_outMainWrite.free; //free buffer
                ~fileAudio_writeOut_nameNum = ~fileAudio_writeOut_nameNum + 1;
                //increment file name suffix in prep for next rec file
                if (~fileAudio_writeOut_nameNum > ~fileAudio_writeOut_maxNum, { //cap
                max file count...
                    ~fileAudio_writeOut_nameNum = 0;
                });
                ~recSamp_output = false; //<--- not sure if this flag's needed right now...
            });
    });

    ~button_reclnOut = Button(~view_sampButtons, Rect(45, 0, 66, 20))
    .states_( [

```

```

["RECinOut", Color.black, Color(0.9, 0.9, 0.3, 0.3)] ,
["RECinOut", Color.black, Color(0.9, 0.9, 0.3)]
])
.font_(Font("Monaco", 12) )
.action_( {
    arg obj;
    if (
        obj.value == 1,
        {
            ~button_recIn.valueAction = 1; //<---NOT GOOD IN LONG RUN...CREATE
            FUNCTION TO TRIGGER AUDIO FILE WRITING SYNTHS...
            ~button_recOut.valueAction = 1;
        },
        {
            ~button_recIn.valueAction = 0;
            ~button_recOut.valueAction = 0;
        }
    );
});
.value_(0);

~button_playSamp = Button(~view_sampButtons, Rect(45, 22, 66, 20) )
.states_( [
    ["PLAY FILE", Color.black, Color(0.0, 0.5, 0.5, 0.3)] ,
    ["PLAY FILE", Color.white, Color(0.0, 0.5, 0.5)]
])
.font_(Font("Monaco", 11) )
.action_( {
    arg obj;
    if (
        obj.value == 1,
        { //BEGIN PLAYING AUDIO FILE
            //play random or pre-selected or wildcard, etc. file(s) from sample folder?
            if (~fileAudio_writeln_nameNum >= ~minNumFilesB4Playback,
                {
                    var randFilePick, fileDir;
                    ~playSamp = true; //yes or no?
                    randFilePick = rand(0, (~fileAudio_writeln_nameNum -
1));
                    fileDir = (~fileAudio_writeln_lastPath++randFilePick+
+ ".wav").standardizePath;
                    fileDir.postln;
                    ~audioBuf_play0 = Buffer.read(s, fileDir, action: {arg
buffer;
                    //~samp0 = Synth(\bufRead_noLoop, [buf,
buffer, \amp, ~fileOutputLevelBus.asMap, \out, ~outMainBus]);
                    ~samp0 = Synth(\bufRead_noLoop, [buf,
buffer, \amp, ~fileOutputLevelBus.asMap, \out, ~stream2sub]); //<---NEW outBus
                    NodeWatcher.register(~samp0);

                    //---put routine here that will turn off button when
sample playback complete? ----
                });
            }
        });
    }
    //STOP Sample playback
    ~playSamp = false; //yes or no?
    if (~samp0.isRunning, {~samp0.free;});
    ~audioBuf_play0.free;
});
});

```

```

/*~label_playSamp = StaticText(~view_sampButtons, Rect(113, 23, 135, 20))
.string_("/PATH/./FILENAME")
.font_(Font("Monaco", 12) )
.align_(\left); */

//+++++ADD FILE SELECTOR INTERFACE (OPEN DIALOG?) FOR SAMPLE LOADING...

//+++++ADD BUFFSCOPE FOR PLAYING / LOADED AUDIO FILE(s) (make into re-usable view for GUI
of multi-channel & multi-sample playback)

//-----

///----- LEVELS -----///

~slider_inFlow1_level = Slider(~view_altLevels, Rect(9, 2, 12, 73))
.background_(Color(0.0, 0.5, 0.9))
.value_(1.0)
.value_(~spec_slider_dB.unmap(0)) //<--
.action_({|obj|
    var mapVal, val = obj.value.round(0.001);
    mapVal = ~spec_slider_dB.map(val).round(0.01); //convert linear amp to db scaling
    ~inputStreamLevelBus.set(mapVal.dbamp.round(0.001));
    ~numBox_inFlow1_level.value_(mapVal);
});

~numBox_inFlow1_level = NumberBox(~view_altLevels, Rect(2, 75, 30, 15))
.value_(0.0)
.clipLo_(\inf)
.clipHi_(0.0)
.decimals_(1)
.scroll_step_(0.1)
.font_(Font("Monaco", 8) )
.align_(\center)
.action_({|obj|
    var mapVal, val = obj.value.round(0.1);
    mapVal = ~spec_slider_db.unmap(val); //convert db num into linear amp scaling
    ~inputStreamLevelBus.set(mapVal.round(0.001));
    ~slider_inFlow1_level.value_(mapVal.round(0.01));
});

/*~slider_inFlow1_level = Slider(~view_altLevels, Rect(9, 2, 12, 73))
.background_(Color(0.0, 0.5, 0.9))
.value_(1.0)
.action_({
    arg obj;
    var val;
    val = obj.value.round(0.01);
    ~inputStreamLevelBus.set(val);
    ~numBox_inFlow1_level.value_(val);
});

~numBox_inFlow1_level = NumberBox(~view_altLevels, Rect(2, 75, 26, 15))
.value_(1.0)
.clipLo_(0.0)
.clipHi_(1.0)
.decimals_(2)
.font_(Font("Monaco", 8) )
.scroll_step_(0.01)
.action_( {
    arg obj;
    var val;
    val = obj.value.round(0.01);
    ~inputStreamLevelBus.set(val);

```

```

~slider_inFlow1_level.value_(val);
}); */

~label_inFlow1_level = StaticText(~view_altLevels, Rect(0, 94, 30, 8))
.string_("S1-IN")
.font_(Font("Monaco", 10) )
.align_(\center);

~slider_sampOut_level = Slider(~view_altLevels, Rect(44, 2, 12, 73))
.background_(Color(0.0, 0.8, 0.2))
.value_(1.0)
~spec_slider_dB.unmap(0) //<--
.action_({|obj|
    var mapVal, val = obj.value.round(0.001);
    mapVal = ~spec_slider_dB.map(val).round(0.01); //convert linear amp to db scaling
    ~fileOutputLevelBus.set(mapVal.dbamp.round(0.001)); //set level control bus value
    ~numBox_sampOut_level.value_(mapVal);
});

~numBox_sampOut_level = NumberBox(~view_altLevels, Rect(37, 75, 30, 15))
.value_(0.0)
.clipLo_(-inf)
.clipHi_(0.0)
.decimals_(1)
.scroll_step_(0.1)
.font_(Font("Monaco", 8) )
.align_(\center)
.action_({|obj|
    var mapVal, val = obj.value.round(0.1);
    mapVal = ~spec_slider_db.unmap(val); //convert db num into linear amp scaling
    ~fileOutputLevelBus.set(mapVal.round(0.001));
    ~slider_sampOut_level.value_(mapVal.round(0.01));
});

/*~slider_sampOut_level = Slider(~view_altLevels, Rect(44, 2, 12, 73))
.background_(Color(0.0, 0.8, 0.2))
.value_(1.0)
.action_( {
    arg obj;
    var val;
    val = obj.value.round(0.01);
    ~fileOutputLevelBus.set(val);
    ~numBox_sampOut_level.value_(val);
});

~numBox_sampOut_level = NumberBox(~view_altLevels, Rect(37, 75, 26, 15))
.value_(1.0)
.clipLo_(0.0)
.clipHi_(1.0)
.decimals_(2)
.font_(Font("Monaco", 8) )
.scroll_step_(0.01)
.action_( {
    arg obj;
    var val;
    val = obj.value.round(0.01);
    ~fileOutputLevelBus.set(val);
    ~slider_sampOut_level.value_(val);
}); */

~label_sampOut_level = StaticText(~view_altLevels, Rect(35, 94, 30, 8) )
.string_("S2-SP")
.font_(Font("Monaco", 10) )
.align_(\center);

```

```

//-----
/*
//----- SCOPE VIEW (AUDIO BUF0) -----

//~scopeView = ScopeView(w.view, Rect(15, 10, 665, 170) );
~scopeView = ScopeView(w, Rect(15, 10, 665, 170) );
~scopeView.server_(s);
~scopeView.xZoom = 1; //changing these values doesn't seem to effect display
//~scopeView.x = 0; //changing these values doesn't seem to effect display
~scopeView.yZoom = 1.0;
*/

//----- PLOTTER -----//
if (~plotterEnabled == true, {
    GUI.skin.plot.fontColor = Color.red;
    GUI.skin.plot.plotColor = Color.blue;
    GUI.skin.plot.background = Color.black;
    GUI.skin.plot.gridColorX = Color.white;
    GUI.skin.plot.gridColorY = Color.cyan;
    //GUI.skin.plot.gridOnY = false;
    //GUI.skin.plot.gridOnX = false;

    ~bufPlot = Plotter.new("Buffer", parent: ~view_bufPlot); //CREATE PLOTTER

    ~audioBuf0.loadToFloatArray(action: {arg array;
        {~bufPlot.value = array}.defer; //create bufPlot matrix w/ as many values as buffer
        (very big...should probably scale this to be smaller more light weight plotter, i.e. divide buffer frames by
        10 or 100 to represent average of every 10 - 100ms slices of buffer...
        ));

    {
        ~bufPlot.setProperties(
            \gridOnY, false,
            \gridOnX, true
        );
        ~bufPlot.plotMode = \linear;
        ~bufPlot.domainSpecs = ControlSpec(0, ~audioBuf0_dur);
        //~bufPlot.resolution = 1; //<---too coarse...
        ~bufPlot.resolution = 0.04; //good compromise for GUI speed, smaller resos (.02, .01)
        causes GUI to lag...
        ~bufPlot.updatePlots.refresh;
    }.defer(0.2); //<---if this is any shorter, these properties won't be set for some reason....

    ~updateBufPlot_prevUpdateFrame = nil;

    ~updateBufPlot = Routine({ //basic incremental buf plotter
        if (~updateBufPlot_prevUpdateFrame == nil,
            { //if first time routine's run since program launch
                var plotDur, currentFrame;

                0.5.yield; //wait for buffer to fill w/ values...

