Expressive Re-Performance

by

Laurel S. Pardue

M.Eng, Massachusetts Institute of Technology (2002)
B.S., Massachusetts Institute of Technology (2001)
B.S., Massachusetts Institute of Technology (2001)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

Author⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯
Program in Media Arts and Sciences
August 12, 2011

Certified by⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯
Joseph A. Paradiso
Associate Professor of Media Arts and Sciences
Program in Media Arts and Sciences
Thesis Supervisor

Accepted by⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯
Mitchel Resnick
Logo Papert Professor of Learning Research
Academic Head
Program in Media Arts and Sciences

Expressive Re-Performance

by

Laurel S. Pardue

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 12, 2011, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

Many music enthusiasts abandon music studies because they are frustrated by the amount of time and effort it takes to learn to play interesting songs. There are two major components to performance: the technical requirement of correctly playing the notes, and the emotional content conveyed through expressivity. While technical details like pitch and note order are largely set, expression, which is accomplished through timing, dynamics, vibrato, and timbre, is more personal. This thesis develops expressive re-performance, which entails the simplification of technical requirements of music-making to allow a user to experience music beyond his technical level, with particular focus on expression. Expressive re-performance aims to capture the fantasy and sound of a favorite recording by using audio extraction to split the original target solo and giving expressive control over that solo to a user. The re-performance experience starts with an electronic mimic of a traditional instrument with which the user steps-through a recording. Data generated from the users actions is parsed to determine note changes and expressive intent. Pitch is innate to the recording, allowing the user to concentrate on expressive gesture. Two pre-processing systems, analysis to discover note starts and extraction, are necessary. Extraction of the solo is done through user provided mimicry of the target combined with Probabalistic Latent Component Analysis with Dirichlet Hyperparameters. Audio elongation to match the users performance is performed using time-stretch. Instrument interfaces used were Akais Electronic Wind Controller (EWI), Fender's Squier Stratocaster Guitar and Controller, and a Wii-mote. Tests of the system and concept were performed using the EWI and Wii-mote for re-performance of two songs. User response indicated that while the interaction was fun, it did not succeed at enabling significant expression. Users expressed difficulty learning to use the EWI during the short test window and had insufficient interest in the offered songs. Both problems should be possible to overcome with further test time and system development. Users expressed interest in the concept of a real instrument mimic and found the audio extractions to be sufficient. Follow-on work to address issues discovered during the testing phase is needed to further validate the concept and explore means of developing expressive re-performance as a learning tool.

Thesis Supervisor: Joseph A. Paradiso
Title: Associate Professor of Media Arts and Sciences, Program in Media Arts and Sciences

Expressive Re-Performance

by

Laurel S. Pardue

The following people served as readers for this thesis:

Thesis Reader _____

<div align="right">

Tod Machover
Professor of Media Arts and Sciences
Program in Media Arts and Sciences

</div>

Thesis Reader _____

<div align="right">

Tristan Jehan
Co-Founder & CSO
The Echo Nest, Sommerville Ma.

</div>

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Most people fall in love with music through songs. The human emotional response to music is deeply innate and it is because of the experience of listening to songs, or even one pivotal song, that many musicians embark on the journey to learn an instrument so that they can realize those songs through their own doing. It is the masterwork of a Hendrix riff, a Beethoven Sonata uplifting the spirits, or a John Coltrane phrase moving one to tears that inspire a musician's practice and dedication. Even a child forced to learn an instrument by his/her parents will cease playing if the music does not eventually move them and become rewarding.

Unfortunately, most people fall well short of reaching their musical dreams. Beginner songs rarely have significantly rewarding expressive opportunities. On any instrument and with any piece, it takes years to learn the logistics of playing the correct notes. Once logistical proficiency has grown, musicality can then develop. It is years after starting that favorite songs, and sometimes even decent sound are achievable. Additionally, much music only makes sense in the context of an ensemble. Ensemble play requires not just fellow musicians, but again, a certain level of proficiency. Most people give up before reaching this proficiency, never having experienced playing exciting music or musical expressivity. But what if there was an intermediate offering? One where the logistics could be simplified leaving the student to play beyond their technical skills and also focus on the expressive aspects of music making?

Expressive re-performance is a system designed to do just that. It aims at lowering or removing

the technical barriers to achieving aurally and emotionally satisfying music-making for aspiring musicians of differing levels of proficiency. An expressive re-performance system enables anyone to perform musical pieces otherwise too technically difficult due to their required skill set and to consider expression more deeply with minimal experience. At its core, the software system takes over responsibility for generally the hardest yet least interesting aspect of learning a song-pitch, leaving the user responsible for expressive content such as timing, vibrato, and timbre.

The fantasy of playing a song like a favorite rocker extends beyond the song to the instrument. How many times have you played air guitar to your favorite Jimi Hendrix track? In order to retain the romantic associations of playing the real instrument, the system interface uses an electronic interface mimicking the original. Just as critical, is the sound itself. Playing something pretending to be a violin that does not sound much like one impedes the fantasy. The same holds true for the accompanying band; what would Louis Armstrong be without the rest of his band? Expressive Re-performance retains the magic of the original audio by actually using it. The solo and accompaniment of an original recording are split to enable the experience to closely match the original.

The complete system comes together in two sections, the pre-processing, and the real-time performance. The target user experience starts with giving the user an electronic instrument as a controller. The instrument is used to trigger a note-by-note step through of the monophonic solo, which is taken directly from any chosen recording. Since the user controls when the next-note triggers, he/she controls timing. The audio used is the original solo from the recording and inherently comprises all the correct pitches thus freeing the user from that logistical challenge. The user also controls additional expressive language, namely volume, vibrato and timbre. Additionally, the original accompaniment can be turned on and off so the user can play along with it as he/she choses. Enabling this real-time re-performance requires two significant pre-processing steps. Firstly, the recording needs to be separated into solo and accompaniment and then the solo needs to be analyzed to correctly identify where notes are so they can be correctly and predictably triggered. The complete expressive re-performance system built for this thesis includes all these parts and three commercial controllers for interfaces: a Roland Electronic Wind Instrument (EWI) [10], a Squier by Fender Stratocaster Guitar and Controller for Rockband 3[1] [15], and an abstract interface based on a Nintendo Wii-mote [19].

---

[1]This is the correct offical name.

This basic system can be elaborated at many points to accommodate varying experience levels and styles. For the moderately interested passing novice, a simple button trigger with easy large scale expression will generate the most rewarding experience. A committed novice might want more refinement and repeatability in the interaction. An expert would likely want controls that most fully compare to the real instrument. This same simplification is true for notes. This can mean a fast passage consisting of multiple notes only requires one trigger, or reflecting stylistic differences. Further, while classical music firmly follows a score, in jazz, much of expression is in improvisational flourishes. An expert might find it useful to simplify a difficult passage enabling practice to focus on technical or expressive issues independently of the notes. Catering to these different interactions lays the basis for the possibilities for expressive re-performance as not just an motivational experience, but an educational one. The amount of control given to the player can be increased to reflect increasing proficiency and through it, the learning of both the genuine notes but also original instrument. Simplification and an intermediate experience of a work might help reduce frustration and encourage a student not to quit before the end goal. Separating not just the solo, but multiple parts could also provide insight into good ensemble playing.

At the core of re-performance is the goal to provide an interaction that is sufficiently understandable, master-able, and interesting. Expressive re-performance is aimed to enable more immediate gratification of musical achievement yet also provide enough complexity to warrant further practice. Through practice the user is hoped to gain better understanding of the song and the instrument. Success is defined not only by whether it is possible to enable the fantasy of playing a certain song, but also to stimulate the user to realize the decisions that make a good performance or a bad one. These are decisions most people never get the chance to make, and they are what define what is successful musicality. For a professional musician, music making is decision making. Not only that, understanding what makes a song creates a better listener as well as performer.

1.1 What is Expression?

With the exception of free improvisation, which by definition, is not re-performance, music as traditionally defined, ranging from classical to folk, to pop, can largely be divided into two performance

factors, the predetermined notes and the expressive choice. The notes, describing specific pitch and relative duration are essentially the score, written in advance of the performance by a composer which the musician executes. The score remains the same for each performance regardless of the performer [48]. In classical music, it is followed very tightly. In jazz it may be a theme and set of harmonic guidelines. While the score may suggest expression, both implied through harmony and melody, and literally through written dynamics, tempo markings, and expressive markings like *dolce* or *furioso*, the meaning and the execution of those markings is up to the performer or interpreter. A performance that is technically flawless, executing all notes exactly as written, will almost inevitably be a boring performance. Alternatively, an expressively rich performance with deviations from the score, including some which may be criticized as errors, is often much more musically satisfying.

The effect of musical performance arises through expression: interpretation of an unwritten musical shape and phrase conveyed in part through intentionally imprecise execution of the score and manipulation of timbre. Expression is manifested through the dynamic, both overall, and in relation to neighboring notes, tempo and tempo variations including particularly subtle shortening and lengthening of individual notes for specific effect [41]. Similarly, slight changes in pitch, such as a *glissando* or the use of expressive intonation, slightly sharpening of the scale's seventh when leading to the root, can be used to affect different mood. [27]

Timbre is often a more abstract expressive concept. Timbre is the sound quality of musical tone. Timbral differences between instruments are easy to understand aurally. The resonance pattern of a violin for a specific central pitch varies substantially from a piano which also varies substantially from a clarinet. Timbre is influenced by differences in the spectral envelope and the temporal envelope, for instance differences in note decay and release. The timbral range on a single instrument is far less than between instruments, but is still highly variable. Stringed instruments in particular have a wide timbral range. Consider for instance the difference between the harsh sound of *sul pont* where the bow is over the bridge to the warm distance of *sul tasto*, with the bow over the fingerboard.

An accomplished musician must master the motor logistics to execute the correct notes, be able to carry out the creative process deciding expressive intent, and also have the musical intuition and

the technical motor capability to execute the intended expression. This takes years of dedicated practice, a dominant portion of which is spent learning the motor logistics just to execute the predetermined part of performance, the correct pitch. In fact, in traditional practice, until pitch has been at least reasonably mastered, expression within a song can not be successfully conveyed between the performer and the listener.

Chapter 2

Existing Re-performance Systems and Related Work

Practical ideas for using modern technology to simplify the logistics of the predetermined part of music so that a musician can focus on expression originated in the 1960s and 1970. Max Mathews was one of the first to express the desire to disentangle the execution of expression from the execution of score. One of his first major forays into musical interfaces was the Sequential Drum, a predecessor to the more famous Radio Baton, in the late 1970s. Matthews explicitly stated one of the motivations for the work was so"Performers can concentrate their brain power—and also their motor control—on the expression they want to put in the music, and computers will presumably fill in the routine parts correctly" [29]. He even titled an early paper on the Radio Baton, "The Radio Baton and Conductor Program, or: Pitch, the Most Important and Least Expressive Part of Music", [48] accurately highlighting the impediment pitch is to expressive music making.

The Radio Baton and Conductor System were not only a first foray for Mathews, but one of the first significant forays for anyone. The late 1980s Radio Baton, sometimes called the Radio Drum, is itself an interface, but was paired with Mathew's conductor system developed along with the Sequential Drum in the 70s. The radio drum consists of batons transmitting low frequency RF to determine x-y-z position and infer motion [47]. The conductor system then takes triggers and continuous controls from the Radio Drum to direct synthesizers. The synthesizer has multiple voices

Figure 2-1: Classic image of Max Mathews with the Radio Baton.

for accompaniment and follows a score with a primary solo voice that is more directly tied to the baton's use. The batons are used to trigger beats similarly to the way a conductor beats the time for an orchestra, hence its name [48].

It is worth clarifying the difference between expressive re-performance and some of the presently popular technical re-performance systems such as Rock Band [20] and Guitar Hero [16]. The two may seem similar as they both employ the concept of simplifying performance to appeal to the novice and also aim to craft an interaction that facilitates a performance fantasy. However, they are vastly different in the role of choice. In Rock Band, Guitar Hero and similar games, the player is given simplified cues indicating a note to play and must trigger that note at the correct time. Each instrument played has its own separate audio track and missing the timing or pressing the wrong button mutes the track, and counts against the user's score. There is no creative choice or decision. The audio is all prerecorded with the player having no control over how it sounds, only whether it sounds. Although these games do provide some sense of what it is like to play music, even for beginners, they focus exclusively on the technical aspects involved in making music, not the creative. The gamer is strictly mimicking the professional. He/she is not actually "making music." Expressive and technical re-performance tackle exactly the opposite aspects of music making.

Figure 2-2: *Rock Band* is an example of technical re-performance not expressive re-performance.

An important parent of conductor systems are "one button play" or tapper systems. These are essentially score step-through systems. The "one button play" feature debuted commercially in 1985 in Casio keyboards such as the PT-1 and can now commonly be found in childrens music toys [46]. First built by Christopher Strangio in the 70s, in a tapper system, a song can be played through by a simple trigger like a button press that causes the next note in a score to be played [59]. Tapper systems do not necessarily enable any expression beyond note timing and early ones did not enable much beyond that. Mathew's system relies on direct triggers, (meaning if a beat is late or not triggered, audio stops), so that appropriate placement of triggers turns Mathew's conductor system into a complex but expressive "one button play" system.

From simple toy versions to more complex and expressive piano mimics, there are a number of tapper systems. Some of the more advanced and interesting systems include Chalabe and Zicarelli's *M* and *Jam Factory* [63] from the mid 1980s. These included a mouse based conductor and MIDI controllable tapper like functionality as part of *M* and *Jam Factory's* composing and performance tools. The tapper functionality is merely one performance aid in a much larger composition and improvisational system, but is none-the-less an early implementation of "one-button-play" in a rich environment including synthesized accompaniment.

In 1988, Stephen Malinowski built *Tapper* in 1988. The *Tapper* uses a computer or MIDI keyboard for input and originated from the desire to record correct expression in difficult passages. Playing

out of context or out of tempo usually leads to different expression than desired when put back into the correct tempo or context. It is polyphonic in that a single trigger can simultaneously play a chord consisting of multiple notes. The *Tapper* is capable of capturing timing and volume.