                //then evaluate bufPlot function from frame 0 to currentFrame
                currentFrame = ~lastRecPhasePos;

                //plotDur = ~lastRecPhasePos + 1; //inclusive of frame 0
                plotDur = currentFrame + 1; //inclusive of frame 0
                ~audioBuf0.loadToFloatArray(index: 0, count: plotDur, action: {l
arrayl
                    {
                        //(">>>INIT PLOT<<<").postln;
                        //("startFrame: 0 | count:" + plotDur + "l
stopFrame:" + currentFrame).postln;

```

```

array.collect({|item, il ~bufPlot.value[i] = item;});
~bufPlot.updatePlots.refresh;
//~updateBufPlot_prevUpdateFrame =
~lastRecPhasePos; //set prevUpdateFrame to currentFrame
~updateBufPlot_prevUpdateFrame =
currentFrame; //set prevUpdateFrame to currentFrame
}.defer;
});
},
{//else if routine's been running: plot from prevUpdateFrame+1 to
currentFrame
var startFrame, currentFrame;

startFrame = ~updateBufPlot_prevUpdateFrame + 1;
if (startFrame >= ~audioBuf0.numFrames, {startFrame = 0;}); //if
~prevUpdateFrame == lastBufFrame, wrap around to 0
currentFrame = ~lastRecPhasePos; //store in local var b/c env var
frequently updated by other routine

//if(~lastRecPhasePos > startFrame,
if (currentFrame > startFrame,
{//simple contiguous span
var plotDur;
//plotDur = ~lastRecPhasePos -
~updateBufPlot_prevUpdateFrame; //inclusive of startFrame
plotDur = currentFrame -
~updateBufPlot_prevUpdateFrame; //inclusive of startFrame
count: plotDur, action: {larrayl
{
//(">>>SIMPLE
//("startFrame:" + startFrame +
array.collect({|item, il
~bufPlot.updatePlots.refresh;
//~updateBufPlot_prevUpdateF
rame = ~lastRecPhasePos; //set prevUpdateFrame to currentFrame
~updateBufPlot_prevUpdateFrame = currentFrame; //set prevUpdateFrame to currentFrame
}.defer;
});
},
{//wrapped around buffer, two chunks...
var chunk1Dur, chunk2Dur;
chunk1Dur = ~audioBuf0.numFrames -
startFrame; //dur from start frame to last buf frame (inclusive of startFrame)
//chunk2Dur = ~lastRecPhasePos + 1; //dur from
frame 0 to lastRecPhasePos (inclusive of frame 0)
chunk2Dur = currentFrame + 1; //dur from frame
0 to lastRecPhasePos (inclusive of frame 0)

//plot from startFrame to lastBufFrame
~audioBuf0.loadToFloatArray(index: startFrame,
count: chunk1Dur, action: {larrayl
{
//(">>>WRAP
//("CHUNK1 - startFrame:" +
startFrame + "l chunk1Dur:" + chunk1Dur).postln;
array.collect({|item, il
~bufPlot.value[startFrame + i] = item;});
}.defer;
});
});

```

```

//plot from frame 0 to ~lastRecPhasePos
~audioBuf0.loadToFloatArray(index: 0, count:
chunk2Dur, action: {arrayl
{
//("CHUNK2 - startFrame: 0" +
"I count:" + chunk2Dur + "I stopFrame:" + currentFrame).postln;
//("TOTAL - startFrame:" +
startFrame + "I dur:" + (chunk1Dur + chunk2Dur) + "I stopFrame:" + currentFrame).postln;
array.collect({litem, il
~bufPlot.value[i] = item;});
~bufPlot.updatePlots.refresh;
//~updateBufPlot_prevUpdateF
rame = ~lastRecPhasePos; //set prevUpdateFrame to currentFrame
~updateBufPlot_prevUpdateFrame = currentFrame; //set prevUpdateFrame to currentFrame
}.defer;
});
}
);
}
);
0.5.yieldAndReset;
});
});

//-----
/*~phaseBusMonRec.get({lval (val/s.sampleRate).postln;}); //junk for monitoring
~phaseBusRec.get({lval val.postln;});
~phaseBusMonRec.get({lval val.postln;});
~phaseBusMonRec.get({lval val.postln;});
~phaseBusMonPly.get({lval (val/s.sampleRate).postln;}); //<----this is where to get phasor value....
*/

////-----RECORD TO BUFFER BUTTON-----////
~button_recBuf = Button(~view_perfButtons, Rect(0, 0, 55, 25)) //REC BUTTON: activate / deactivate
recording to buffer
.states_( [
["REC", Color.black, Color(0.95, 0, 0, 0.3)] ,
["REC", Color.white, Color(0.95, 0, 0)]
] )
.font_(Font("Monaco", 18) )
.action_( {
arg obj;
if (
obj.value == 1, // when REC button enabled (start recording to buffer)
{
if ((~lastRecPhasePos >= ~recInFrame) && (~lastRecPhasePos <=
~recOutFrame), {
//if rec head is still w/in rec In/Out range
~phasor_rec = Synth(\phasor, [\out, ~phaseBusRec, \buf,
~audioBuf0, \rate, 1, \start, ~lastRecPhasePos, \end, ~recOutPosBus.asMap] ).moveToHead;
~recPosLatchTrig = true; //used to trigger reset of \start arg to
~recInPosBus.asMap after init
}, { //else ~lastRecPhasePos is outside of rec In/Out range
~phasor_rec = Synth(\phasor, [\out, ~phaseBusRec, \buf,
~audioBuf0, \rate, 1, \start, ~recInPosBus.asMap, \end, ~recOutPosBus.asMap] ).moveToHead; //have
phasor start at ~recInFrame
});

~a2k_rec = Synth(\a2k_patch, [\in, ~phaseBusRec, \out,
~phaseBusMonRec] ).moveAfter(~phasor_rec); //<--don't fully understand why this allows me to monitor
phasor value and why I can't do so by getting value of ~phaseBusRec (doing so always outputs 1)...
```



```

~bufRecorder = Synth(\bufRec, [\phaseBus, ~phaseBusRec, \buf,
~audioBuf0, \sendBus, ~inputSendBus, \loCut, ~loCutBus_in.asMap, \loCutQ, 2, \mul,
~inputLevelBus.asMap] ).moveAfter(~a2k_rec);

NodeWatcher.register(~bufRecorder); //this ensures all synths associated w/
buf play back are positioned after all these buf recording synths in the node-tree, no matter when they're
created

//~lastRecPhasePos.value.postln;

//START REC SLIDER UPDATE ROUTINE...
~getBus_updateGUI_recPhase.reset;
AppClock.play(~getBus_updateGUI_recPhase);

/////PLOTTER -----
if (~plotterEnabled == true, {
~updateBufPlot.reset;
AppClock.play(~updateBufPlot);
});

/*
//////////SCOPE
~scopeBuf_audioBuf0 = Buffer.alloc(s, ~audioBuf0.numFrames, 1); //create
additional buffer for displaying contents of audio buffer...is this the only way to do this?? what happens if
I 'scopeOut' to ~audioBuf0?
~scopeView.bufnum = ~scopeBuf_audioBuf0.bufnum; //<----
//~scopeView.bufnum = ~audioBuf0.bufnum; //<----this will also work (no
addtl buffer needed)...
~scopeOut = Synth.tail(s, \scopeOut, [\inBus, ~inputSendBus, \buf,
~scopeBuf_audioBuf0]); //<----
//~scopeOut = Synth.tail(s, \scopeOut, [\inBus, ~inputSendBus, \buf,
~audioBuf0]); //<--this works also
~scopeView.start; //<----
//////////
*/

},
{ //when REC button disabled (when recording stops)
~phaseBusMonRec.get( {arg val;
~lastRecPhasePos = val.round(1); //store last pos of recPhase
//("LastRecPhasePos = " ++ ~lastRecPhasePos).postln;
});
~phasor_rec.free;
~a2k_rec.free;
~bufRecorder.free; //stop recording to buffer and free RecBuf_loop synth

~getBus_updateGUI_recPhase.stop;

if (~plotterEnabled == true, {~updateBufPlot.stop;}); //PLOTTER
if (~syncRecPlayOffset == true, {~syncRecPlayOffset = false;
~button_delaySync.value = 0;}); //disable sync
/*
//FOR SCOPE
~scopeView.stop; //<----
~scopeOut.free; //<----
~scopeBuf_audioBuf0.free; //<----
*/
}

);
//-----

////----- PLAY FROM BUFFER BUTTON -----////

```

```

~button_playBuf = Button(~view_perfButtons, Rect(56, 0, 55, 25)) //activate / deactivate playback output
.states_( [
    ["PLAY", Color.black, Color(0.0, 0.8, 0.2, 0.3)] ,
    ["PLAY", Color.white, Color(0.0, 0.8, 0.2)]
] )
.font_(Font("Monaco", 18) )
.action_( {
    arg obj;
    if (obj.value == 1,
        {
            if (~syncRecPlayOffset == true, {~recPlayDelayOffset.(offset: ~delayTime);});
//sync playhead if SYNC enabled
            if ((~lastPlayPhasePos >= ~playInFrame) && (~lastPlayPhasePos <=
~playOutFrame), {
                //if play head is still w/in play In/Out range
                ~phasor_play = Synth(\phasor, \out, ~phaseBusPly, \buf,
~audioBuf0, \rate, ~rateBus.asMap, \start, ~lastPlayPhasePos, \end, ~playOutPosBus.asMap] );
                ~playPosLatchTrig = true; //used to start playPhasor at
lastPlayPhasePos
            }, { //else ~lastPlayPhasePos is outside of play In/Out range
                ~phasor_play = Synth(\phasor, \out, ~phaseBusPly, \buf,
~audioBuf0, \rate, ~rateBus.asMap, \start, ~playInPosBus.asMap, \end, ~playOutPosBus.asMap] );
                //have phasor start at ~playInFrame
            });

            if (~bufRecorder.isRunning, {~phasor_play.moveAfter(~bufRecorder);});
//regulate node-tree order

            ~a2k_play = Synth(\a2k_patch, [\in, ~phaseBusPly, \out,
~phaseBusMonPly]).moveAfter(~phasor_play); //converting phaseBus values at ar to kr and sending to
phaseBusMon so sliderPlayPos can receive current value of phasor
            ~bufPlayer = Synth(\bufRead, [\amp, ~play1LevelBus.asMap, \out,
~playOut1Bus, \phaseBus, ~phaseBusPly, \buf, ~audioBuf0] ).moveAfter(~a2k_play);

            NodeWatcher.register(~bufPlayer);

            ~pShift = Synth(\pitchShifter,
                [\inBus, ~playOut1Bus, \outBus, ~pitchShiftOut, \pitchRatio,
~pitchBus.asMap]).moveAfter(~bufPlayer);