Harvey Friedman implemented a similarly working MIDI score based system, *Instant Pleasure* in 1992 also played using computer or MIDI keyboards. Like Malinowski's *Tapper*, it was polyphonic but introduced a separation between the melody and accompanying chords. There were four degrees of control: triggering all notes separately, using one hand to trigger the melody and another to trigger accompaniment chords, triggering melody and accompaniment chords through melodic based triggers, and one where the accompaniment plays independently straight through and the user just triggers the melody. [46].

James Plank's 1999 *MusPlay* [49] is somewhat similar to the most complex mode in Freidman's system where all notes in a chord are triggered independently. The ability to trigger notes within a chord separately enhances expressive capability but increases difficulty. Plank simplified triggering all notes in a chord by creating time windows where notes were expected. Within a chord's timing window, the triggers struck generated pitches based off the score so that the notes played are always correct but not necessarily in unison. If a note in the chord is not triggered in time, the system moves on. This means notes can be missed but are never wrong.

Finally, the most relevant "one button play" or tapper related system was developed in 1993 by Sam Wantman and Thomas Zimmerman. Wantman and Zimmerman were more interested in the simplification of playing a traditional instrument, again looking to free performers from the technical requirements and practice necessary for enjoying popular instruments. Though naturally sharing some practical similarities with existing tapper and conductor systems, the focus on traditional performance practice and not just keyboard-based instruments were significant deviations in intent.

This thesis gains its name from Wantman and Zimmerman's 1993 patent for an electronic musical re-performance and editing system [64] which shows first intent at expressive re-performance. Though much of it remained to be implemented, their patent touches on many of the central ideas for re-performance. The work that Zimmerman and Wantman accomplished was the construction of a controller roughly simulating a violin and a processing system that accomplished similar framework

to the work described in this paper though using synthesized audio played from a loaded score. A survey by Stephen Malinowski completed in 2005 of relevant works in re-performance [46] found their work still unique and, apart from Plank's direct attempt at mimicking piano performance, effectively a percussion instrument, the attempt to expressively relate to traditional instruments and practice is otherwise unexplored. There is little evidence that things have changed over the past 6 years.

Chapter 3

System Design and Architecture

Building a re-performance system reflective of an emotive instrument and musically satisfying requires breaking the end product into a number of sub-problems. First, there must be an audio source for the solo and accompaniment. There needs to be an expressive interface capable at a minimum of capturing both discrete triggers and continuous expression. In order to capture the imagination, the interface should be physically similar to the original instrument and if technique is to transfer between the mimic and the original, the closer the interface and its means for modifying tone are to the original, the better. The user supplied control input from the instrument must then be coordinated through something acting like a score to control the audio. Further mechanisms must respond to the continuous control input to effect the audio shaping expression.

For the expressive re-performance system built in this thesis, the approach to each of the aforementioned components dramatically effected design and response. A major decision was to use original recordings and perform source separation to split the solo and the accompaniment. Though this adds significant complexity, one of the aims is to enable the fantasy of playing a favorite original. This means the music should sound like the original. With some exceptions, most easily affordable synthesizers do not do a very good job mimicking non-percussive instruments. Quality sample libraries and nicer virtual studio technology (VST) and other plugins can be exceedingly expensive and still not be particularly satisfactory. Even then, finding "the right instrument" sound can require extensive auditioning and configuring. How to find the quality of a rocker's amp and particular effects
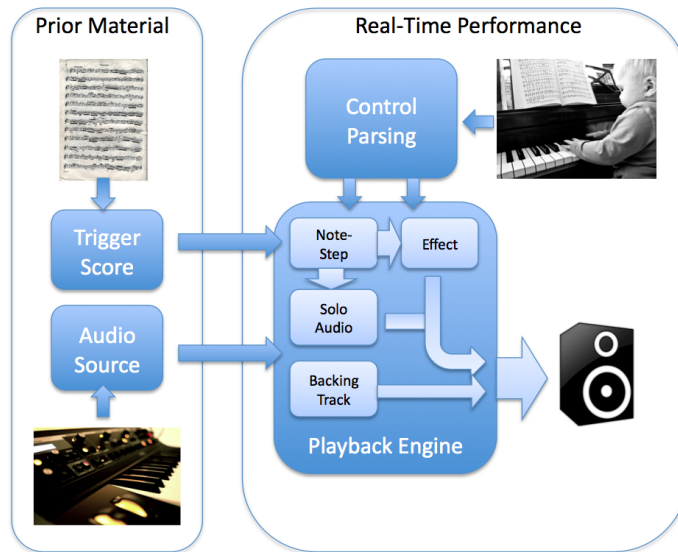
Figure 3-1: **Typical Re-performance System Architecture–** The system is split between the non real-time aspects chosen in advance, typically a musical score and synthesizers for generating the instruments sounds. The real-time interface might be a keyboard that triggers stepping through the score to produce synthesized audio in response to the user input.

pedal? Considering that recreating the full original requires not just a good solo sound, but the rest of the accompanying band too, finding and recreating the sounds using virtual and real synthesizers is cumbersome. All previous one button play systems have used synthesized audio. The expressive re-performance sounds like the original because it is directly derived from the original.

Why attempt source extraction rather than use master tracks? Master tracks are the original audio, but significantly, there is not general access to master tracks. They are often owned by record labels unwilling to release them without expensive and difficult to obtain contracts. Even if expressive re-performance had the clout of say, Rock Band, the ability to integrate any monophonic recording remains valuable as it enable the performance of music that is relatively obscure. There is no need to rely on an available library. Master tracks would actually be preferable, and in fact can be relatively seamlessly used in the system as built, but they were not treated as good general solution.

Having decided to obtain the audio through source separation, the next question was what to use as a score. Virtually all "one button play" and conductor systems have relied on a formal as-written score. The *Personal Orchestra* [28] originally implemented for an installation at Vienna's Haus der Musik is the only conductor system to manipulate recorded instrument tracks instead of using

28

synthesizers, but it focuses on conducting tempo, rather than specific note triggers so has different scoring requirements. In the expressive re-performance case there was really only one option, to find event triggers through automated analysis. In order to create a reasonable user experience, a selected note trigger must correspond with a starting transient for a note in the audio. If a note trigger is placed too early, the last note bleeds, as if there is a grace note. If too late, the quality of the attack is undermined and the previous note has a tail that does not match. Any pre-existing score would not be directly useful to accurately place note triggers against a recording as it is common knowledge that a performer will not exactly follow the score as written. Additionally, computer readable scores are not always easy to find.

Next is the interface. In order to create the feel of playing a real instrument and enable learning and existing skills to transfer between the original and the re-performance experience, it is best to use an interface as close to the original as possible. The most ideal option would be to take the original instrument, outfit it with sensors, remove its capability to sound, and use the collected information to understand the intended gesture. This would approach would as require customizing the interface with additional inputs to interact easily with rest of the system and pass along system instructions. Not surprisingly, augmenting an original instrument is fairly complicated. For instance, while there are good systems to capture aspects of violin performance technique, no one has yet found a practical means to capture the full range of how a bow is used. And then there is the question of damping the played sound. Building an interface from scratch is likely more difficult.

It was decided that building a custom interface was beyond the scope of this thesis. Capturing performance technique in some instruments could easily be a thesis in itself. Instead, commercially available interfaces for a wind instrument and guitar would be used along with a Nintendo Wii-mote. The Wii-mote is clearly not a traditiona instrument but is included as it is fairly simple, affordable, has expressive potential and is easy to use.

Having selected an interface, the commands from it, both discrete and continuous, need to be parsed and understood in order to correctly reflect the intent of the user. Once the commands are parsed, the audio needs to be altered to follow them. Since original audio is being used rather than synthesizers, extending note-length beyond the original recording is done through time-stretch. The remaining interactions are hand mapped. MAX/MSP [2] was used for handling and routing MIDI and OSC

29

messages to impact audio. Effects were supplied by Ableton Live [9] through Max for Live. These programs were chosen based on familiarity, documentation, and reliable performance.
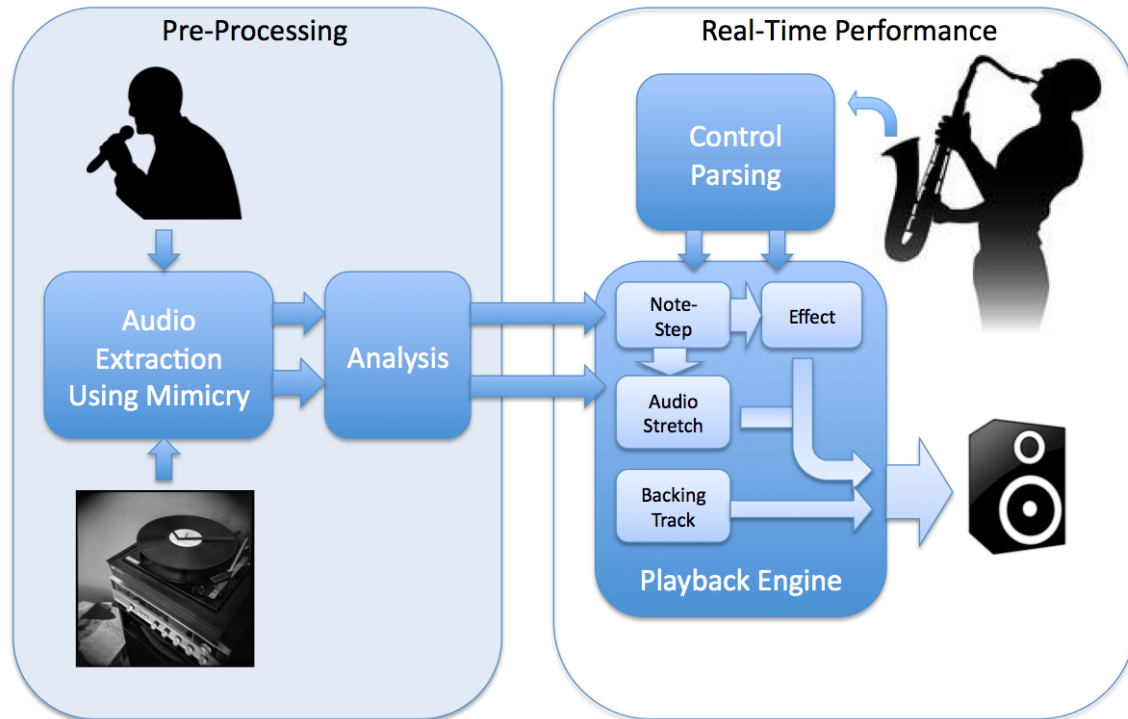
Figure 3-2: **Expressive Re-Performance System Architecture–** The system is split between the pre-proccesing sub-systems of audio extraction and audio analysis, and real-time performance which takes the pre-processing results and maps the user expression provided through an instrument like controller onto the audio.

Accompaniment is treated as a separate audio source and runs largely independent of the primary track. For true interactive performance, the accompaniment should loosely follow the solo. While there are a number of existing systems from Barry Vercoe and Miller Puckette's early *Synthetic Performer* [61] to today's state of the art, IRCAM's Antescofo by Arshia Cont [32], an anticipatory score following system, the are generally all score based. Again, the decision to use pre-recorded audio means that there is no score, and that such a system would need to rely on beat tracking. Only B-Keeper, [53] a tool by Andrew Robertson for timing Ableton Live off live drums presently accomplishes something similar to real-time accompaniment using recorded audio. Again, while creating a new non-score based auto accompaniment system would be preferable and rewarding, it is a significant undertaking and was not accomplished within the scope of this thesis. Instead, the

30

simplicity of a non-tracking accompaniment was settled on. It still enables significant fantasy.

The end result is the expressive re-performance system divided into two sections. The first is the pre-processing to separate audio and find note triggers. Then there is the real-time half which includes the interface, instruction parsing, and then time-stretch and mapping execution to execute the user instructions to create the end audio product.

3.1   Pre-Processing Systems

3.1.1   Audio Extraction & Source Separation

A major decision for the re-performance system was to use audio derived from source recordings rather than synthesized audio. This is a non-trivial challenge. Until recently it was even generally considered an intractable problem. An audio recording usually ends up in stereo with two waveforms, a left and a right channel. Each waveform is the sum of all the instruments mixed into that channel. Often instruments will share frequency space making it hard to distinguish the contributions of one instrument from another. Even if an instrument's primary range is low, the instrument has characteristic frequency response well above the fundamental note frequency. The result is everything gets mixed together. The human ear might be able to distinguish instrument lines, but computers still have a hard time with it [50].

Prior Approaches to Extraction and Source Separation

An obvious attempt to separate a particular source instrument might start by trying to characterize the spectral response of an instrument and look for that. This ends up being a poor solution as within an instrument type, different individual instruments can have substantially different characteristics and even on a single instrument, there can be significant range depending on how it is played. Not only that, different recording environments and mixing and mastering can change the characteristics further. There is no guarantee that one saxophone recording's characteristics will sufficiently

31

match another one's for mathematical identification and extraction. Instrument identification in a polyphonic environment is a challenge in its own right without adding in the goal of extraction [33].

The remaining techniques, while improving, have also traditionally suffered poor performance. Source separation through techniques like phase vocoding [44], Independent Component Analysis (ICA) [42] and Non-Negative Matrix Factorization (NMF) [55] have been slowly reaching an accuracy level to providing meaningful results. The best progress has been in proprietary software with Celemony's state-of-the-art Melodyne [11]. Affordable or non-proprietary offerings remain not readily available, but even if they were, the problem of classifying which note belongs to which source throughout a song remains. This is true even for Melodyne. The techniques mentioned are in fact called "blind" source separation since they do not include knowledge of the audio content and just seek to separate the elements within the content.

The main technique presently used to extract a single source from a recording is center channel isolation. Center channel isolation is a channel based solution that does not actually involve intelligent analysis of the audio. Instead it takes advantage of a typical recording practice used in stereo recordings where the lead vocal track is recorded in mono and then mixed in to a recording separately from the backing. It is placed centrally in the mix using the same mono waveform split between the left and right channels. Since the same waveform makes a contribution in both channels, it is possible to largely remove it by inverting the amplitude of one channel and adding it with the other channel. In the case where the lead is asymmetrically split, extraction requires figuring out the mixing weights and reversing similar to above [23]. Once the lead is out, further un-mixing can be used to compute the lead as a solo. Virtual Dub has one of the better options for this technique as well as a nice discussion of the process [22]. However trick has the obvious major problem that the target for extraction must be split and unmodified. In instrumental music this is rarely the case.