            ~pShiftMix = Synth(\XFade, [
                \in1, ~playOut1Bus,
                \in2, ~pitchShiftOut,
                \out, ~outMainBus,
                \out, ~stream1sub, //<---NEW
                \amountControlBus, ~pitchShiftMixValBus.asMap,
                \amp, ~inputStreamLevelBus.asMap
            ]).moveAfter(~pShift);

            ~getBus_updateGUI_playPhase.reset;
            AppClock.play(~getBus_updateGUI_playPhase);

        },
        {
            ~phaseBusMonPly.get( {Ival ~lastPlayPhasePos = val.round(1);} );
            ~phasor_play.free;
            ~a2k_play.free;
            ~bufPlayer.free;
            ~pShift.free;
            ~pShiftMix.free;
            ~getBus_updateGUI_playPhase.stop;
        }
    );
});

//-----REVERSE BUTTON -----

```

```

~button_rev = Button(~view_perfButtons, Rect(113, 0, 45, 25)) //Reverse playback
.states_( [
    ["REV", Color.black, Color(0.8, 0.5, 0, 0.3)] ,
    ["REV", Color.white, Color(0.85, 0.5, 0)]
] )
.font_(Font("Monaco", 18) )
.action_( {
    arg obj;
    if (
        obj.value == 1,
        { //reverse playback when enabled
            ~rateBus.get({
                arg val;
                if (val > 0, {
                    var newRate = -1 * val;
                    //("newRate=" + newRate).postln;
                    ~rateBus.set(newRate);
                });
            });
            ~revRateBusMon.set(-1);
            //~reverseFlag = true;
            if (~syncRecPlayOffset == true, {~syncRecPlayOffset = false;
~button_delaySync.value = 0;}); //disable sync
        },
        { //normal playback when disabled
            ~rateBus.get( {
                arg val;
                var newRate = val.abs;
                //("newRate=" + newRate).postln;
                ~rateBus.set(newRate);
            });
            ~revRateBusMon.set(1);
            //~reverseFlag = false;
        }
    );
})
.value_(0);

//----- DELAY OFFSET -----
~button_delaySync = Button(~view_perfButtons, Rect(160, 0, 55, 25))
.states_( [
    ["SYNC", Color.black, Color(0.7, 0.7, 0, 0.3)] ,
    ["SYNC", Color.white, Color(0.8, 0.8, 0)]
] )
.font_(Font("Monaco", 18) )
.action_( {
    arg obj;
    if (
        obj.value == 1, // when sync button enabled
        {
            var currentRate, currentRev;
            ~syncRecPlayOffset = true;

            currentRate = ~rateBus.getSynchronous.round(0.01);
            if (currentRate != 1,
                {
                    ~rateBus.set(1);
                    ~slider_rate.value_(0.5);
                    ~numBox_rate.value_(0);
                }
            );

            currentRev = ~revRateBusMon.getSynchronous;
            if (currentRev == -1,
                {

```

```

~revRateBusMon.set(1);
~button_rev.value = 0;
    }
);

~recPlayDelayOffset.(offset: ~delayTime);

},
{ //when button disabled
  ~syncRecPlayOffset= false;
}
);
})
.value_(0);

~numBox_delay = NumberBox(~view_perfButtons, Rect(217, 0, 55, 25))
.value_(~delayTime)
.clipLo_(0)
.clipHi_(~audioBuf0_dur - 0.01)
.font_(Font("Monaco", 14) )
.decimals_(2)
.scroll_step_(0.01)
.step_(0.01)
.action_( {obj|
  ~delayTime = obj.value.round(0.01);
  if (~syncRecPlayOffset == true, {~recPlayDelayOffset.(offset: ~delayTime);}); //sync playhead if
  SYNC enabled
});

~label_delayTime = StaticText(~view_perfButtons, Rect(275, 2, 78, 20) )
//~label_delayTime = StaticText(w, Rect(720, 157, 78, 20) )
.string_("Offset
(sec)")
.font_(Font("Monaco", 8) )
.align_(\left);

//-----

/*
//Monitoring support GUI objects for ~button_Behavior1 activity
~numBox_counter_rateT2rand = NumberBox(w, Rect(313, 378, 50, 25))
.value_(0.0)
.clipLo_(0)
.clipHi_(120)
.font_(Font("Monaco", 14) )
.decimals_(1);

~label_counter_rateT2rand = StaticText(w, Rect(368, 380, 30, 20) )
.string_("T2R")
.font_(Font("Monaco", 12) )
.align_(\left);
*/

//----- BEHAVIOR BUTTONS -----

~button_Behavior1 = Button(~view_perfButtons, Rect(0, 25, 55, 25)) //randomize rate and pitchShift
over time
.states_( [
  ["!!!", Color.black, Color(0.7, 0.3, 0.7, 0.3)] ,
  ["!!!", Color.white, Color(0.8, 0.3, 0.8)]
] )
.font_(Font("Monaco", 18) )
.action_( {
  arg obj;
  if (

```

```

        obj.value == 1, // when randomize button enabled
    {
        ~mode_randProcLiveBuf = true; //<---NEW mode flag

        ~randomRoutine_rate.reset;
        ~randomRoutine_freq.reset;
        ~randomRoutine_pitchShiftMix.reset;
        ~randomRoutine_reverse.reset;
        ~randomRoutine_playHeadBehavior.reset;

        AppClock.play(~randomRoutine_rate);
        AppClock.play(~randomRoutine_freq);
        AppClock.play(~randomRoutine_pitchShiftMix);
        AppClock.play(~randomRoutine_reverse);
        AppClock.play(~randomRoutine_playHeadBehavior); //<---spruce this
routine up to be consistent w/ above
    },
    { //when randomize button disabled
        ~mode_randProcLiveBuf = false; //<---NEW mode flag

        ~randomRoutine_rate.stop;
        ~updateGUI_rate.stop;

        ~randomRoutine_freq.stop;
        ~updateGUI_freq.stop;

        ~randomRoutine_pitchShiftMix.stop;
        ~updateGUI_pitchShiftMix.stop;

        ~randomRoutine_reverse.stop;
        ~randomRoutine_playHeadBehavior.stop;
        ~playHeadFollowRecHead.stop;
    }
);
})
.value_(0);

~button_Behavior2 = Button(~view_perfButtons, Rect(55, 25, 30, 25)) //Randomize pitch shift, pitch shift
mix, and output level control
.states_( [
    ["!!", Color.black, Color(0.0, 0.3, 0.9, 0.3)] ,
    ["!!", Color.white, Color(0.8, 0.3, 0.8)]
] )
.font_(Font("Monaco", 18) )
.action_( {
    arg obj;
    if (
        obj.value == 1, // when button enabled
    {
        },
    { //when button disabled
        }
    )
);
})
.value_(0);

~button_Behavior3 = Button(~view_perfButtons, Rect(86, 25, 25, 25)) //randomize Reverse and
PlayHead Behavior control
.states_( [
    ["!", Color.black, Color(0.6, 0.9, 0.0, 0.3)] ,
    ["!", Color.white, Color(0.8, 1.0, 0.0)]
] )
.font_(Font("Monaco", 18) )

```

```

.action_( {
    arg obj;
    if (
        obj.value == 1, // when button enabled
        {
            ~listen_sound.value(active: 1, input: ~inputSendBus, outBus: \busSum_lop,
sumSynth: \sumSyn_1, post: 0); //START ~listen_sound ROUTINE <-----
            //~autoConvolve1A.(1, featureBus: ~busSum_lop, outBus: ~stream2sub,
ampBus: ~fileOutputLevelBus.asMap, rout1: \autoC1, rout2: \autoC2, initFlag: \autoC1state,
repeatCounter: \autoCnt1, synth1: \autoS1, preRecPad: 0.1, postRecPad: 0, impulse: 1, freq1: 400,
freq2: 3000, attTime: 5, susTime: 3, relTime: 6, impPostDel: 0.5, timeStretch: true, timeStretchRate:
0.75, post: 1); //<---solo laptop preset
            ~autoConvolve1A.(1, featureBus: ~busSum_lop, outBus: ~outMainBus,
ampBus: ~fileOutputLevelBus.asMap, rout1: \autoC1, rout2: \autoC2, initFlag: \autoC1state,
repeatCounter: \autoCnt1, synth1: \autoS1, preRecPad: 0.2, postRecPad: 1, impulse: 3, impPostDel: 0,
timeStretch: true, timeStretchRate: 0.67, post: 1); //<---adjust pre and post pads to accommodate reverb
tail of performance space and audio content (e.g. speech vs. claps, etc...) ---LIGHT LAB SETTINGS
<-----

            //~autoConvolve1A.(1, filePath: ~fileAudio_writeln_lastFile, featureBus:
~busSum_lop, outBus: ~stream2sub, ampBus: ~fileOutputLevelBus.asMap, rout1: \autoC1, rout2:
\autoC2, initFlag: \autoC1state, repeatCounter: \autoCnt1, synth1: \autoS1, preRecPad: 0, postRecPad:
0, impulse: 0, freq1: 600, freq2: 606, attTime: 0.5, susTime: 2.5, relTime: 2, impPostDel: 0.4,
timeStretch: true, timeStretchRate: 0.75, post: 1); //<---FOR VOXELS

            //~autoConvolve2_replim_voxels.(1, filePath: ~fileAudio_writeln_lastFile,
featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus: ~fileOutputLevelBus.asMap, rout1: \autoC1,
rout2: \autoC2, initFlag: \autoC1state, repeatCounter: \autoCnt1, repeatLimit: 7, synth1: \autoS1,
preRecPad: 0, impulse: 0, freq1: 600, freq2: 606, attTime: 0.5, susTime: 3.5, relTime:
2.5, impPostDel: 0.4, timeStretch: true, timeStretchRate: 0.75, post: 1); //<---FOR VOXELS _ 0
            //~autoConvolve2_replim_voxels.(1, filePath: ~fileAudio_writeln_lastFile,
featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus: ~fileOutputLevelBus.asMap, rout1: \autoC1,
rout2: \autoC2, initFlag: \autoC1state, repeatCounter: \autoCnt1, repeatLimit: 8, synth1: \autoS1,
preRecPad: 0, postRecPad: 0, impulse: 2, freq1: 100, freq2: 20000, susTime: 0.5, impPostDel: 0,
timeStretch: true, timeStretchRate: 0.6, post: 1); //<---FOR VOXELS _ 1

            //~autoConvolve2_replim_voxels.(1, filePath: ~fileAudio_writeln_lastFile,
featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus: ~fileOutputLevelBus.asMap, rout1: \autoC1,
rout2: \autoC2, initFlag: \autoC1state, repeatCounter: \autoCnt1, repeatLimit: 4, synth1: \autoS1,
preRecPad: 0, postRecPad: 0, impulse: 1.1, freq1: 250, freq2: 1000, attTime: 1.67, susTime: 2.24,
relTime: 3.87, impPostDel: 0, timeStretch: true, timeStretchRate: 0.6, post: 1); //<---FOR VOXELS _ 2

            //~autoConvolve2_replim_voxels.(1, filePath: ~fileAudio_writeln_lastFile,
featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus: ~fileOutputLevelBus.asMap, rout1: \autoC1,
rout2: \autoC2, initFlag: \autoC1state, repeatCounter: \autoCnt1, repeatLimit: 7, synth1: \autoS1,
preRecPad: 0, postRecPad: 0, impulse: 1.2, freq1: 180, freq2: 850,/* attTime: 1.67, susTime: 3.59,
relTime: 3.87,*/ impPostDel: 0, timeStretch: true, timeStretchRate: 0.6, post: 1); //<---FOR VOXELS _ 3

            //~autoConvolve1A.(1, featureBus: ~busSum_lop, outBus: ~stream2sub,
ampBus: ~fileOutputLevelBus.asMap, rout1: \autoC1, rout2: \autoC2, initFlag: \autoC1state,
repeatCounter: \autoCnt1, synth1: \autoS1, preRecPad: 0.2, postRecPad: 0.2, impulse: 3, impPostDel:
0.4, timeStretch: true, timeStretchRate: 0.75, post: 1); //<---piezo live sound settings
            //~autoConvolve2_replim_selfTerminate.(1, activeFlag: \autoConvActive,
filePath: ~fileAudio_writeln_lastFile, featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus:
~fileOutputLevelBus.asMap, rout1: \autoC1, rout2: \autoC2, initFlag: \autoC1state, repeatCounter:
\autoCnt1, repeatLimit: 3, selfTerminate: true, synth1: \autoS1, preRecPad: 0, postRecPad: 0, impulse:
0, attTime: 0.5, susTime: 3.5, relTime: 2.5, impPostDel: 0.2, timeStretch: true, timeStretchRate: 0.75,
post: 1); //<---FOR VOXELS _ 0
            //~autoConvolve2_replim_selfTerminate.(1, activeFlag: \autoConvActive,
filePath: ~fileAudio_writeln_lastFile, featureBus: ~busSum_lop, outBus: ~stream2sub, ampBus:
~fileOutputLevelBus.asMap, rout1: \autoC1, rout2: \autoC2, initFlag: \autoC1state, repeatCounter:
\autoCnt1, repeatLimit: 3, selfTerminate: false, synth1: \autoS1, preRecPad: 0, postRecPad: 0, impulse:
1.1, freq1: 300, freq2: 850, impPostDel: 0, timeStretch: true, timeStretchRate: 0.65, post: 1);

```

```

        //~autoConvolveSequence_voxelBot1.(1, initFlag: \autoConvSeqInit,
impulseFlag: \autoConvSeqImpulse, routine: \autoConvSeqRout, post: 1);
    },
    { //when button disabled
        ~listen_sound.value(active: 0, outBus: \busSum_lop, sumSynth:
\sumSyn_1);
        ~autoConvolve1A.(0, rout1: \autoC1, rout2: \autoC2, synth1: \autoS1); //this
will throw error if envName isn't provided for rout1 arg.....

        //~autoConvolve2_repLim_voxels.(0, rout1: \autoC1, rout2: \autoC2,
synth1: \autoS1);
        //~autoConvolve2_repLim_selfTerminate.(0, rout1: \autoC1, rout2: \autoC2,
synth1: \autoS1);

        //~autoConvolveSequence_voxelBot1.(0, routine: \autoConvSeqRout, post:
1);
    }
);
})
.value_(0);
//-----

// ----- REC HEAD / PLAY HEAD -----

~slider_recPos = Slider(~view_transport, Rect(5, 25, 610, 20))
.background_(Color(0.95, 0.0, 0.0))
.action_( {
    arg obj;
    var pos;
    pos = obj.value.linlin(0, 1, 0, ~audioBuf0.numFrames - 1).round(1); //convert slider value to
frame

    if (~button_recBuf.value == 1,
        { //update ~phasor_rec phasor synth position when actively recording to buffer
            ~phasor_rec.set (\resetPos, pos, \trig, 1);
            if (~syncRecPlayOffset == true, {{~recPlayDelayOffset.(offset:
~delayTime)}}.defer(0.01)); //sync playhead if SYNC enabled (defer is perhaps not elegant, but
necessary b/c otherwise ~recPlayDelayOffset uses current/old/nonupdated rec phase bus position
when ~phaseBusMonRec.getSynch call is made...
        },
        { //update phasor pos when not actively recording to buffer
            ~phaseBusMonRec.set(pos); //<--- update control bus too, which other funcs
and routines use
            ~lastRecPhasePos = pos;
        }
    );

    ~numBox_recPos.value = (pos/s.sampleRate).round(0.01); //convert frames to seconds and
display

})
.value_(0.0);

~numBox_recPos = NumberBox(~view_transport, Rect(625, 25, 65, 20)) //numBox PlayPos
.value_(0)
.clipLo_(0)
//.clipHi_(~audioBuf0_dur) //HOW COME UNCOMMENTING THIS MESSES UP THE GUI?????
.font_(Font("Monaco", 14) )
.decimals_(2);
//.action_( {
    //arg obj;
    //~slider_recPos.valueAction = obj.value / ~audioBuf0_dur; //<---
//});

~label_recPos = StaticText(~view_transport, Rect(692, 32, 100, 10) )

```

```

.string_("RecPos(sec)")
.font_(Font("Monaco", 10) )
.align_(\left);

~slider_playPos = Slider(~view_transport, Rect(5, 48, 610, 20))
.background_(Color(0, 0.8, 0.2))
.action_( {
    arg obj;
    var pos;
    pos = obj.value.linlin(0, 1, 0, ~audioBuf0.numFrames - 1).round(1); //convert slider value to
frame
    //~phasor_play.set (\resetPos, pos, \trig, 1);
    if (~button_playBuf.value == 1,
        { //update ~phasor_play phasor synth position when actively playing from buffer
            ~phasor_play.set(\resetPos, pos, \trig, 1);
            if (~syncRecPlayOffset == true, {~syncRecPlayOffset = false;
~button_delaySync.value = 0;}); //disable sync
        },
        { //update phasor pos when not actively playing from buffer
            ~phaseBusMonPly.set(pos); //<--- update control bus too, which other funcs
and routines use
            ~lastPlayPhasePos = pos;
        }
    );
    ~numBox_playPos.value = (pos/s.sampleRate).round(0.01); //convert frames to seconds and
display
})
.value_(0.0);

~numBox_playPos = NumberBox(~view_transport, Rect(625, 48, 65, 20))
.value_(0)
.clipLo_(0)
//.clipHi_(~audioBuf0.numFrames-1)
.font_(Font("Monaco", 14) )
.decimals_(2);

~label_playPos = StaticText(~view_transport, Rect(692, 53, 100, 10) )
.string_("PlayPos(sec)")
.font_(Font("Monaco", 10) )
.align_(\left);

~slider_recInOut = RangeSlider(~view_transport, Rect(5, 2, 610, 20))
.lo_(0.0)
.hi_(1.0)
//.background_(Color(0.0, 0.5, 0.2))
.action_( {
    |slider|
    var recIn, recOut;
    recIn = slider.lo.round(0.0001).linlin(0, 1.0, 0, ~audioBuf0.numFrames - 1).round(1);
    ("recIn=" + recIn).postln;
    recOut = slider.hi.round(0.0001).linlin(0, 1.0, 0, ~audioBuf0.numFrames - 1).round(1);
    ("recOut=" + recOut).postln;
    ~recInPosBus.set(recIn);
    ~recInFrame = recIn;
    ("RecInFrame=" + ~recInFrame).postln;
    ~recOutPosBus.set(recOut);
    ~recOutFrame = recOut;
    ("RecOutFrame=" + ~recOutFrame).postln;
    //~numBox_playIn.value_((playIn * ~audioBuf0_dur).round(0.1));
    ~numBox_recIn.value_( (recIn / s.sampleRate).round(0.01) ); //
    //~numBox_playIn.value_( (slider.lo.round(0.0001) * ~audioBuf0_dur).round(0.1) );
    //~numBox_playOut.value_((playOut * ~audioBuf0_dur).round(0.1));
    ~numBox_recOut.value_( (recOut / s.sampleRate).round(0.01) );
});

```



```

~numBox_recIn = NumberBox(~view_transport, Rect(625, 2, 46, 20))
.value_(0.0)
.clipLo_(0.0)
//.clipHi_(~audioBuf0_dur)
.clipHi_(~numBox_audioBuf0_dur.clipHi)
.font_(Font("Monaco", 12) )
.decimals_(2);
/*
.action_( {
    arg obj;
    ~slider_playInOut.lo.valueAction_(obj.value); //updates value & position of slider and triggers
    action when numberBox value is changed...
} ); */

~numBox_recOut = NumberBox(~view_transport, Rect(672, 2, 46, 20))
.value_(~audioBuf0_dur)
.clipLo_(0.0)
//.clipHi_(~audioBuf0_dur)
.clipHi_(~numBox_audioBuf0_dur.clipHi)
.font_(Font("Monaco", 12) )
.decimals_(2);
/*
.action_( {
    arg obj;
    ~slider_playInOut.lo.valueAction_(obj.value); //updates value & position of slider and triggers
    action when numberBox value is changed...
} ); */

~label_recInOut = StaticText(~view_transport, Rect(720, 3, 90, 20) )
.string_("In/Out-Rec")
.font_(Font("Monaco", 10) )
.align_(\left);

~slider_playInOut = RangeSlider(~view_transport, Rect(5, 71, 610, 20)) //SLIDER => Play In/Out
.lo_(0.0)
.hi_(1.0)
//.background_(Color(0.0, 0.5, 0.2))
.action_({
    |slider|

    var playIn, playOut;

    //playIn = slider.lo.round(0.0001);
    playIn = slider.lo.round(0.0001).linlin(0, 1.0, 0, ~audioBuf0.numFrames - 1).round(1);
    ("playIn=" + playIn).postln;
    //playOut = slider.hi.round(0.0001);
    playOut = slider.hi.round(0.0001).linlin(0, 1.0, 0, ~audioBuf0.numFrames - 1).round(1);
    ("playOut=" + playOut).postln;