Extraction through PLCA and Dirichlet Hyper-Parameters

A new technique presented by Paris Smaragdis and Gautham Mysore in 2009 presents a promising level of success [56]. They suggest using mimicry as a means of labeling the target audio line and then using Probabilistic Latent Component Analysis (PLCA) with Dirichlet Hyper-parameters for

the extraction. Ben-Shalom [24] and Raphael [52] have attempted to use a score to make source separation and extraction less "blind" through time-varying filters and classification respectively. However problems with identifying precise time-frequency components leaves the results non-optimal. Along with the some of the general performance strengths found using PLCA, the use of a direct mimic rather than a score should improve results as minor temporal variance in a performance can be correctly labeled.

Smaragdis has previously proposed PLCA as a means to decompose audio into its separate elements, for example, the notes a piano plays in a simple piece. PLCA successfully identifies repeated patterns in a data set, such as what a single note from an instrument might be. As described in [57], PLCA is an Expectation-Maximization algorithm related to NMF [55] techniques. Briefly, it works by estimating a latent component z_i along with probability of its occurrence. In the case of audio, a latent component will have two dimensions, or two marginal variables, time, $P(t|z_i)$ with the set represented in matrix form by H and frequency, $P(f|z_i)$ with the set represented in matrix form by W. Time represents the characteristic behavior over time of the a component while frequency describes the related set of frequencies that resonate when a component i.e. note occurs. The combined probabilities of a component $P(z_i)$ is represented in matrix form by Z. The below example is the decomposition of the first 11 seconds of the clarinet part from a clarinet arrangement of "Summer Time" by George Gershwin.

The extraction process begins by recording a mimic closely following the target for extraction. It does not need to be an exact spectral mimic though not surprisingly, the closer the better. But for instance, voice works sufficiently for most cases. Most examples used were based on extractions using voice, violin, or a MIDI synth. Then the mimic is used to estimate the related probabilities Z of each of the M components within the target audio, the frequency power spectrum W and the temporal power spectrum H similar to the example provided in Figure 3-4.

Now the actual mixed recorded audio is used. The goal is to use the estimate from the mimic to split the M components meant to describe the target from N remaining backing components. Dirichlet Hyper-parameters are used to impose the estimated M components from the mimic onto the original recording. The components derived from the mimic are a starting estimate for finding the component description for the extraction target lead. The weight of this initial estimate decreases

as the algorithm iterates produces *N* components that can be used to resynthesize the backing and *M* components relating to the genuine audio rather than the mimic, that can be used to resynthesize the target.

Figure 3-3: Score for opening solo segment of Gershwin's "Summer Time" as analyzed below. The actual segment analyzed includes a further *D* at the end.
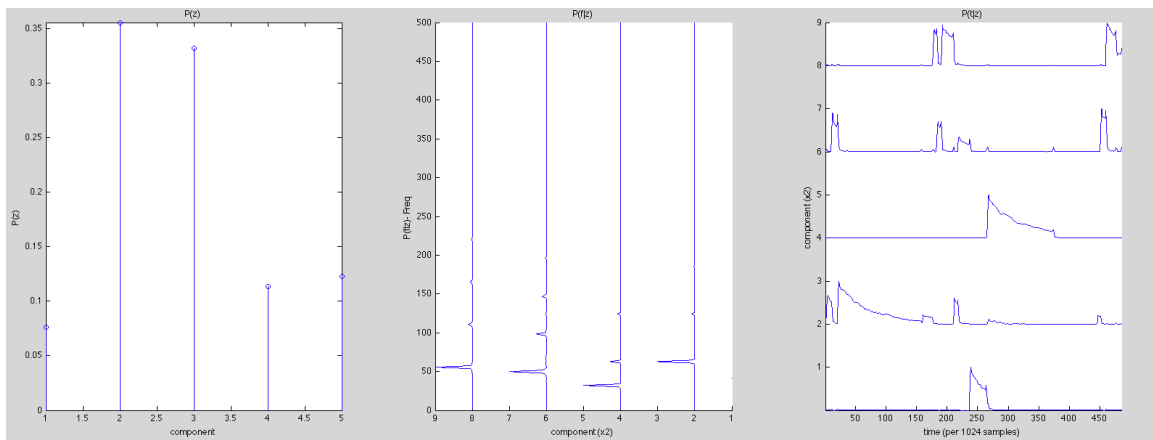
Figure 3-4: Estimated latent components from opening of lead in George Gershwin's"Summer Time". As seen in above in Figure 3-3, the section analyzed is composed of 14 note events and 5 distinct pitches. The graph on the left represents the overall probability of a particular component. The middle graph is the frequency make-up of a component, and the right graph depicts the likelihood of a component at a given time. The musical pattern can be easily understood looking at the right-most graph.

Regaining the full spectral energy by using the posterior distribution finally ends in a set $P(z \cup Z_i | f, t)$ which represents a spectral mask for the extracted lead while the remaining set provides a spectral mask of the backing. Modulating the mask from the *M* components with the original spectral distribution will yield the extracted target and modulating with the mask from the *N* components yields the backing. For more complete details, please refer to [56], [57] which presents a shift-invariant version of PLCA but is still relevant, and the extraction code included in Appendix D.

Parameters for Optimization

Within the main algorithm there are also a variety of parameters that impact the quality of the extraction. The most important is the number of components used to describe the audio, especially the mimic. Other performance impacting arguments include the number of algorithm iterations, the starting priors, the segment length, and the use of sparsity in all dimensions, W, H, and Z, in an effort to reduce undesirable minor spectral components.

The most important variable for a good extraction is finding the appropriate number of components N and M that describe the lead and the backing. Too few components and the extraction ends up an ill-defined blur that fails to include all the notes. The audio can not be sufficiently described so only the parts with the strongest spectral power make it into the extraction. In essence, a discovered component will be the average of the repeated spectral characteristics it describes. Too many components and the audio is over-described. This results in components that have minor spectral importance but are too heavily weighted, providing the opportunity for the backing to get dragged into the extraction. Under-description and over-description must also be avoided when finding the backing. If N is too small, the backing can not be sufficiently described and the re-synthesis of it will be incomplete.

Having a score, it might be easy to count how many distinct pitches occur in an extraction segment and from that, how many components N and M should exist. However, there is no score and in practice, it is not so straight-forward. For instance, one of the test extraction samples was"One Note Samba" which features extended segments with only one pitch. In theory, the existance of only one pitch would suggest that the solo could be extracted with only one component for M. In practice, this sounds very poor and using three components will sound much better.

The other major variables are segment length, sparsity, and algorithm iteration. Shorter segments are easier to describe and computationally simpler. Longer segments provide more data with which to establish a component. Which ends up higher quality and compuationally more advantageous?

Sparsity works to eliminate unimportant components and reduce excess information. Although it is successful in removing undesired audio, sparsity in W, the frequency range, tends to result in

frequency bleed, while sparsity in *H* leads to distortion. Overall, sparsity leads to better extraction clarity at the expense of sound quality.

As for algorithm iteration, not surprisingly, the more algorithm iterations, the better the result as the estimate converges closer and closer to local maxima. But more iterations are computationally costly. A relevant question is what is the minimum number of iterations required for a decent result? At what point do further iterations yeild insufficient improvement to justify computation?

A last performance variable is the starting estimate. An on-going problem in EM based algorithms is how to select starting criteria that will yield the optimal solution with the fewest iterations. Even more, how is it possible to avoid local maxima to instead find the global maxima which is the best result?

Extraction as Implemented in Expressive Re-Performance

Smaragdis's work on extraction using PLCA with Dirichlet Hyper-parameters is proprietary to Adobe. Although some of his earlier work was available, the extraction algorithm had to be freshly re-implemented and will be made available open-source. Having an algorithm is also far from having an easily usable and optimally programmed algorithm. A few additions were made to improve the algorithm results. The input audio is first cleaned up spectrally and temporally by using a binary thresholded mask. Any part of the spectral magnitude that has insufficient power either in frequency or in time, is reduced to zero. A similar binary mask can be used following the extraction.

To determine reasonable performance results, a study of all the parameters for optimization was performed. While the findings leading to these guidelines will be discussed further in the Results chapter, the end implementation tends to use shorter audio segments (10-20 seconds), 40 iterations, no sparsity, and binary masking prior to analysis. The algorithm is iterated over a range of components centered around a user provided guess with results saved for user audition. A simple effective method to deal with local minima is to run the complete algorithm multiple times starting with randomized priors and look for the best results. Naturally, this is computationally costly so the algorithm was only run 3-4 times per *N, M* selection.
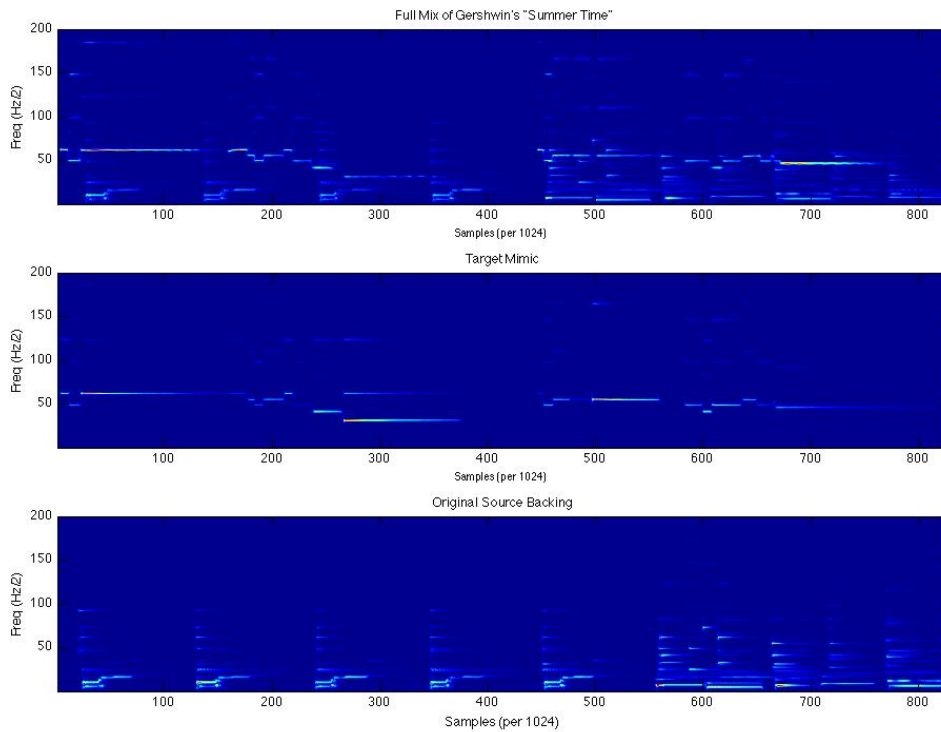
Figure 3-5: Sample Spectral Content of Opening of Gershwin's "Summer Time". Each graph shows the spectral power of at different frequencies as the song progresses. The top graph is the full mix, the second, the mimic used for extracting the solo, and the bottom is the original sourced backing audio.

Once the extraction process is complete, the various results for different component number *M* for each segment are auditioned and the best extractions are patched together to generate the whole. Depending on the care put into making the mimic, that can take five minutes to a couple hours. Splitting the mimic into shorter computational segments and choosing appropriate component estimates is fairly quick, but the extraction process itself is quite time consuming. A full suite of tests is run as per the Matlab file *runTests* found in Appendix D. A ten second segment can take up to roughly 30 minutes meaning a full 4 minute song extraction takes up to 12 hours. After this there is still auditioning and reassembly, though this can be fairly quick. Having performed a number of extractions, intuition demonstrates that the present set of tests is far more expansive than necessary so this process can definitely be sped up and plenty of room for further optimization exists. However, as will be discussed, automated tools for estimating the quality of the result are still challenging and

until a properly satisfactory shortcut can be found, the extraction process will remain lengthy. While present work points to many possibilities and has already found some useful parameter guidelines, further improving algorithm efficiency remains a substantial task worthy of independent study but is beyond what was possible in the time frame provided by this thesis. Having yielded quality results sufficient for testing the remaining sub-systems, work priorities shifted to other areas.

3.1.2   Event Analysis & Score Generation

As described in the architecture overview, the basic one-button playback technique involves playing the audio corresponding to one note and then jumping to the beginning of the next note in the recording when the user presses the trigger. This process steps through the notes. To work, the specific sample or timing location for the start of each note needs to be known and, to behave as the user expects, this determination must be very exact. A note's attack is one of its defining characteristics and must be captured, yet if the trigger is too early, it captures the tail of the previous note too. An early trigger is aurally confusing and leads to a feeling of latency. Even more confusing is when there are extra trigger markings suggesting a note changes when it does not, or missed triggers markings when notes do change. The analysis to determine trigger placement must be accurate both in catching when an event has happened and exactly where it happened.

Prior Music Analysis Tools within MAX/MSP

Fiddle˜ a classic MSP external by Miller Puckette, one of MAX/MSP's creators and a major contributor to audio analysis, has been the staple pitch analysis tool in MAX since its introduction. Puckette also released Bonk , a tool for finding percussion onsets. Fiddle˜ uses a maximum-likelihood determination for monophonic or polyphonic pitch detection while Bonk uses a bounded-Q analysis for onset detection [51] [30]. Tristan Jehan subsequently combined Fiddle˜ with additional analysis to create the Analyzer˜ external [39]. These all work in real-time on short audio time windows.