    ~playInPosBus.set(playIn);
    //~playInFrame = playIn.linlin(0, 1.0, 0, ~audioBuf0.numFrames - 1).round(1);
    ~playInFrame = playIn;
    ("currentInFrame=" + ~playInFrame).postln;
    ~playOutPosBus.set(playOut);
    //~playOutFrame = playOut.linlin(0, 1.0, 0, ~audioBuf0.numFrames - 1).round(1);
    ~playOutFrame = playOut;
    ("currentOutFrame=" + ~playOutFrame).postln;

    //~numBox_playIn.value_((playIn * ~audioBuf0_dur).round(0.1));
    ~numBox_playIn.value_( (playIn / s.sampleRate).round(0.01) ); //
    //~numBox_playIn.value_( (slider.lo.round(0.0001) * ~audioBuf0_dur).round(0.1) );

    //~numBox_playOut.value_((playOut * ~audioBuf0_dur).round(0.1));
    ~numBox_playOut.value_( (playOut / s.sampleRate).round(0.01) );

```

```

});

~numBox_playIn = NumberBox(~view_transport, Rect(625, 71, 46, 20))
.value_(0.0)
.clipLo_(0.0)
//.clipHi_(~audioBuf0_dur)
.clipHi_(~numBox_audioBuf0_dur.clipHi)
.font_(Font("Monaco", 12) )
.decimals_(2);
/*
.action_( {
    arg obj;
    ~slider_playInOut.lo.valueAction_(obj.value); //updates value & position of slider and triggers
action when numberBox value is changed...
} ); */

~numBox_playOut = NumberBox(~view_transport, Rect(672, 71, 46, 20))
.value_(~audioBuf0_dur)
.clipLo_(0.0)
//.clipHi_(~audioBuf0_dur)
.clipHi_(~numBox_audioBuf0_dur.clipHi)
.font_(Font("Monaco", 12) )
.decimals_(2)
/*
.action_( {
    arg obj;
    ~slider_playInOut.lo.valueAction_(obj.value); //updates value & position of slider and triggers
action when numberBox value is changed...
} ); */

~label_playInOut = StaticText(~view_transport, Rect(720, 72, 90, 20) )
.string_("In/Out-Play")
.font_(Font("Monaco", 10) )
.align_(\left);
//-----

// ----- OTHER PLAYBACK CONTROLS -----

//RATE --
~slider_rate = Slider(~view_procControls1, Rect(0, 0, 300, 20))
.background_(Color(0.0, 0.5, 0.9))
.value_(0.5)
.action_({|obj|
    var rateTransp = obj.value.linlin(0, 1, -48, 48).round(0.01);
    if (~button_rev.value == 1,
        {~rateBus.set(rateTransp.midiratio * -1);}, //negative rate when REV button enabled
        {~rateBus.set(rateTransp.midiratio);} //positive rate when REV button disabled
    );
    ~numBox_rate.value_(rateTransp); //update numBoxPitch value
    ~pitchShiftValPrev_rate = rateTransp; //<---NEW, to support ~randomRoutine_reverse
    if (rateTransp != 0, {
        if (~syncRecPlayOffset == true, {~syncRecPlayOffset = false;
~button_delaySync.value = 0;}); //disable sync
    });
});
~numBox_rate = NumberBox(~view_procControls1, Rect(308, 0, 48, 20))
.value_(0)
.clipLo_(-48)
.clipHi_(48)
.font_(Font("Monaco", 12) )
.decimals_(2)
.scroll_step_(0.01)
.step_(0.01)
.action_( {
    arg obj;

```

```

var semiToneVal, pos;
semiToneVal = obj.value.clip(-48, 48).round(0.01);
if (~button_rev.value == 1,
    {~rateBus.set(semiToneVal.midiratio * -1);}, //negative rate when REV button enabled
    {~rateBus.set(semiToneVal.midiratio);} //positive rate when REV button disabled
);
pos = obj.value.clip(-48, 48).linlin(-48, 48, 0, 1).round(0.001);
~slider_rate.value_(pos);
~pitchShiftValPrev_rate = semiToneVal; //<---NEW, to support ~randomRoutine_reverse
if (semiToneVal != 0, {
    if (~syncRecPlayOffset == true, {~syncRecPlayOffset = false;
~button_delaySync.value = 0;}); //disable sync
});
});

~label_rate = StaticText(~view_procControls1, Rect(358, 4, 80, 10))
.string_("Rate(semi)")
.font_(Font("Monaco", 12) )
.align_(\left);

//PITCH SHIFT --
~slider_pitch = Slider(~view_procControls1, Rect(0, 22, 300, 20))
.background_(Color(0.0, 0.7, 0.9))
.value_(0.linlin(-48, 24, 0.0, 1.0))
.action_( {
    arg obj;
    var pos;
    pos = obj.value.linlin(0, 1, -48, 24).round(0.01);
    ~pitchBus.set(pos.midiratio);
    ~numBox_pitch.value_(pos);
    ~pitchShiftValPrev_freq = pos; //<---new, to support ~randomRoutine_pitchShiftMix
});

~numBox_pitch = NumberBox(~view_procControls1, Rect(308, 22, 48, 20))
.value_(0)
.clipLo_(-48)
.clipHi_(24)
.font_(Font("Monaco", 12) )
.decimals_(2)
.scroll_step_(0.01)
.step_(0.01)
.action_( {
    arg obj;
    var semiToneVal, pos;
    semiToneVal = obj.value.clip(-48, 24).round(0.01);
    ~pitchBus.set(semiToneVal.midiratio); //change pitch (in semi tones)
    pos = obj.value.linlin(-48, 24, 0, 1).round(0.001);
    ~slider_pitch.value_(pos);
    ~pitchShiftValPrev_freq = semiToneVal; //<---new, to support ~randomRoutine_pitchShiftMix
});

~label_pitch = StaticText(~view_procControls1, Rect(358, 27, 80, 10))
.string_("Pitch(semi)")
.font_(Font("Monaco", 12) )
.align_(\left);

// PITCH SHIFT MIX --
~slider_pitchShiftMix = Slider(~view_procControls1, Rect(165, 44, 135, 15))
.background_(Color(0.0, 0.6, 0.8, 0.2))
.value_(0.0)
.action_( {
    arg obj;
    var mixAmount;
    mixAmount = obj.value.round(0.001);

```

```

~pitchShiftMixValBus.set(mixAmount);
//("sliderVal2PBus=" + mixAmount).postln;
mixAmount = mixAmount.round(0.01);
~numBox_pitchShiftMix.value_(mixAmount * 100);
});

~numBox_pitchShiftMix = NumberBox(~view_procControls1, Rect(308, 44, 48, 20))
.value_(0)
.clipLo_(0)
.clipHi_(100)
.font_(Font("Monaco", 12) )
.decimals_(0)
.scroll_step_(1)
.step_(1)
.action_( {
    arg obj;
    var mixVal, pos;
    mixVal = obj.value.clip(0, 100).round(1);
    //("mixValIn=" + mixVal).post;
    mixVal = mixVal/100;
    //(" | mixValToPbus=" + mixVal).postln;
    ~pitchShiftMixValBus.set(mixVal); //change pitch (in semi tones)
    ~slider_pitchShiftMix.value_(mixVal);
});

~label_pitchShiftMix = StaticText(~view_procControls1, Rect(358, 49, 135, 10))
.string_("PitchShiftMix")
.font_(Font("Monaco", 12) )
.align_(\left);
};
);

```

File – “AutoBoot.scd”:

```

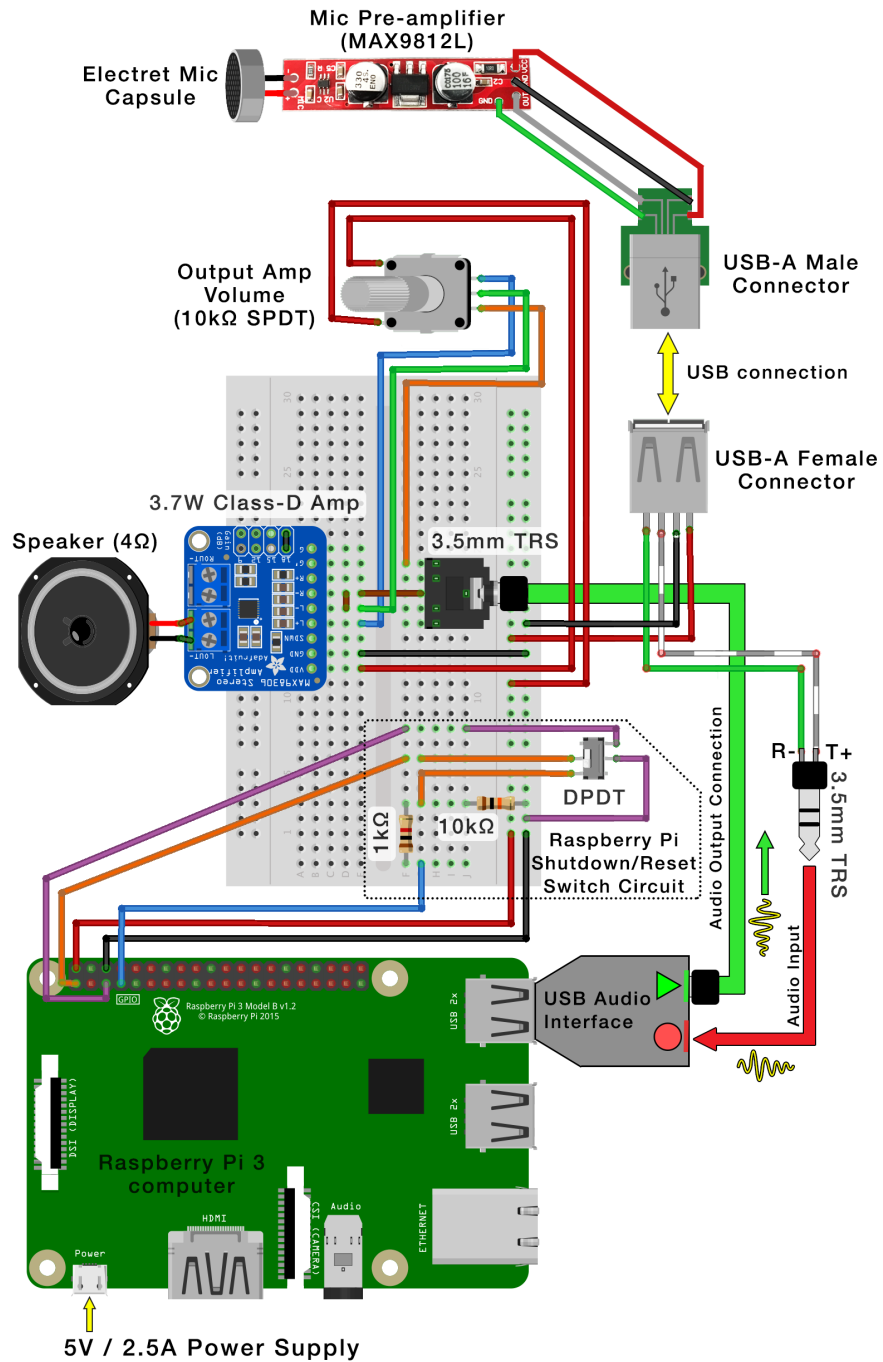
//Sonic Mirror AutoBoot

//evaluate to launch and configure SM automatically

s.waitForBoot({
    ~startup = Routine({
        ~dir = "/Users/Scott/Dropbox/DocSync/Project Docs/DigitalArt/SonicMirror/sonic-
mirror_SC_git/dev/";
        0.25.yield;
        (~dir ++ "02_SynthDefs.scd").load;    // SynthDefs
        0.25.yield;
        (~dir ++ "03_Busses.scd").load;       // Busses
        0.25.yield;
        (~dir ++ "04_Functions.scd").load;    // Functions and Routines
        (~dir ++ "05_InitSM.scd").load;       // Main SM Initialization Code
        0.25.yield;
        ~init_SonicMirror();                  //Launch
        0.25.yield;
        AppClock.play(~initSMpreset_2); //configure preset settings (these presets live in
"04_Functions.scd")
    });
    AppClock.play(~startup);
});
}

```

Sonic Mirror Hardware Instrument Circuit (Implementation 1)

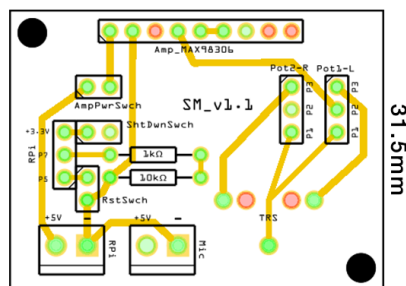


Sonic Mirror Hardware Instrument – Custom PCB (31.5 x 42mm)

Multi-purpose circuit that:

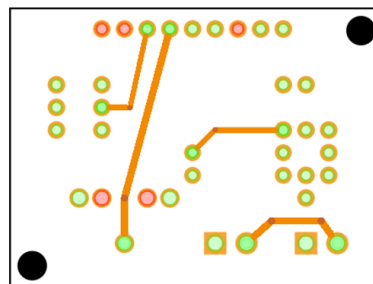
- connects audio output to speaker amp
- breaks out power and volume control connections of speaker amp
- breaks out Raspberry Pi GPIO pins for hardware switch soft-shutdown and reset functionality
- connects Raspberry Pi power supply to microphone pre-amp power input

Top:

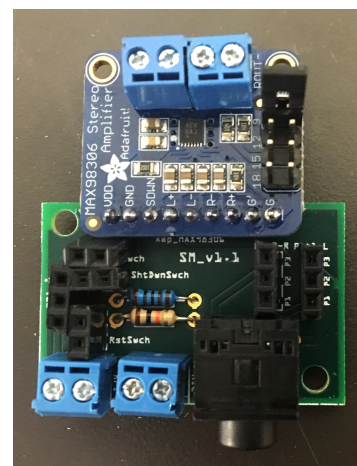
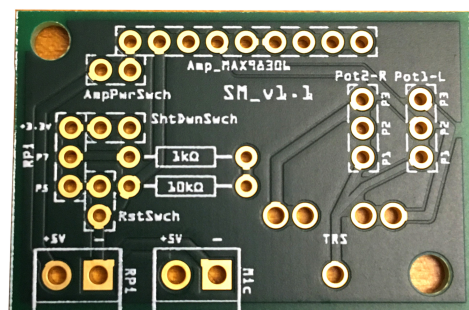


31.5mm

Bottom:

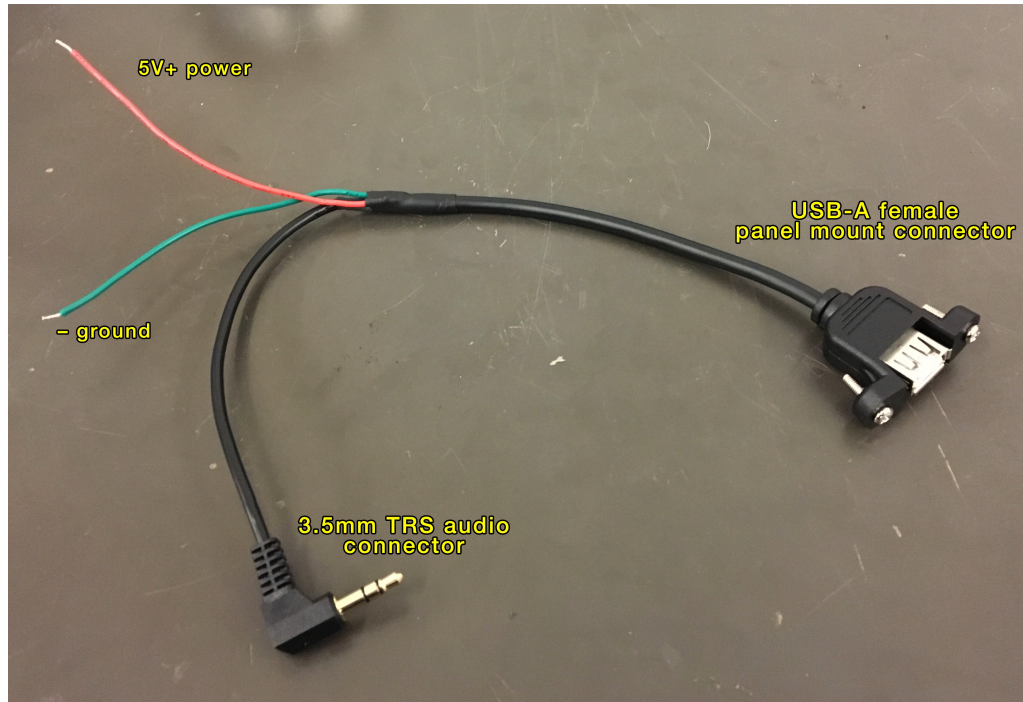


42mm



Sonic Mirror Hardware Instrument – Modified Panel Mount USB-A Female Audio Cable:

Custom cable that connects the microphone's audio output to the audio interface, and the Raspberry Pi's power supply to the microphone pre-amp circuit.

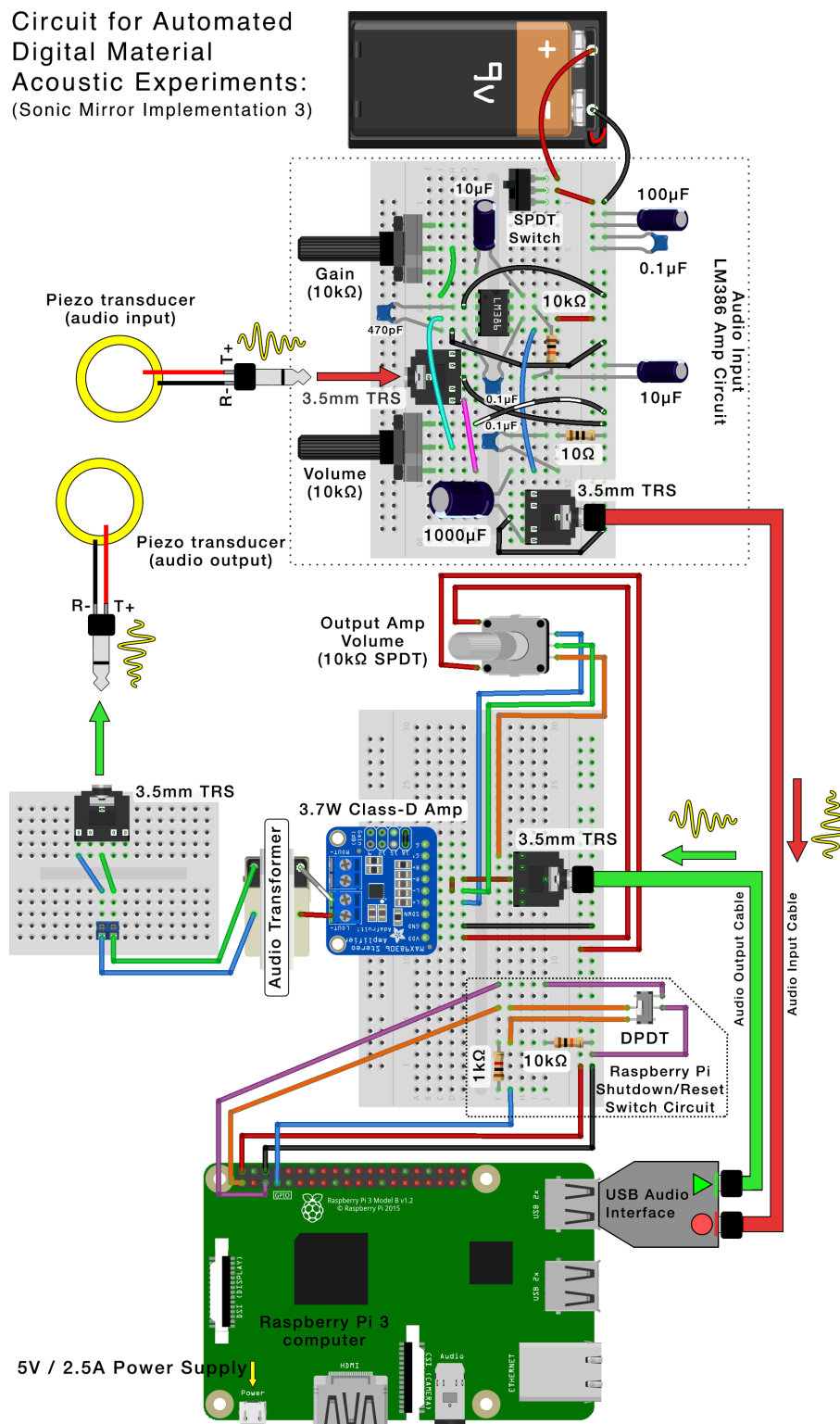


Sonic Mirror Hardware Instrument Component List:

(Implementation 1)

- Electret condenser microphone capsule (9.5mm)
- Microphone Pre-Amplifier (MAX 9812L)
- Speaker Driver (50W / 3.5in / 4Ω)
- Speaker Amplifier (3.7W / Class-D / MAX 98306)
- Sabrent USB Soundcard (audio interface w/ stereo input and output)
 - “Sabrent USB External Stereo Sound Adapter for Windows and Mac”
 - 48kHz / 24-bit (sampling rate / bit-depth)
- Raspberry Pi 3
- DPDT Switch (on-off-on shutdown & reset switch for Raspberry Pi)
- 10kΩ Potentiometer Switch (as speaker amp power switch and volume control)
- Custom PCB (31.5 x 42mm)
- Modified Panel Mount USB-A Female Microphone Cable
- Panel Mount Extension USB Cable - Micro-B Female to Micro-B Male (for Pi power supply)
- Audio Cable (male TRS to male TRS / 6in.)
- 3.5mm TRS audio jack
- (5x) Female to Male Jumper Wire Cables
 - connects Raspberry Pi's GPIO pins to Custom PCB
- 5V 2.5A power supply (for Raspberry Pi)

Circuit for Automated Digital Material Acoustic Experiments: (Sonic Mirror Implementation 3)



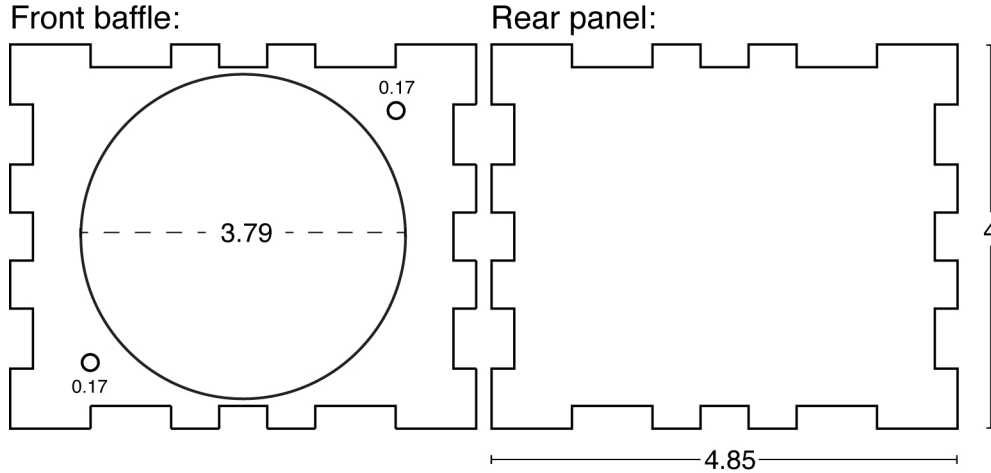
Digital Material Acoustic Experiment Circuit Component List:

Sonic Mirror Piezo Transducer Interface (Implementation 3)

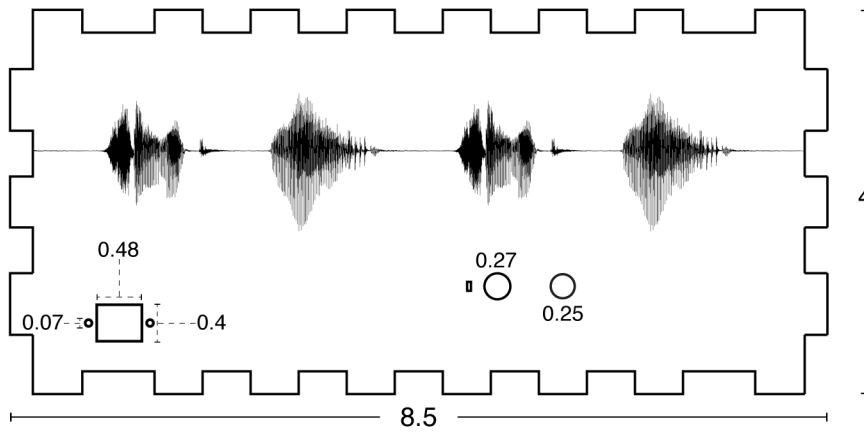
- Piezoelectric disc transducers:
 - 35mm and 15mm diameters piezoelectric discs
 - (2x) 3.5mm TS audio connectors
 - (2x) unbalanced audio cable
- LM386 Amplifier Circuit acting as Pre-Amplifier for piezo input
 - modeled after this LM386 circuit design (Circuit Basics, 2017)
 - Resistors: 10 Ω and 10k Ω
 - Capacitors: 470pF, (3x) 0.1 μ F, (2x) 10 μ F, 100 μ F, 1000 μ F
 - (2x) 10k Ω potentiometers
 - SPDT switch
 - LM386 op-amp
 - (2x) 3.5mm TRS audio jack
- Audio Output Transformer:
 - Impedance (input: 1V/1KHz; Secondary: 8 ohm load): 1K Ω +/-10% Primary
 - Open impedance (1KHz 1V, Secondary: no load): 4K Ω min.
 - Resistance (at 24 Deg C): 70 Ω +/- 20% Primary, 0.62 Ω +/- 20% Secondary
- 10k Ω Potentiometer on/off (SPDT) switch
- (2x) 10k Ω Potentiometers
- Speaker Amplifier (3.7W / Class-D / MAX 98306)
- 3.5mm TRS audio jack
- Sabrent USB Soundcard (audio interface w/ stereo input and output)
 - “Sabrent USB External Stereo Sound Adapter for Windows and Mac”
 - 48kHz / 24-bit (sampling rate / bit-depth)
- Raspberry Pi 3
- 5V 2.1A 20,000 mah lithium-ion polymer (lipo) battery supply (for Rasp. Pi)
- 9V battery (for LM386 circuit)

Appendix C: Select Laser Cut and Engraving Designs for Sonic Mirror Hardware Instrument (Implementation 1)

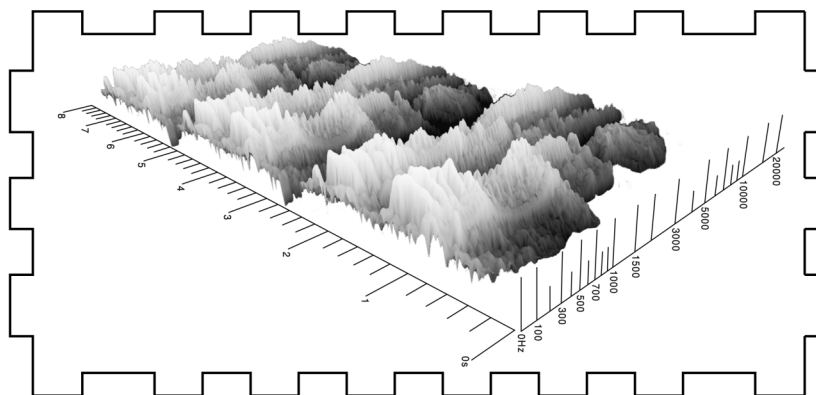
Half-scale (units = inches):



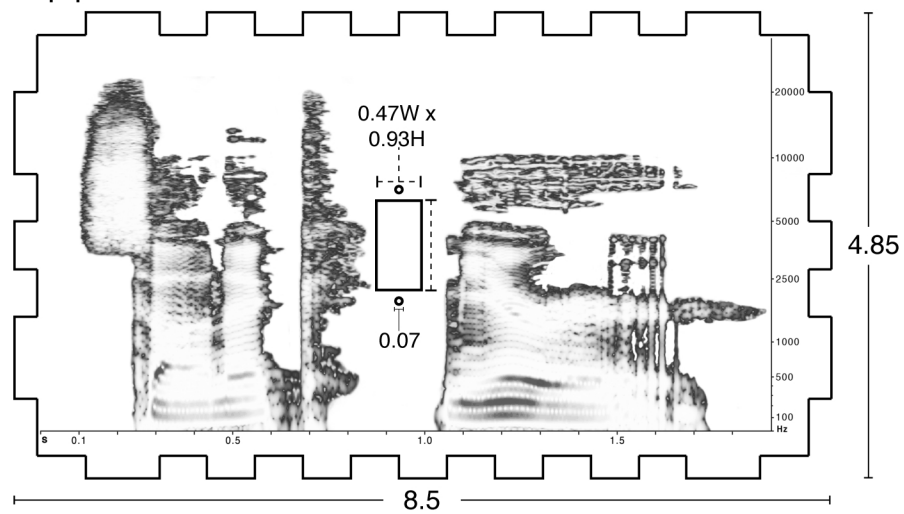
Left panel:



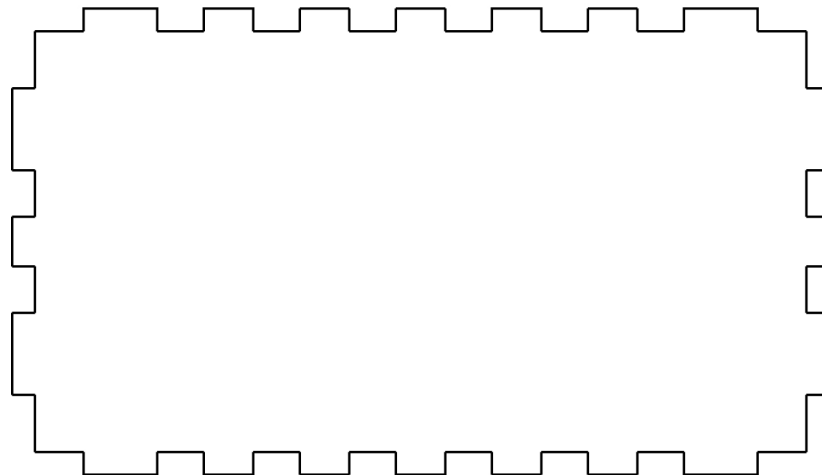
Right panel:



Top panel:



Bottom panel:



Supplemental Files

- *Tooby_DCS-RAC-Experiment_WormBot_Config1-unbroken.wav*
 - Audio file corresponding to spectrogram of Figure 39 (p.89)

- *Tooby_DCS-RAC-Experiment_WormBot_Config2-broken.wav*
 - Audio file corresponding to spectrogram of Figure 40 (p.89)

- *Tooby_DCS-RAC-Experiment_Quadruped.wav*
 - Audio file corresponding to spectrogram of Figure 41 (p.90)

- *Tooby_ultraSHIFT-SonicMirror-Implementation-1_sample.wav*
 - Audio sample of the Sonic Mirror hardware instrument installation at the *ultraSHIFT* DANM MFA show at UCSC, April 2017. The recording features the sounds of five active instruments.

- *Tooby_ultraSHIFT-SonicMirror-Implementation-2_sample.wav*
 - Audio sample from the sixth cycle of the RAC process with pitch-invariant time-stretching in a room at *ultraSHIFT*. The initial sound source was a melodica.

- *Tooby_ultraSHIFT-SonicMirror-Implementation-3_sample.wav*
 - Sample from the fifth cycle of the RAC process with pitch-invariant time-stretching in the Danser Lab's quadrupedal DCS robot at *ultraSHIFT*. The initial sound source was five seconds of sustained white noise.

References

- Basanta, A. (2015a). *Bio / CV*. Retrieved from <http://adambasanta.com/bio-cv/>
- Basanta, A. (Producer). (2015b). *A Room Listening to Itself*. [Streaming video]. Retrieved from <https://vimeo.com/135961039>
- Cramer, N., Cellucci, D., Cheung, K.C., Swei, S., & Teodorescu, M. (2017). Design and Testing of FERVOR: Flexible and Reconfigurable Voxel-based Robot. Submitted for publication.
- Cheung, K.C. (2012). *Digital Cellular Solids: reconfigurable composite materials* (Doctoral dissertation). Retrieved from <http://cba.mit.edu/docs/theses/12.09.Cheung.pdf>
- Circuit Basics. (2017). Build A Great Sounding Audio Amplifier from the LM386 Retrieved from <http://www.circuitbasics.com/build-a-great-sounding-audio-amplifier-with-bass-boost-from-the-lm386/>
- Cybulski, K. (2017). *krzysztofcybulski*. Retrieved from <http://krzysztofcybulski.com/>
- Cybulski, K. (2017, May 15-18). Feedboxes. *New Interfaces for Musical Expression (NIME) 2017*. Aalborg University Copenhagen, Denmark (p. 509-510).
- Dolson, M. (1985). Recent advances in musique concrète at CARL. In B. Truax (Ed.) *Proceedings of the 1985 International Computer Music Conference* (pp. 55-60). San Francisco: International Computer Music Association.
- Dunn, D. (1988). Wilderness as Reentrant Form: Thoughts on the Future of Electronic Art and Nature. *Leonardo*, 21(4), 377-382. doi:10.2307/1578700. Retrieved from <http://www.jstor.org/stable/1578700>
- Dunn, D. (1996). Entrainments 1 and 2. *On Music, Language and Environment* [CD]. St. Paul, MN: Innova.
- Dunn, D. (2006). *The Sound of Light in Trees* [CD]. Santa Fe, NM: Earth Ear.
- Dunn, D., & Crutchfield, J. (2009). Entomogenic Climate Change: Insect Bioacoustics and Future Forest Ecology. *Leonardo*, 42(3), 239-244. Retrieved from <http://www.jstor.org/stable/20532652>

- Dunn, D., & Lampert, M. (1989). Environment, Consciousness, and Magic: An Interview with David Dunn. *Perspectives of New Music*, 27(1), 94-105. doi:10.2307/833258
- Dunn, D., & Van Peer, R. (1999). Music, Language and Environment. *Leonardo Music Journal*, 9, 63-67. Retrieved from <http://www.jstor.org/stable/1513478>
- Gottschalk, J. (2016). *Experimental Music Since 1970*. New York, NY: Bloomsbury Publishing Inc.
- Hofstetter, R. W., Dunn, D., McGuire, R., & Potter, K. (2014). Using Acoustic Technology to Reduce Bark Beetle Reproduction. *Pest Management Science*, 70(1), 24-27. doi: 10.1002/ps.3656.
- Jenett, B., Calisch, S., Cellucci, D., Cramer, N., Gershenfeld, N., Swei, S., Cheung, K.C. (2017). Digital Morphing Wing: Active Wing Shaping Concept Using Composite Lattice-Based Cellular Structures. *Soft Robotics*, 4(1), 33-48. doi: 10.1089/soro.2016.0032.
- Jenett, B., Cellucci, D., Gregg, C., Cheung, K. (2016). Meso-scale Digital Materials: Modular, Reconfigurable, Lattice-based Structures. *Proceedings of the 2016 Manufacturing Science and Engineering Conference*, MSEC2016-8767.
- Kahn, D. (2013). *Earth Sound Earth Signal: Energies and Earth Magnitude in the Arts*. Berkeley and Los Angeles, CA: University of California Press.
- Krause, B. (1987). Bioacoustics, Habitat Ambience in Ecological Balance. *Whole Earth Review*, 57(Winter), 15-19.
- Krause, B. (1999). The Niche Hypothesis: How Animals Taught Us to Dance and Sing. Retrieved from http://xenopraxis.net/readings/krause_nichehypothesis.pdf
- Krause, B. (2002). The Loss of Natural Soundscapes. *Earth Island Journal*, 17(1), 27-29.
- Krause, B. (2008). Anatomy of the soundscape: Evolving perspectives. *Journal of the Audio Engineering Society*, 56(1-2), 73-80.