Event Analysis and Detection in Expressive Re-Performance

Analyzer~ based event detection– Being the most up-to-date and having reasonable performance, the analyzer~ external was selected for initial event analysis. The primary audio analysis is contained in the *eventDetect.maxpat* MAX patch. The extracted solo is played back in real-time and fed to the analyzer~ object. Amongst other output formats, analyzer~ provides the estimated MIDI values present paired with the estimated amplitude at any given moment. If the estimated MIDI note changes, this should indicate a note event. If the amplitude drops and then quickly rises again, this too should probably be a note event. The java script object, *extractEvent.js*, tracks the estimated MIDI note and volume to implement this basic note event extraction. It attempts to catch errors in the pitch detection by filtering out momentary changes in note classification or estimates that have low associated amplitude. A low amplitude reading often indicates error in classification or, alternatively, if the amplitude is very low, it is likely that the true event is a rest even though an estimated pitch is provided.

Re-attacks were classified based on drops in amplitude regardless of estimated pitch. A frequently reliable classifier was, if the amplitude rose consistantly by 10db in under 24ms, it was a re-attack. The threshold is actually variable. A larger db threshold may miss some re-attacks while a lower one may incorrectly insert an event. The analyzer~ external provides an attack analysis but it failed to find many events. *extractEvent.js*'s MIDI output approach proved to work more usefully, in part as it is possible to set the amplitude at which something is considered an attack or re-attack. For the specific re-performance test cases, performance using the 10db threshold, seemed a reasonable compromise.

extractEvents.js limits rapid note changes. It is expected that there will be no new notes, either through re-attack or pitch change, within 85 ms of a previous event. This should hold true based on performance practice and eliminates accidental classification as a new note settles. *extractEvent.ja* also estimates rests and when a note ends. Lastly, analyzer~ has a natural latency roughly equal to the window size of the FFT involved so that is taken into account when placing note triggers. The primary two items stored are the event's start sample along with the amplitude. A note amplitude of zero is a rest.

While the analyzer˜ and *extractEvents.js* based event determination turned out to be quite good at identifying note events both through re-attack and pitch change, there was too much variable latency for tightly determining the exact trigger point. For instance, the latency for pitch changes was different than the latency for re-attacks. The resolution for where to start was also limited by hop size. This can be reduced at the cost of computation. Still, based on judgment during test runs, the location for note start needs to be within 5ms of the true start. The analyzer˜ based event detection was insufficient as a lone source for trigger locations.

EchoNest analysis based event detection– EchoNest presently provides state of the art analysis tools through the internet [4]. Upon registering with EchoNest, it is possible to upload audio for analysis. Though it runs remotely, it is faster than real-time. EchoNest has released an MSP external en-analyzer˜ which takes the audio buffer to be analyzed along with a free user key and returns a JavaScript Object Notation (JSON) file with the resulting analysis. The MAX patch *en-eventDetect* provides a host for the en-analyzer. It selects found"segments" and saves them into a collection in the correct format. It suffers from timing drift, but once accomodated for, the advantage of the en-analyzer˜ is that the placement of note-triggers is very exact. Apart from the requirement for the internet to use it, the drawback is that it both finds triggers where there are none as well as missing denser triggers.

Detailed results of the performance of both approaches are provided in the Results chapter.

User Assisted Event Trigger Editing

Since no analysis tool was going to be completely accurate the user must become involved. The Event Trigger Editing Graphical User Interface (GUI) shown in Figure 3-6 provides an easy environment for editing the note triggers. The top of the GUI displays the waveform of whatever audio sample is loaded into the MAX/MSP buffer using the MSP waveform˜ object. Display settings are echoed in a scroll bar directly beneath. The scroll bar displays what section of the buffer is presently displayed and also enables direct selection of the viewing area. Beneath these two bars are the event or note bars. Two note bars enable the editing of two analysis files at once which can
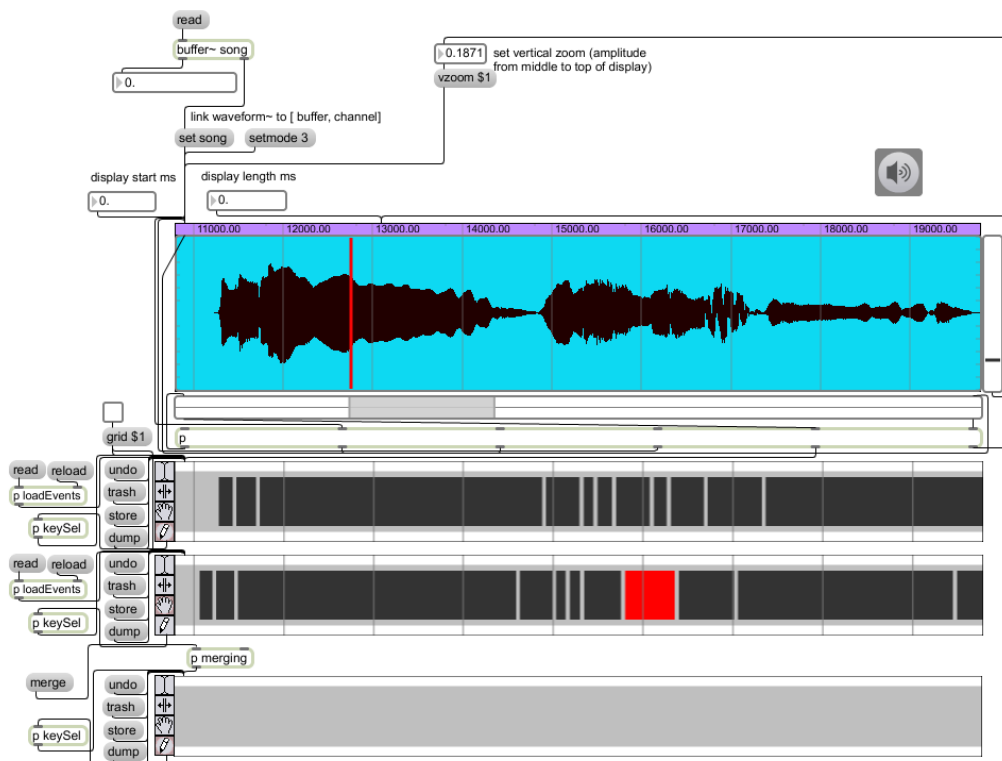
Figure 3-6: **Event Trigger Editing GUI–** The main window represents the loaded waveform. The three bars below display note events in comparison to the waveform. The top two event bars can be loaded independently and then merged for a final result displayed in the third event bar. All three event bars are fully editable using move, trim, select, and delete commands. The bar to the right of the waveform allows scaling of the waveform view.

then be merged. The merged result can then be edited and saved. This set up allows the merging of a more complete data set found using the analyzer~ based patch *eventDetect.* with the more accurate data set produced through the en-analyzer~ based patch *en-eventDetect.*

The *chooseEvent.maxpat* GUI implemented in MAX/MSP allows most standard editing features found in a MIDI editor. There were no obvious MIDI editing tools readily available for reuse in MAX. There are some desired non-standard properties so the GUI was custom designed. A trigger can be moved. A new trigger can be drawn. An event can be trimmed on either side. Multiple events can be selected and moved. An event can be deleted and so on. There is an undo feature so that any accidental changes are not problematic. Lastly the wave itself can be played back starting from the red bar in the window. This bar, and with it audio start, is moved by clicking on the waveform.

41

Along with the visual representation, the play bar is particularly useful for finding correct note placement.

The automated tools provide a great start for finding event triggers. The *chooseEvent* GUI provides a fairly complete tool for the user to step in and fix those things the automation failed to get right.

3.2 Real-Time Systems

The real-time systems are those systems that are involved in the user playing experience. This starts with the interface in-hand and includes the data parsing and processing to understand the control messages delivered by the interface. It continues through to the note-playback, time-stretching and audio effects routines, creating and altering the audio in order to execute the user's musical intent.

3.2.1 The Instrument Interfaces

One of the most important factors in user experience is the interface. Considering that an explicit goal of expressive re-performance is to simplify a real instrument, the ideal interface would play, look, and feel like the original but somehow be easier. As previously explained, the decision was made to use commercial data capable imitations of real instruments. Implementing a custom instrument or instrument sense capability is a substantial undertaking and beyond the scope of this thesis.

The instrument interfaces used in expressive re-performance must be data capable and for practical reasons, affordable. Data capable implies that information is captured about the playing, fingering, and expressive controls and sent digitally through MIDI, Open Sound Control (OSC) or some other digital format. As stringed instruments are often considered the most expressive instruments and are also very difficult to get started on, they seemed a logical choice for instrument to model on. Though there are many affordable electric violins and cellos, there are no suitable data capable instruments. Keith McMillen Instruments [18] offers some interesting sensing systems for real violins but they are prohibitively expensive for this work.

With string instruments eliminated, that left woodwind and brass instruments. Akai makes the Electronic Wind Instrument (EWI) [10] and Yamaha makes the WX5 [7]. There is also a commercial MIDI trumpet, the Morrison Digital Trumpet [12] , but it is expensive at close to $3000. The EWI most closely resembles a saxophone but can be set up to finger similarly to a clarinet or brass instrument. Internet user comments and reviews found them both quality instruments. The WX5 is somewhat closer to the traditional instrument, but the Akai EWI was rated as slightly more robust and samples data at a much higher rate than the Yamaha [8]. In the end, the EWI was chosen over the WX5 for a test controller due to data consistency.

The other instrument targeted especially considering its popularity is the guitar. Gibson has been making high end digital guitars for a few years. These remain quite expensive. Fender has been working on a new MIDI capable guitar for Rock Band which was scheduled to be available for general release towards the end of thesis work. Luckily, Harmonix were generous enough to loan one of their few pre-release Rock Band Fender Stratocasters for temporary use. Unfortunately, it had to be returned prior to final user tests, but it did make it into earlier preliminary system tests.

Lastly, a Nintendo Wii-mote was added to the mix. While clearly not an instrument mimic, it was useful as an alternate interface to test with. It provides both an opportunity to test the user response to a traditional interface vs. the response to a non-traditional interface and also an opportunity to look at system performance separate from the EWI. As will be discussed later, despite simplifying play over a real clarinet or saxophone, the EWI is still a complicated instrument with its own challenges. The Wii-mote contains a large number of both continuous and discrete sensors giving it wide expressive possibilities making it a good option as an expressive interface.

Before moving on, what about a piano? A piano was not used for two main reasons. Firstly, it is a percussion instrument with expression achieved primarily through note volume and timing. It can be well defined through its impulse and release. This is the most significant reason it has been successfully sampled and replicated electronically. This is not to claim it is not an expressive instrument worth consideration for expressive re-performance, but it has already received significant treatment by others. Apart from Zimmerman and Wantman, all previous one note play systems have focused on keyboard interfaces. Plank's previously mentioned system is a compelling pass at achieving the same goals as expressive re-performance but for a piano. The expressive re-performance system

intends to target new interactions rather than retread old ones. Beyond that, any piano oriented system is limited by the natural inability to shape the middle of a note envelope. The expressive re-performance system features many divergences and expansions beyond prior systems, but more importantly a piano fails to capture the flexibility and range of other instruments.

The other major reason this thesis does not target pianos is that they are a poor target for audio extraction. Virtual pianos based on samples are mature, easy to come by and sound good. Audio extraction for source audio is being used in part to gain access to sounds not well synthesized which is not the case for pianos. Outside of classical performance, sample based electronic pianos are widely accepted in professional circles. Additionally, piano parts are rarely monophonic. The piano is successful as an expressive instrument because of its ability to play many notes at once. This means it is usually played across wide frequency ranges with significant spectral complexity. This makes any extraction of it more likely to suffer from outside instruments bleeding in and desired note triggers difficult to find. A novice pianist may still enjoy using the expressive re-performance system for all the benefits it offers but a piano does not make the optimal test candidate.

3.2.2   Making Time Flexible

One of the most important means for impacting expression is timing. Even placing a note milliseconds early or late can have a noticeable expressive impact. As a user steps through a song, compressing time is easy, the system can easily skip playing a note's tail and jump to the next one. But time works both ways. In order to allow the user to stretch time, the audio must be modified to accommodate. Simply slowing down audio play back rates will also drop the pitch as the audio signals are correspondingly stretched out too. Avoiding this requires the introduction of a much more complex tool, a time-stretch unit.

Prior and Existing Time-Stretch Implementations

Audio time-stretch is the process of changing the speed of an audio source without changing the pitch, or transversely, changing the pitch without changing the speed. There are handful of tech-

niques for doing this. They mostly fall into three categories: phase vocoding, time domain harmonic scaling (TDHS), and spectral modeling [26].

Flanagan and Golden suggested phase vocoding as the first means for re-timing in 1967 [54]. The basic concept is to transfer the audio into the Fourier domain using a Short Time Fourier Transform (STFT). The Fourier domain enables the classification of the frequency components within the segment. The audio sequence is then re-timed retaining the same frequency content found in the STFT. For an example of how this works consider shifting all the frequency content related in the STFT twice as high as it was originally, and then replay the resulting audio at half speed. The half speed playback will drop all frequencies by 50% meaning the raised content returns to the original frequency but now the song is twice as long. A complicating factor to this approach is that, as the STFT is a digital process executed on a finite window of audio, the STFT is not sufficiently exact. Using 8 bits to represent an audio signal ends up classing multiple amplitudes as one amplitude. Similarly, frequency resolution of the STFT results in the classification of like frequencies together. The phase shift of the signal across an analysis window is used to help effectively increase resolution [25]. While there are other artifacts and error introduced in the process, the main result of the phase vocoder is a spread in the frequency domain. This is particularly distorting for items with fast transients like percussion. Computing the STFT and phase differences are also fairly math heavy and computationally costly.

TDHS and other time domain related algorithms are a lower computational cost solution. They attempt to estimate the frequencies by looking at the periodicity in a short waveform segment rather than through the Fourier transform [45] [38]. Having estimated the primary frequencies, through auto-correlation or other methods, in order to expand the audio, the algorithm fills time with the detected frequencies though overlap and add, maintaining smooth phase. In compression, it removes segments sized to a multiple of the period, again maintaing phase. TDHS fails if the target periodicity is incorrectly estimated. Monophonic signals with strong fundamentals are easily estimated but polyphonic signals are problematic. While TDHS is good at temporal performance, the lack of frequency detail leads to distortions.