- LAST Festival. (2017). Retrieved from <http://www.lastfestival.org/>

- Lucier, A. (1990). *I Am Sitting in a Room* [CD]. New York, NY: Lovely Music, Ltd.
- Lucier, A. (1998). Origins of a Form: Acoustical Exploration, Science and Incessancy. *Leonardo Music Journal*, 8, 5-11. doi:10.2307/1513391
- Lucier, A. (2017). *Biography*. Retrieved from <http://alucier.web.wesleyan.edu/bio.html>
- Lucier, A. & Simon, D. (2012). I Am Sitting in a Room. *Chambers: Scores by Alvin Lucier* (pp. 29-30). Middletown: Wesleyan University Press.
- Lovely Music Ltd. (2017). *I Am Sitting in a Room*. Retrieved from <http://www.lovely.com/titles/cd1013.html>
- Magenta. (2017). Retrieved from <https://magenta.tensorflow.org/>
- Mumma, G. (2017). *Biography*. Retrieved from <http://brainwashed.com/mumma/bio.htm>
- Mumma, G. (2015a). Editor's Introduction. In M. Fillion (Ed.), *Cybersonic Arts: Adventures in American New Music* (pp. 39-42). Chicago, IL: University of Illinois Press. Retrieved from <http://www.jstor.org/stable/10.5406/j.ctt17w8g01.15>
- Mumma, G. (2015b). A Brief Introduction to the Sound-Modifier Console and Sun(flower) Burst: (1972). In M. Fillion (Ed.), *Cybersonic Arts: Adventures in American New Music* (pp. 65-72). Chicago, IL: University of Illinois Press. Retrieved from <http://www.jstor.org/stable/10.5406/j.ctt17w8g01.20>
- Nikiforuk, A. (2011). The Song of the Beetle. *Empire of the Beetle* (pp. 144-161). Vancouver: Greystone Books.
- Oppenheim, A.V. & Schafer, R. (1975). *Digital Signal Processing*. Englewood Cliffs: Prentice-Hall.
- Oppenheim, A. & Willsky, A. (1983). *Signals and Systems*. Englewood Cliffs: Prentice-Hall.
- Oliveros, P. (2005). *Deep listening: A composer's sound practice*. Lincoln, NE: Deep Listening Publications
- Packer, R. (2003). The Pepsi Pavilion: Laboratory for Social Experimentation. In: J. Shaw & P. Weibel (Eds.), *Future Cinema: The Cinematic Imaginary After Film* (p. 145). Cambridge, MA: The MIT Press.

- Pijanowski, B.C., Villanueva-Rivera, L.J., Dumyahn, S.L., Farina, A., Krause, B.L., Napoletano, B.M., Gage, S.H., & Pieretti, N. (2011). Soundscape Ecology: The Science of Sound in the Landscape. *BioScience*, 61(3), 203-216. doi: 10.1525/bio.2011.61.3.6
- Rabiner, L. & Gold, B. (1975). *Theory and Applications of Digital Signal Processing*. Englewood Cliffs: Prentice-Hall.
- Roads, C., Strawn, J., Abbott C., Gordon, J., & Greenspun, P. (1996). *The computer music tutorial*. Cambridge, Mass: MIT Press.
- Root Division. (2017). *MFA Now 2017*. Retrieved from <https://www.rootdivision.org/exhibition-rd-gallery/mfa-now-2017>
- Saarinen, E., Breer, R., Myers, F., Tudor, D., Whitman, R., & Experiments in Art and Technology (Organization). (1969). *The great big mirror dome project*. [Streaming video] Experiments in Art and Technology Los Angeles records, 1969-1979. Retrieved from <https://vimeo.com/24850126>
- Schafer, R. M. (1977). *The tuning of the world*. New York: A.A. Knopf.
- Schafer, R. M. (2004). The Music of the Environment. In C. Cox & D. Warner (Eds.) *Audio culture: Readings in modern music* (pp. 29-39). New York: Continuum.
- Southworth, M. (1969). The Sonic Environment of Cities. *Environment and Behavior* 1(1), 49-70. doi: 10.1177/001391656900100104
- Tenney, J. (1995). The Eloquent Voice of Nature. In *Reflections/Reflexionen: Interviews, Scores, Writings* (pp. 12 – 19). By Alvin Lucier. Cologne: MusikTexte.
- Tooby, S. (2016a, October 30). "SCML" GitHub repository. Retrieved from <https://github.com/stooby/SCML>
- Tooby, S. (2016b). (Producer). *Sonic Mirror at the Santa Cruz MAH, 2016* [Streaming video]. Retrieved from <https://vimeo.com/172147490>

- Tooby, S. (2017a, April 22). *Sonic Mirror*. Retrieved from <http://scott-tooby.com/sonic-mirror.html>
- Tooby, S. (2017b, May 15). "sonic-mirror" GitHub repository. Retrieved from <https://github.com/stooby/sonic-mirror>
- Truax, B. (1978). *Handbook for Acoustic Ecology*. Vancouver, BC: A. R. C Publications.
- Truax, B. (1984). *Acoustic Communication*. Norwood, NJ: Ablex Publishing Corporation.
- Wallace, J. (2016, May 18). Inspired and enraptured with the Cosmos. *Santa Cruz Sentinel*. Retrieved from <http://www.santacruzsentinel.com/article/NE/20160518/FEATURES/160519711>
- Wekinator. (2017). Retrieved from <http://www.wekinator.org/>
- World Forum for Acoustic Ecology. (2017). *About the WFAE*. Retrieved from <http://wfae.net/about.html>
- Wrightson, K. (1999). An Introduction to Acoustic Ecology. *Journal of Electroacoustic Music*, 12, 10-13. Retrieved from https://web.archive.org/web/20140415001407/http://www.wfae.proscenia.net/library/articles/wrightson_intro.pdf