A more recent approach is spectral mapping. The overall idea is similar to the previously discussed audio extraction works in terms of finding components and a set of bases that can be reused. Rather

than only consider the frequency (as is done in the phase vocoder) or time (as in TDHS) spectral mapping looks to find spectral patterns and"understand" the temporal and spectral characteristics in order to spread it appropriately. While there are a variety of implementations, for the most part, they are generally proprietary. For instance Prosoniq has one of the more advance options, Multiple Component Feature Extraction. Details refer to"artificial neural networks" but is devoid of further revealing information [6].

The typical present application approach to time-stretch is to combine the different methods. Phase vocoding has continued to improve as computers become more powerful and works well for polyphonic melodies. TDHS works well for transients like drums and remains the champion when low computational demands are most important. Then there are the various spectral algorithms, some some of which have the benefit of offering a wider range of applicability. Phase vocoding and TDHS usually have a range of 50-200%. Some of the spectral modeling techniques for instance Paul Stretch [5] can slow the audio down to essentially infinite length. This does obviously impact sound quality.

A Time-Stretch External For MAX/MSP

Clearly high quality time-stretch is a fairly complicated subject and most quality solutions are the results of in depth work and evaluation. Expressive re-timing required a time-stretch algorithm that could work in MAX/MSP, the host program, and respond to real-time time expansion controls. There appeared to be only one present time-stretch algorithm readily available to MAX/MSP, elastic˜ by Elasticmax [14]. Being inexpensive and ready-made, it was the initial easy solution. However the the audio quality turned out to be completely unsatisfactory. Playing back at the original recording rate without adding any pitch or time stretch, the audio already sounded distorted. Unsurprisingly, stretching it just sounded worse. A different solution was required. A new external had to be made to provide MAX/MSP with a decent pitch-shift and time-stretch engine.

Thankfully, unlike with extraction, there are a handful of good open source time-stretch options available. The primary four options are Rubberband [1], Dirac [13], Soundstretch [21], and Apple's Audio Unit (AU) TimePitch plug-in. Dirac and Soundstretch were eliminated for licensing

constraints and quality issues, respectively.

This left the choice between Rubberband and AU's TimePitch plug-in. Rubberband has been used in major open source applications including Ardour, a prominent open-source digital audio workstation (DAW). It is reasonably well supported and still being developed. Rubberband also has a virtually unlimited stretch range. Rubberband does have a real-time API that is claimed to work, however at development time it was difficult to find an example of Rubberband's real-time API being used in order to verify Rubberband's real-time performance. Ardour only uses Rubberband to re-time files outside playback. Other software using Rubberband plug-ins followed suit, only using the non-real-time engine.

The Audio Unit plug-in TimePitch is part of Apple's Audio Lab initiative to provide outside developers a means to create plug-ins for use with the Mac OS. There is no source code for the plug-in itself. TimePitch maintains a high sound quality and had the major advantage that AU Lab comes with every Mac so it was easy to test the real-time performance [3]. Rendering audio outside of real-time Rubberband outperformed the AU's TimePitch. While AU TimePitch also lacked examples for how to use any of the conversion plug-ins in real-time and had a fairly convoluted API, AU Lab provided an easy opportunity to verify its performance. AU's TimePitch only scales time 25%-400% time, but as it had guaranteed acceptable sound quality, it was chosen for the adaption into a MAX/MSP external.

The AUTimePitch~ MAX/MSP external primarily acts as a straight-forward fully-capable mono host for the TimePitch AU plug-in. It is composed of two files, AUConvert.c and an accompanying header file AUConcert.h. It is written in C following the standard MAX/MSP APIs and relies on the Apple AU Core Services and Audio Unit C++ libraries. Due to changes in these libraries it is only compatible with Mac OS 10.6 and later. It provides full access to all TimePitch parameters, from analysis window size to window overlap.

The AUTimePitch~ external reads from a MAX audio buffer into the AU TimePitch plug-in and then outputs the resulting audio signal. Along with the buffer name, the main inputs are an MSP signal, where to play in the buffer, the time ratio to play back at, any pitch alterations, and whether to loop audio. The output is just the audio signal and a loop sync flag that signals when requested audio
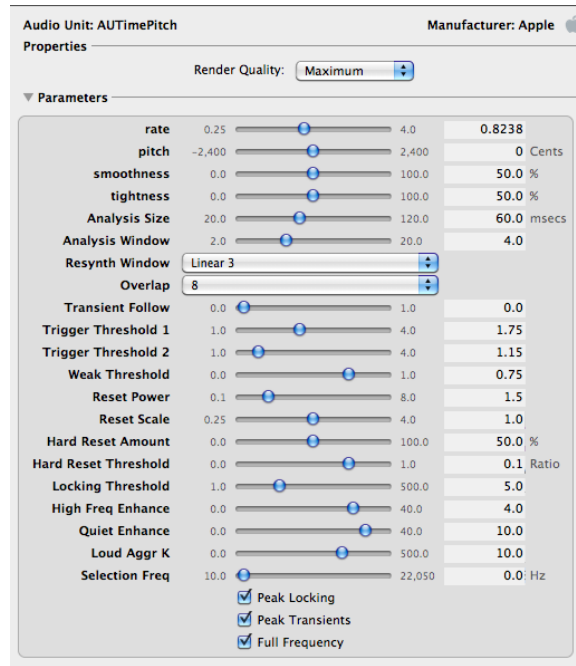
Figure 3-7: AU TimePitch plug-in hosted by Apple's AU Lab

has been completed. The loop option and the loop sync are not requirements for the Expressive Re-performance system but improve re-usability as it has been posted for general public use with the MAX/MSP community run by Cycling74.

One important feature implemented in the AUTimePitch~ external is a cross-fade. Suddenly switching between one wave and another will induce a click, particularly if the switch introduces a discontinuity in the waveform. The typical fix for this is to introduce a cross-fade between the finishing signal and the starting signal to smoothly transition between the two waveforms. AUTimePitch~ uses a custom cosine cross-fade not just to fade between loop end and start as usually found in looping software, but also between all provides cross-fade between segments to accommodate the jumpy nature of playback in expressive re-performance that is outside of normal audio playback behavior. Additional commands and details can be found in Appendix C which includes the AUTimePitch~ MAX/MSP help file and the source code itself. AUTimePitch~ can also be found on its own web-site residing at *media.mit.edu/punk~/Public/AUTimePitch/*.

Table 3.1: Playback rate per portion of note completed.

Portion of Note Played	0-.1	.1-.25	.25-.35	.35-.45	.45 - 1
Playback Rate	1	.9	.6	.4	.25

Integrating Time-Stretch into the Expressive Re-Performance system

When used in Expressive Re-performance, a note is identified by its start and end point as determined through the automated analysis. Playing the note occurs by requesting the note segment using its start and end time. Since the length of time the player would like a note to last cannot be predicted, Expressive Re-performance works to maximize note length with minimal degradation of sound. The slowest playback rate TimePitch can support is $1/4$ the initial speed. This means with the present algorithm, no note can ever last more than four times its original length. However, if the user was playing the real instrument, even playing at a much lower tempo, the attack of a note will likely stay the same. The attack is often the most complex spectral moment and hardest to reconstruct. In order to retain the character of an attack, the note initially plays back at its original speed before slowing down to the minimum supported by the TimePitch plug-in. In general, the further from the original tempo, the more distorted the audio will sound. Also, a rapid significant change in play-back rate will introduce artifacts. With these two issues in mind, the playback rate will progressively slow down in an effort to maintain a best quality to length ratio:

Using the playback rate per portion of original sample completed as in Table 3.1 enabled a note length about three times the original maintaining smooth quality audio. These ratios were chosen through informal testing. A number of other possibilities were auditioned and observationally judged before arriving at the present playback rates. The playback segment is determined in a MAX JavaScript applet, *noteReplay.js*, reading the analysis file, triggering the new note segment and then determining the appropriate playback rate at any given time and forwarding that parameter to the AUTimePitch˜ external.

3.2.3 Expressive Mappings

Mappings were all done in a MAX/MSP patch embedded in Ableton Live using MAX For Live. MAX/MSP is a graphical music programming language useful as it has a lot of tools for automatically parsing and routing MIDI along with loading and modifying audio. It has an expansive user community meaning a couple functionalities not included in MAX, such as OSC routing and rough time-stretch were available without needing custom coding. It also allows custom JavaScript applets which enable embedding normal programming practice into a patch. The graphical interface is not well designed to handle simple programming concepts like "for loops" so the JavaScript option becomes quite useful and in fact, the vast majority of data handling and manipulation was accomplished within custom JavaScript applets.

Ableton Live is an audio performance, recording, and mixing software with a wide range of instruments and effects. All of these are mappable and can be controlled from outside Live using MIDI or OSC. Max for Live is a special crossover platform for embedding MAX/MSP patches into Live. It includes automatic MIDI routing and provides means to forward MAX/MSP data to control Live parameters. Not much was actually used from Live. It ended up as just an audio host otherwise using just a handful of rather simple effects. While it was certainly useful for the initial flexibility auditioning and selecting effects, the expressive re-performance patch could easily be moved outside of Live. This would remove the necessity for a particularly expensive piece of software.

Mappings were chosen to match both the normal instrument playing style and the sound if possible. There are some obvious standard expressive relationships which were natural starting points. For instance, every instrument has its standard means of making sound and choosing pitch, be it fingers on strummed strings or breath combined with pressed buttons. Both the Fender and the EWI normally generate MIDI note ons. These note-ons were the obvious choice to trigger note advancement. Timing expression is inherently controlled through the note triggers. Other expressive interactions varied from instrument to instrument but included volume, pitch, some sort of timbral alteration, and for the EWI and Wii-mote there was the ability to add trills and improvisation.

The Fender Stratocaster

The Squier by Fender Stratocaster Guitar and Controller [15] designed for Rock Band 3 is a new affordable MIDI guitar that can be played as a normal electric guitar or can act as a controller for Rock Band 3. Electric contacts on the frets detect fingerings and a special dual-bridge pick-up recognizes strumming. It includes two dimensions of tilt sensors to detect the physical pitch at which it is held. For instance it detects if the guitar is pointing floor to ceiling, held in a normal playing position, or considering the other axis, if the guitar face is facing the floor. It also includes various game remote style buttons for game play. All of this data is MIDI friendly.

Figure 3-8: The Squier by Fender Stratocaster Guitar & Controller.

Striking a chord triggers MIDI note ons from the Fender indicating to advance a note. In the expressive re-performance system, a chord should be treated as a single event but playing a chord generates multiple note-ons. This was solved by adding a de-bounce function. De-bounce turned out to be generally useful not just for the Fender but for accidental triggers with the EWI too. What the de-bounce does is disallow input within a certain period after the a first input. In this case, multiple strings were intentionally hit, but only one response was desired, The de-bounce effectively simplified the data to one event. The de-bounce is 100ms which is roughly as fast as a musician can play [43]. There should be no intentional note strikes faster than this and anything within that window is treated as part of the first strike.

Like the mapping for note trigger, volume should also follow naturally from instrument play. With the Fender, though it was supposed to capture the volume through the strike force, as a would happen with a real guitar, things did not seem to work as expected. Plucking a single note would rarely register. Strumming a chord without a pick often resulted in missed notes too. The only

51

time it behaved reliably was with aggressive chord play using a pick which always resulted in high volume.

Figure 3-9: Roland FC-300 MIDI foot controller

A Roland FC-300 MIDI foot controller with pedals was added to compensate for the lack of reliable volume data and also to expand the expressive interface in a way typical for guitars. One of the pedals controlled volume while a second pedal was used for expression. The extra expression pedal was directly linked through Live to delay. Delay was chosen as a typical favored guitar effect, but Live has a wide variety of the standard possible effects from reverb to distortion. As it is directly mapped, bypassing MAX, switching effects is very easy.

Pitch expression on the Fender was determined by the X-Y angle or pitch the instrument was held at. The continuous data rate from the Fender was slow enough that direct mapping pitch shift produced a mildly discontinuous and jumpy effect. Adding data smoothing in MAX solved that problem. Due to the Fender being available only during earlier stages of the work, no improvisation was ever added to the mappings. One convenient attribute of the Fender is that it is designed to also be a game controller. It has buttons for basic controls, convenient for some of the non-note non-musical commands such as *reset*.

The EWI

The most complex and expensive interface of the group was the EWI. Again, the Akai EWI, based on original work by noted musical interface designer and EVI inventor Nyle Steiner [58], is modeled most closely on a saxophone. It uses capacitive sensing for key press and blowing through the

instrument is required to both play a note and control volume. The other requirement to play the instrument is that one thumb must be on either of the two rear ground plates. This is done for the benefit of the capacitive sensing. The EWI is definitely not the same as the real instrument and comes with a handful of differences.

Figure 3-10: The Akai EWI 4000S

Per the manufacturer, vibrato is controlled through biting the mouthpiece. This is reminiscent of, though not the same as, how pitch is altered on a reed instrument and ends up a little odd. Beyond the mouthpiece there are also pitch bend plates on the back, at either side of the right thumb ground plate. One of the more difficult to master features is the *octave rollers* where the left thumb sits. Moving up or down between rollers changes the octave the EWI plays at. On one side of the octave rollers is a ground strip, and on the other is a portamento strip. Touching and then increasing pressure on the portamento impacts the rate at which the controller moves from one note to another in a continuous manner. The last two features on the EWI are the *hold* and *octave* buttons. If using the EWI's internal sound engine, these will hold a note or add an octave to what is being played. All of these are reconfigurable to send MIDI, which can then be reinterpreted.

The major mappings for the EWI were straight-forward. Along with note advancement, breath continued to control volume and the control data from the bite continued to control vibrato. Vibrato is implemented by changing the pitch parameter into the time-stretch unit, raising and lowering the pitch performance slightly and never more than 64 cents, or just more than half a semi-tone. This is well within reason for realistic vibrato. The pitch bend plates are ignored. The control data from

53
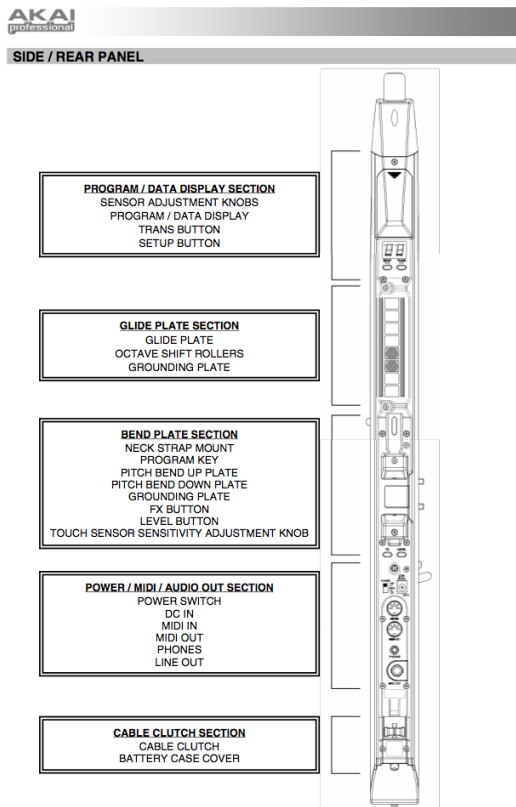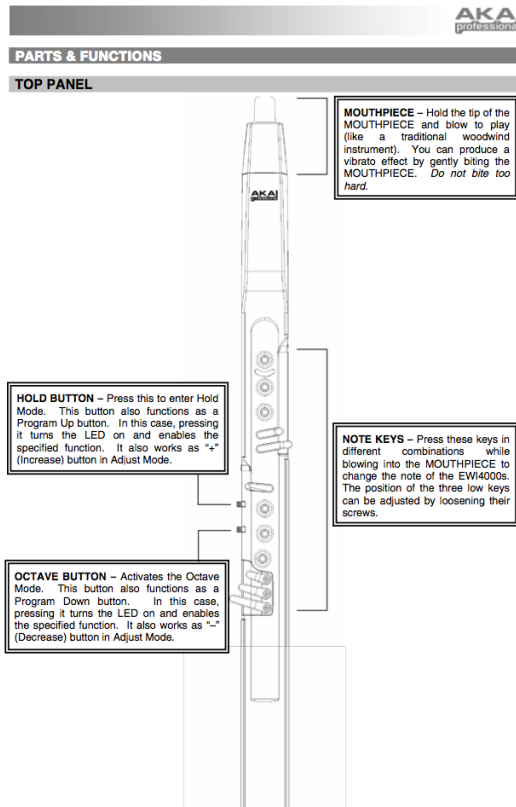
(a) image 1   (b) image 2

Figure 3-11: EWI interface and controls as taken from the reference manual

the portamento strip was used to control a different timbral effect chosen to reflect a more normal effect a musician can inflect. A combination of Ableton effects aimed at thinning or warming the sound was chosen. Originally, the effects consisted of altering the wet/dry output of a recording cabinet mimic, and reducing the high and low frequencies through a three band EQ. These were effective in achieving the desired timbral change when auditioned. In performance, it was hard to hear the impact of these when the solo was played with the accompaniment. A light set of resonators was added, which makes the change a bit more dramatic and noticeable at the expense of making the instrument sound a bit more synthetic. Pressing the bar raises the amount of sound from the resonator, cabinet mimic, and increases the level of band-passes on the audio.

The hold button and the octave button were put to good use too. They were changed to send

MIDI control data. The hold is used to repeat a note. While held, any new note event re-triggers the last event played. This is useful if a user playing with the accompaniment gets ahead or the user just wants to play around on a longer note. The octave button was re-appropriated to enable improvisation. The improvisation button is latched and when on, the pitch response more-closely follows the real fingered values. All pitch change intervals are made relative to the original fingered key started from so that playing up a third from the original button, raises the audio a third. This makes for easy addition of trills and quite a bit of creative license. When improvising, breath becomes the only note trigger as keys now affecting pitch can no longer can be used to advance notes. This allows improvisation to continue while stepping through the notes, though it does impact articulation.

Lastly, removing all fingers from the buttons was used to reset the note event tracking to the beginning of the song. This is clearly not an expressive interaction, but is highly useful from a simple usability standpoint.

It is definitely worth stating that the EWI, in its quest to satisfy many styles of professional play, is an easy instrument to make a mistake with. The biggest most confusing part of the interface was the octave rollers. The use and even existence of these take a while to get used to. Trials found plenty of accidental note changes due to accidentally moving the left thumb especially as one of the ground plates is right next to it, so the thumb always ends up on the octave rollers. The propensity for the rollers to cause error and confusion was removed by JavaScript filter code written so that with the exception of improvisation, changes in octave are ignored and treated as no change at all.

Accidental button press was also an issue. The EWI is highly sensitive and it turns out it is quite easy to accidentally change more notes than intended. The sensitivity can be set on the EWI itself, and was indeed modified. Combined with the de-bounce that was built for the Fender, this definitely improved playability, however it remains only through practice that a user begins to learn the sensitivity and reduce accidental notes. Since the interface is intended to be accessible quickly, two note trigger settings were created. The first note setting essentially defaults to the EWI's normal note triggering minus the octave input from the confusing octave bar. Putting down a finger changes the note and causes an event trigger as does removing the finger. Any button touched is just as valid to trigger note change. The second setting is more of a button mode. This time, only two notes are

treated as valid notes to trigger event changes from. Also, only the note press is considered. It is more like pressing one of two buttons. The button mode did have to be amended after user tests. Holding the EWI, many users would accidentally touch one of the accidental keys unintentionally, changing the note value so it was no longer the "button" note.

The Wii-mote

The Wii-mote is of course, not modeled on any real instrument. It is a fairly well endowed controller containing accelerometers in three directions, infrared (IR) tracking, and gyros. Being a remote, it also includes a number of thoughtfully placed buttons and communicates to a Nintendo base station using Bluetooth. Communicating through Bluetooth makes it readily hack-able, and it has become very popular as an expressive interface.
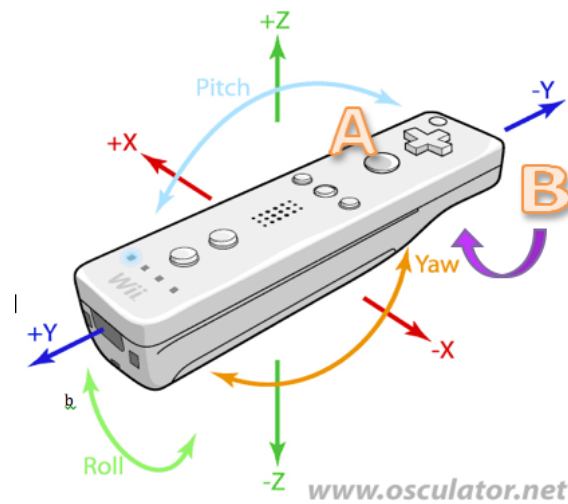
Figure 3-12: Wii-mote sense capabilities and interpretation. *From Osculator.net*

The top"A" button made for a useful and natural event trigger. The"B" button underneath was also convenient for implementing a repeat button just as done with the EWI. Holding"B" and pressing"A" repeats a note. The direction buttons on the Wii-mote were used for improvisation. The left button raises the pitch 200 cents, equivalent to a whole step. The up button raises pitch 100 cents to a half step, while the down button drops the pitch a half step and the right button drops the pitch a whole step. This is much more limited than the EWI's improvisational range, but the Wii has far

fewer buttons and it is still sufficient for any trill. The "home" button triggers the solo track to reset and go back to the beginning.

The Wii-mote attempts to track acceleration in the x, y, and z directions. As depicted in 3-12, it also tracks pitch (whether it is pointed up or down lengthwise), roll (whether it is point up or down widthwise), and yaw (rotation in the x-y plane). The original idea was to use overall velocity to control volume, similar to how the bowing motion effects volume on a stringed instrument. Vibrato was linked to roll in an attempt to relate it to the next most natural motion. The Wii-mote controlled the same timbral effects as the EWI but in the Wii-mote's case, this was linked to physical pitch (not frequency). The IR tracking was not used as instrument play does not typically include pointing at abstract locations. That is more of a conducting style interaction, and not natural musical behavior for detailed performance.

Unfortunately, it turns out none of the Wii-mote behavior used is entirely precise and the behavior is not always obvious. Firstly, the idea of relating volume to the velocity of the Wii-mote turned out to be effectively impossible to implement. It turns out derivations of velocity of motion from the Wii-mote use the IR tracking, not the accelerometers. In fact, an initial pass at integrating the acceleration signal found it always increasing. Looking at the Wii-mote's behavior and using the stationary reading as (0,0,0), it usually registers far more positive acceleration than deceleration. Integrating the data it provides, the Wii-mote only gets faster and barely slows down. Obviously this does not follow either the laws of physics nor the actions that are actually happening. More problematically, the behavior was somewhat dependent on the axis and movement involved. The only guarantee was that there would be a reaction which was reasonably repeatable.

There were a few passes to deal with this, from considering the derivative, using the acceleration directly to the chosen route using a variant on a windowed integral. The speed of movement was approximated by taking the absolute value of the difference between the momentary acceleration readings and the stationary readings over a period of time. This made sure that even if the acceleration over a movement did not have the expected equal positive and negative components, the computed value would eventually return to zero when the Wii-mote was again stationary. As might be expected with a window, making the window larger smoothed it, reducing some of the effects of variability at the cost of reduced reaction time. This solution only slightly suffered from drift away

(a) Wii-mote Picked Up   (b) Wii-mote Flat   (c) Wii-mote on Side
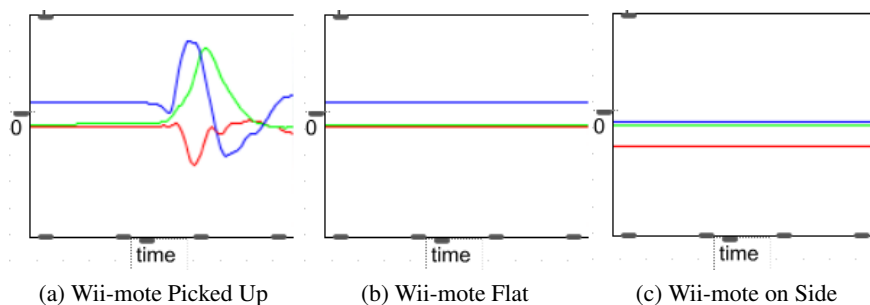
Figure 3-13: **Acceleration data from the Wii-mote–** The first graph is data from picking up the wii-mote. The y axis (blue) reacts largely as expected with both positive and negative components compared to its sitting value. However the x (red) and z (green) both show only positive (z) or negative (x) acceleration which is not expected. The remaining graph demonstrates the difference in base acceleration readings while at rest. The first is the acceleration readings when sitting flat, while the second represents the acceleration readings with the Wii-mote sitting on its side. All data is arbitrarily normalized to range between 0 and 1.

(a) Windowed Integral per Axis   (b) Total Windowed Integral as Wii-mote Picked Up   (c) Total Windowed Integral as Wii-mote Shaken
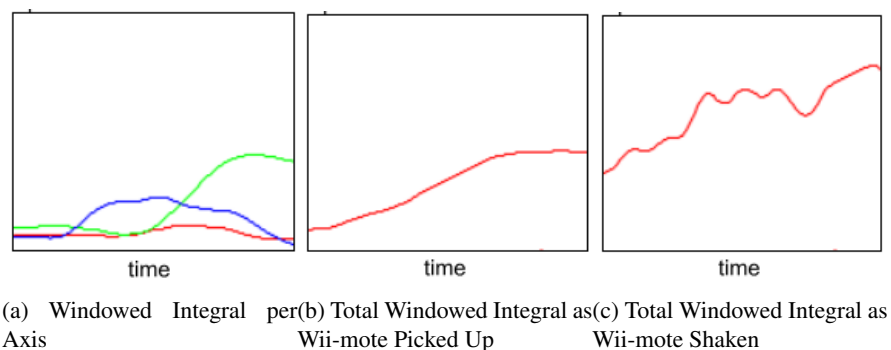
Figure 3-14: **Windowed Integrals from the Wii-mote–** All three images demonstrate a windowed integral calculated while moving the Wii-mote. The first two demonstrate the response found by picking up the Wii-mote, one displays the windowed integral for each axis, while the second is defined by the sum of the axes. The third image c) is of a cumulative windowed integral that occurred while shaking the Wii-mote. Notice that it is not a smooth curve. Extending the window length will alleviate this irregularity at the cost of reaction speed. This data has an arbitrary range.

(a) Altering *pitch*   (b) Altering *roll*
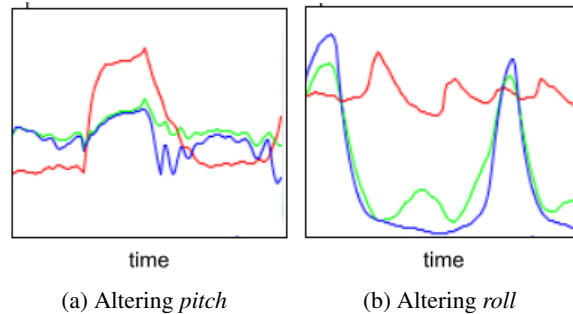
Figure 3-15: In both graphs, red represents *pitch*, blue *yaw*, and green *roll*. The data arbitrarily ranges between 0 to 1 on the Y axis. The first image is the result of changing the *pitch*. It is easy to see the red line reacting as expected with some noise seen in the other two fields. In the second image though, the wii-mote is rolled. While the *roll* data is as expected, *yaw* also follows it.

from the stationary calibration and seemed like a good workable solution. This held true until the Wii-mote is rolled, at which point the stationary x-y-z readings change due to gravity as depicted in Figure 3-13 and the calibration used for calculating baseline differences is no longer valid. Rather than deal with the complications required to properly incorporate gravity, the code was left the same. The result was that in practice, it was easier to effect volume by rolling the Wii-mote than shaking it. This was not as intended but was the situation for user tests.

Yaw is another troublesome data input as its behavior often follows *roll*. Thankfully *pitch* and *roll* provided sufficient continuous options that it did not need to be used and was not a problem. *Pitch* and *roll* are reasonably reliable, only passing false readings in odd positions.

3.2.4   Bringing it All Together

As previously discussed, all the components for real time performance are embedded in a single MAX for Live patch. This includes the AUTimePitch˜ MAX/MSP external for time-stretch, means for reading the extracted audio to play back, and to load the event file that describes the event locations. All the expression mappings previously described are hosted in this patch as well. This includes parsing whether an event is a note trigger, and the resulting control of the audio play back.

MIDI messages from the EWI and Fender MIDI guitar are passed into the computer through a MIDI interface such as an Avid MBox or MIDI keyboard to Ableton Live. Live automatically routes the

MIDI to the re-performance Max for Live Patch. The Wii-mote uses Bluetooth which is intercepted by Osculator, robust software for grabbing the Wii-mote data. Osculator is capable of interacting with other Bluetooth devices but the Wii-mote is clearly a popular choice as it is well supported. Osculator comes with sample files already configured to parse Wii-mote data and pass it along through OSC. It also comes with a MAX patch that, combined with CNMAT's osc.routing external, demonstrates bringing the Wii-mote into MAX. This was re-purposed as the basis for Wii-mote data mapping computation. Audio for the accompaniment is handled entirely within Live. The MAX patch handles the solo, reading the audio from the extraction and performing all processing apart from expression effects. MAX then hands the altered extracted audio solo to Live which routes it through the user controlled timbral effect before mixing it with any active accompaniment and sending out to speakers.

Figure 3-16: The primary user interface for Expressive Re-performance. It includes buttons for opening audio and analysis files, selecting instruments and EWI performance mode, resetting the song, calibrating the Wii and visual feedback on expressive input and data inflow. Double clicking on the scroll patch will bring up the visual progress reference.

The re-performance patch has a small GUI that embeds in a Live window. This GUI contains buttons so that the user can select and load the necessary previously generated audio and audio event files. It should be obvious, that without these, the system does not work. There is a tab for selecting the instrument. The instrument tab switches which continuous controller data acts on the output within the patch. The Wii-mote continuously sends data when on, and the EWI is capable of doing the same. Multiplexing the data is necessary to avoid conflicting data arriving at output actuators. The GUI also contains the button required to calibrate the Wii-mote and save the stationary state. None of the pre-processing functionality relating to extraction or analysis are included.

Additional buttons include a button to toggle between the previously discussed EWI fingering modes

60

and also to reset the audio the user controls back to the first event in the song. This does not impact the accompaniment. Controls for the accompaniment are presently all done through Live. The Live host file is set up so that hitting textitplay using /textitspace will start looping the accompaniment audio. Changing song requires switching tracks and modifying the loop zone to match the different track length. In future, this feature could quite easily be integrated into the Re-performance patch so that the Live UI never needs to be touched.

There are also a few items included to provide visual feedback. If OSC data is being received and routed, the"OSC" LED will light up as will the"note" button if there are note events. Live already has a small indicator for MIDI data being received. These three can be used for any troubleshooting. Errors tend to be simple, such as interfaces not turned on, files not loaded, or the MIDI input connection has been lost and Live needs to be restarted.

The pitch and width knobs are for visual feedback only. Pitch displays the pitch bend derived from vibrato (not pitch shift due to improvisation) while the warmth knob reflects timbral changes on the solo line. A warmth value of one corresponds to no timbral effect. There is a box to indicate when improvisation is toggled on the EWI, and also the output gain is displayed, showing both the volume set through the controller and the actual output level of the audio.
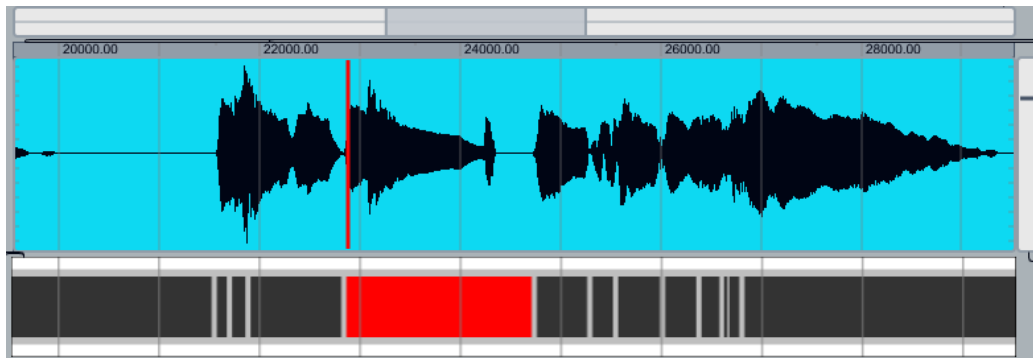
Figure 3-17: The event scrolling window presents visual progress through the song. The waveform is displayed with the note event sequence just below it displayed in the same time frame as the waveform. The event presently played is highlighted in red. The uppermost bar relates where in the song the user is.

The last feature in the GUI is the scrolling button. Double clicking on the button brings up a GUI that depicts progress through the song. Though not strictly required, it can sometimes be difficult for the user to keep track of where they are in the song. There may be forgotten ornamentation in

the recording that is parsed as quick short notes. These short notes easily become stumbling blocks. More confusingly, if a song has a series of repeated notes, it is easy to lose track of which note the user is on. If the user makes a mistake, without visual feedback, it can become especially difficult to know how to get back on track.

The scrolling visual is designed much like the GUI for editing event triggers. On top is the audio waveform with a red bar to mark the location of the beginning of the note just triggered. Below that is a depiction of the note events. Since no pitch information is ever stored, it is one-dimensional conveying only the rhythm and expected note timing. A black bar indicates the start and end of the note event. If the bar is red, that indicates it is the event being played. The scroll window is fixed to display 10 seconds of time, as recorded, which should be long enough to ensure it includes multiple note events (unless there are long rests) and that short events are still discernible. It scrolls smoothly in response to note progress and completes the user's expressive re-performance experience.

Chapter 4

Evaluation

Testing was split between the pre-processing tasks and the real-time user interaction. There are two tests of any artistic technological endeavor: does the technology work as intended, and is the experience successful? Testing of pre-processing tasks focused on technical achievement. The focus of testing for the real-time user interaction was experience based. The real-time user tests also implicitly tested technical functionality.

4.1 Pre-processing Tests

4.1.1 Audio Extraction & Source Separation

Audio extraction and source separation was tested through extractions attempted on segments from eight songs using a variety of mimics for a total of 13 extractions. Five of the songs included tests that were done using original unmixed tracks from the master for the labeling mimic rather than an actual mimic. Using the original was useful for validating that deviations in the extraction from the original were a result of the algorithm, not a poor mimic. In three cases, an extraction was done using a real mimic along with the master-based extraction in order to appreciate the impact of the mimic on the extraction quality and to validate that a mimic differing from the source audio worked. No unmixed master tracks were available for the three remaining songs, and the target audio

could only be labeled through real mimicry. Only the Larry Linkin and Coltrane extractions used any sort of pre-processing before extraction. Simple filtering to reduce potential excess harmonic noise noticeably improves the extraction. Many of the test cases are vocal extractions, whereas the extraction process will primarily be for instrumental purposes. This was due to limited availability of master tracks for instrumentals. Otherwise, the tests strove to span a diverse range of extraction targets and a reasonable range extraction means.

Extraction Targets–   The song segments tested were:

- Oasis's "Wonderwall"– Three extractions performed, all starting at .49 seconds and lasting 11.33 seconds. The vocals and bass-line were extracted using master tracks with an additional extraction of the vocals using a vocal mimic [36].

- Radiohead's "Nude"– The lead vocal was extracted using the source vocal track starting at 1.27 seconds for 9.07 seconds [62].

- George Cochran's "Entertaining Circumstances"– Two extractions, one the lead vocal, the other the bass, starting at 1.44 seconds and lasting 9.07 seconds. Both were extracted using master tracks [31].

- Larry Linkin performing Jobim's "One Note Samba"– The lead clarinet from the entire 3 minute song has been extracted. There is no source audio in this case. The extraction mimic was generated using a MIDI keyboard playing a guitar synthesizer from Ableton Live. This was spectrally and temporally filtered before extraction [40].

- Larry Linkin performing Gershwin's "Summer Time"– The lead clarinet for the first 50 seconds, over 2 million samples, was extracted using the same mimic process as for "One Note Samba". The mimic was spectrally and temporally filtered before extraction [37].

- Ableton Live Toy Set– A melodic lead was extracted from a personal toy Ableton Live session. The extraction was performed using a mimic played on violin.

64

- Pardue's "Remeji"– The lead violin was extracted from my own composition/arrangement titled "Remeji". It was extracted twice: once using the original source solo, and once using a mimic played on violin.

- John Coltrane performing "Greensleeves"– The first of 97 seconds, over 4 million samples, of Coltrane's lead saxophone was extracted using a MIDI based guitar mimic. The mimic was spectrally and temporally filtered before extraction [60].

Parameters for Optimization– Besides validating that the algorithm worked, tests were intended to investigate the impact of different parameters:

- Performance for different number of algorithm iterations.

- Performance for different numbers of target descriptor components, M.

- Performance for different numbers of backing descriptor components, N.

- The effects of introducing sparsity on different vectors to reduce effect of weak components.

- Effect of spectral and temporal filtering pre-extraction.

- Different risk boundaries for the Bayes binary masking.

- Effective means for performance evaluation.

Qualitative results are the most important measure of performance with audio extraction. Results from quantitative analysis do not always correspond well to user perception of "best". This presented a challenge for determining a metric to judge the best quality of performance. The Mean Square Evaluation (MSE) between the extracted target and mimic was found to be a useful quantitative standard. During early testing, it was found that the absolute results from MSE are not reliably accurate to identify best extraction, but trends in MSE are often linked to better performance.

The Blind Source Separation Evaluation (BSS_Eval) Toolbox [35] by Fvotte, Gribonval, and Vincent located at `http://bass-db.gforge.inria.fr/bss_eval/` was also used. It works better for speech, and results do not always correspond to best audio; but it is a good option. The

BSS_Eval Toolbox results compare the wave of an original source versus the extraction, returning the Signal to Distortion (SDR), Signal to Artifacts Ratio (SAR), Signal to Noise Ratio (SNR), and Source Interference Ratio (SIR). SIR measures the leakage between the two extracted tracks and is the best measure for success. SDR, SNR, and SAR relate more closely to general audio quality. A high SDR, high SNR, low SIR, and low SAR are preferable. SNR did not provide very useful results for the mimicked extraction test case and was not considered in tests. When trying to determine parameter optimizations, rather than original source, the mimic was used in evaluating lead target separation. The extraction and the mimic are not usually the same, so some errors stemming from the discrepancies between the two are inevitable; but enough similarity should exist to allow for trend based evaluation and using the mimic/extraction is the best option when the true original lead is unavailable [34].

MSE is calculated while still in the Fourier domain, which is the format required for most of the algorithm. BSS uses the wave version of the data. Computing the wave requires non-trivial computation, making MSE particularly valuable, as it is less computationally expensive. It is hence used in intermediate stages within the algorithm.

There are no practical publically available implementations of audio extraction except through center channel isolation (CCI). Some comparison has been done using one of the most basic means of extraction, the straight application of a Bayes Binary Mask. Otherwise, direct comparison with other extraction means has been restricted to extractions using Virtual Dub center channel isolation. CCI only really works on voice, so comparison is further limited to vocals only.

4.1.2 Event Analysis & Score Generation

In order to assess the event trigger detection, test statistics were collected on the results from the two primary audio samples that are the focus of the remainder of this thesis, clarinet arrangements of George Gershwin's "Summer Time" and Antonio Carlos Jobim's "One Note Samba". Event analysis was performed on the mimic for extraction, rather than the extraction itself. The mimic works better in practice than using the extraction for event analysis. The mimic should be spectrally cleaner, and the re-synthesis should follow it tightly enough that the analysis results are accurate

for both sets of audio. For collecting statistics, the mimic is even more helpful, as in this case, both mimics were created using MIDI based synthesizers and had accompanying MIDI files, enabling accurate timing comparison.

(The existence of a MIDI score may raise the question, why perform event analysis at all? As voice or other instruments are used in extraction, a MIDI file is not always expected to be available. If an extraction is done using a MIDI based mimic, there is a MAX patch for converting MIDI into a format compatible for expressive re-perfomance. In this case, the MIDI events can be used without requiring the analysis stage, although minor editing may still be required if there are differences between the extraction and the mimic.)

The "Summer Time" segment analyzed was 50 seconds long, and contained 49 distinct note events. The "One Note Samba" segment was 94.74 s long (over 4 million samples) with 318 note events. While the "Summer Time" sample had a moderate tempo with significant pitch variation, "One Note Samba" is both faster, containing appropriately shorter notes, and has highly repetitive sections with many notes at the same pitch. It is a more difficult test for finding re-attacks and quick pitch changes.

Aside from technical tests to verify code correctness, the GUI did not receive user testing. It was not included in the real-time test, as that already contained substantial instruction, and including additional GUI tests would have been overly demanding. Also, the GUI follows many industry-standard MIDI interaction and editing styles and is therefore not particularly novel.

4.2   Real-Time Performance

While the pre-processing was not user tested, the real-time user interaction was. The user test consisted of having the user use both the EWI and the Wii-mote to step-through an extraction of Gershwin's "Summer Time" or Jobim's "One Note Samba", both played on clarinet by Larry Linkin. As previously stated, the Fender interface was only on loan for part of the thesis and was unavailable during formal testing. Each person was provided the written instructions included in Appendix A along with an oral version and small demonstration. The instructions provided details on what controls, as described in Section 3.2.3, were available, and how they worked. Otherwise, the basic

goal of the experience, to provide an expressive tool for novices, was briefly explained. Further instruction was basically to try it, and to please consider expression, as it is otherwise easy for users to get caught up in just re-playing the piece.

The primary test piece was the first verse of "Summer Time", which is roughly 50 seconds. This piece was chosen over "One Note Samba" both because the former is more popular and provides more expressive opportunity than the more rhythmical "One Note Samba". "One Note Samba," which was available in its entirety at 1 minute 35 seconds, was offered to people who moved through re-performance of "Summer Time" quickly or were particularly enthusiastic. There was no time-limit on the length of interaction. Users could experiment for as long as they chose. Normal interaction ran from around 20 minutes to over 90 minutes, with most people spending around 30 minutes. Users were started off without the accompaniment and were free to turn it on and off as they chose. Half the users were presented with the EWI first, and half started with the Wii-mote.

Tests were performed with 10 users. Of the 10, three had substantial experience with a clarinet, saxophone or other wind instrument. Though two of them are music professionals, none of the three play a wind instrument professionally. Four users were complete musical novices with little to no experience while the remaining three test volunteers were professional musicians without experience playing wind instruments.

Upon completion of the interaction, users filled out the questionnaire included in Appendix E. The survey starts by asking about the interaction using the EWI and Wii-mote. For example, how easy was each interface to understand and use? Was an interface expressive? And was the interaction interesting and fun?

Chapter 5

Results

In short, the project worked as it was intended to. That is, it functioned sufficiently that the test users could experience a meaningful interaction. Analysis of pre-processing tasks was largely but not entirely objective. Although it would seem that extraction quality should be an entirely quantifiable metric, mathematical evaluation was used to provide significant performance insight but cannot yet entirely replace user judgment. The results of event analysis are fully objective. The success was determined through statistical correctness. As no user experience for extraction has been built, neither pre-processing subsystem received user tests. In contrast, the experience created through the real-time system from the mappings, responsiveness, audio, and success-enabling expression were only testable through user trials. Technical assessment of real-time systems was carried out implicitly through the real-time user experience.

5.1 Pre-processing Tests

5.1.1 Audio Extraction & Source Separation

The overwhelming result from audio extraction tests is that the extraction process works. The best evidence for verifying the extraction performance is through listening. The website Audio Source

Separation in a Musical Context that accompanies the work in this paper provides a full discussion of the audio extraction that was performed, along with sample extraction results. The Results page, the Future Work page, and the audio folder (*http://mit.edu/˜punk/Public/ AudioExtraction/audio/*) contain links to the extractions performed per the list of extraction targets from Section 4.1.1.

Qualitative Audio Results   If one were to discuss each of the extraction targets listed in section 4.1.1, the result would be to show that in each case, the extraction worked, and worked well. More interesting is to review the first extraction, and then provide brief commentary on any unusual or notable results in further test targets.

The extraction of the "Wonderwall" vocals are fairly clear and contain very little leakage from the backing. This is true not only for the extraction using the original source audio, but also the extraction performed using a vocal mimic. The target solo has been fairly well eliminated from the backing in both cases. Add the solo and the backing back together and, apart from the conversion to mono and a mild deadening, the result is a sound very close to the original. Neither the source nor the vocal mimic have even received any pre-filtering to reduce spectral and temporal noise, which definitely would have improved results. Compare this with simple spectral masking. The spectral mask extraction either sounds like the original mimic or else retains significant leakage, making the end product completely unsatisfactory.

The extractions for "One Note Samba" and "Summer Time" include pre-filtering to remove the chance of leakage outside the primary activity in the mimic. Pre-filtering provides clear improvement as compared to cases where it is not used. One interesting artifact from the extraction process is the section where a second clarinet joins the solo clarinet around the 33rd second. In the recording, the second clarinet follows the solo clarinets exactly in musical thirds. The extraction includes both clarinet parts, only dropping the second clarinet when it finally shifts harmony. The thirds in the extraction is actually an expected result. The audio is segmented during the algorithm processing, and that segment features the two instruments together almost entirely. As the parts move together and there is no reference for the solo clarinet alone, the algorithm classifies the two instruments as a single instrument with a spectral characteristic combining the two. This inability

70
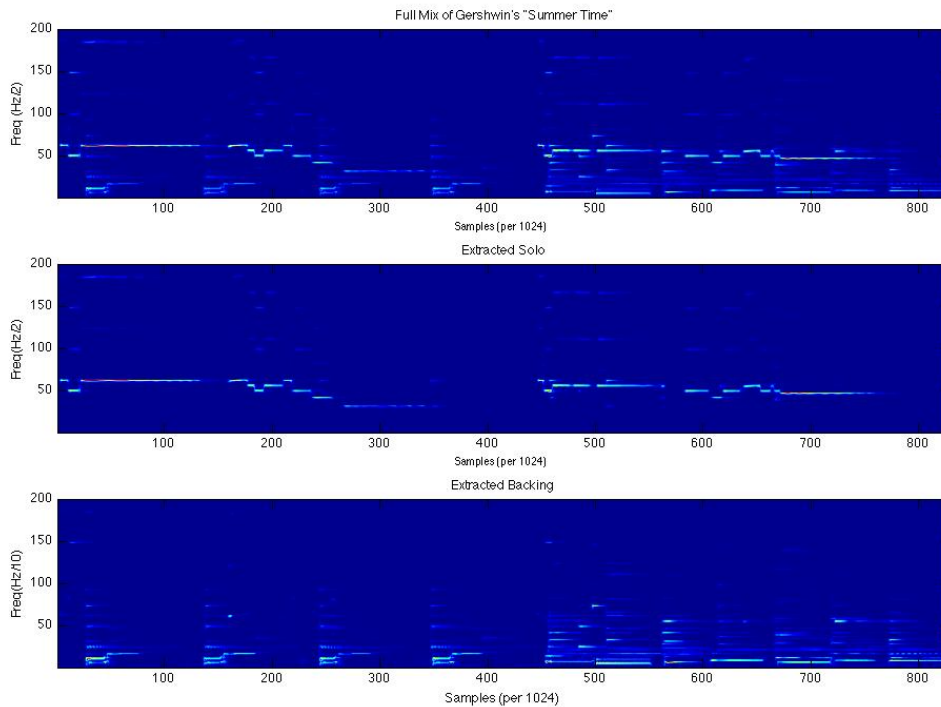
Figure 5-1: Sample Spectral Separation of Opening of Gershwin's "Summer Time". As in Figure 3-5, each graph shows the spectral power of at different frequencies as the song progresses. The top graph is again the full mix. However, in contrast to Figure 3-5, the bottom two graphs are the actual extraction. The middle graph is the extracted target and the bottom graph is the remaining separated backing. Notice how successfully this set of graphs matches Figure 3-5.

to distinguish parts moving in parallel is a clear limitation. It can be overcome to some extent by making segments include music without overlap between parts, but it will still happen. In practice, it was not particularly distracting and, overall, is a tolerable flaw.

The extraction of the melody from the toy Ableton set is highly successful and interesting for the use of violin for the extraction means. Likewise, the success in extracting the violin solo in the Remeji segment is highly encouraging. Strings are notoriously difficult to deal with in separation and synthesis due to the complex extensive resonance patterns. The Remeji example is made harder as there are other string parts in the mix and there is extensive reverb on the solo and mix. The extractions are not as clean and clear of interference from the accompaniment as many of the other examples but the extraction process still works admirably well. The elimination of the violin from

the accompaniment is entirely convincing. Again, these extractions were not performed with any pre-filtering which would undoubtably improve results.

One case where the extraction provides mixed results is the extraction of Coltrane playing "Greensleeves". The thing with this extraction is that the accompaniment sounds good. The removal of Coltrane's saxophone is fairly complete. The complete mix of re-synthesized audio, i.e. the recombining the solo and backing, also sounds good. What is surprising, is that the extraction of the solo does not sound good. Part of this may be the recording—it sounds like it was recorded live without many channels and the saxophone does not sound that great in the original mix. Additionally, this extraction was done from an mp3, a compressed source, so that may be a source of distortion. Nonetheless, it is not clear why the solo extraction sounds as poor as it does.

Optimizing the Algorithm   Investigations into optimization of the algorithm produced a number of useful results. The previously referenced web-site Audio Source Separation in a Musical Context contains details of findings and audio examples of how different parameters affect the output. Though this paper usually talks in terms of optimizing the lead extraction, an equally important target is removal of the lead to obtain the backing. The quality of the removal is definitely relevant, especially outside re-performance. Optimizations for the different parameters are not always the same for the lead and the backing.

Accurate Evaluation Criteria   One of the main challenges is determining the quality of an extraction. MSE and the BSS_Eval toolbox presented in Section 4.1.1 were used but as will be seen, were imperfect judgment tools. Within one set of parameters, MSE is a reliable best differentiator. When parameters are different, it is far less consistent. Figure 5-2 shows two pairs of extractions, one for the vocal in Oasis's "Wonderwall", and one for the violin solo from my own "Remeji". Notice how, after starting "low", the MSE increases with M for the lead extraction with "Wonderwall". Knowing that more accompaniment will bleed into the extraction as M increases, this behavior might make sense and suggest a good value for M. But then consider how MSE decreases with M for the lead from "Remeji". The behavior is nonsensical and not backed up by listening tests. It is consistently true that for a certain M and all other parameters not changing, the extraction with lowest MSE will

be the best. Some typical pattern recognition metrics like Kullback-Leibler Divergence are not ideal for determining optimal *M* as the process just splits information differently between more components rather than loosing information. Determining when MSE and BSS are useful is important. They are often useful in suggesting optimal behavior, but only audio audition can really choose the best extraction.
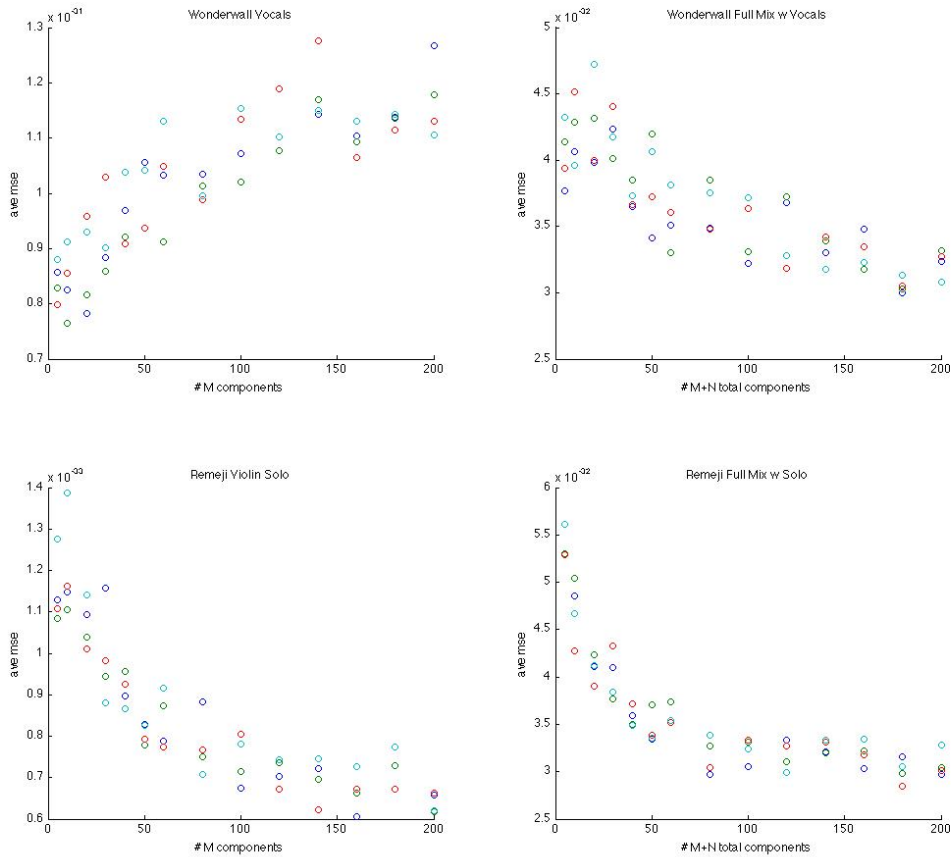
Figure 5-2: Investigating the impact of large changes of *M* on MSE. The left pair of graphs consider finding optimal *M* for the target solo only. The right pair consider finding optimal total components $M + N$ for the entire mixed song. The color indicates which extraction iteration a data point originates from.

Algorithm Iteration   Proceeding through the parameter investigations listed on in Section 4.1.1, the first target was the number of algorithm iterations in the separation stage. Figure 5-3 demon-

strates that excess iterations are not only computationally expensive, but do not necessarily yield improved results extracting the solo. The accompaniment extraction does improve. Discrepancies in the extraction are more distracting in the lead target so optimizing for best results with the solo suggests roughly 40 iterations. This was a fairly consistent finding backed up by audio auditions.
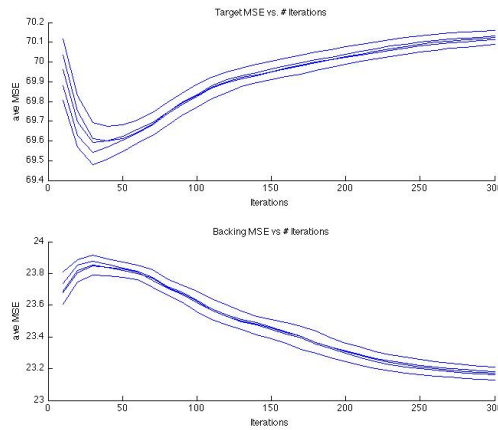
Figure 5-3: Investigating the impact of algorithm iterations on MSE.

Optimal *M* and N   In many ways, the most important variable is *M*, the number of components used to describe the mimic. Too many, and there is too much bleed through. Too few, and the lead audio becomes a wash. Appropriate *M* will change for each extraction segment depending on its complexity. *N*, the number of components used to describe the accompaniment, is similarly relevant but has a much wider acceptable range. As long as *N* is sufficiently large (100), the solo extraction will not be significantly impaired by non-optimal *N*. The impact on the quality of the extracted accompaniment however, is significant. If *N* is too small, it can not sufficiently describe the accompaniment. A large *N*, however, actually reduces bleed and often sounds better. The drawback is that a large *N* increases the computation. There is no definitive value for optimal *M* and *N* making automated guess valuable. Figure 5-4 demonstrates the impact of changing *M* and *N* during an extraction of Oasis's "Wonderwall". Figure 5-5 depicts the same test but collecting BSS_Eval metrics. As a reminder, a high SDR, low SIR, and low SAR are considered good. For each *M*, a range of *N*s were tested and vice versa. Each *M*, *N* pair was tested three times in a simple attempt to deal with local extrema.
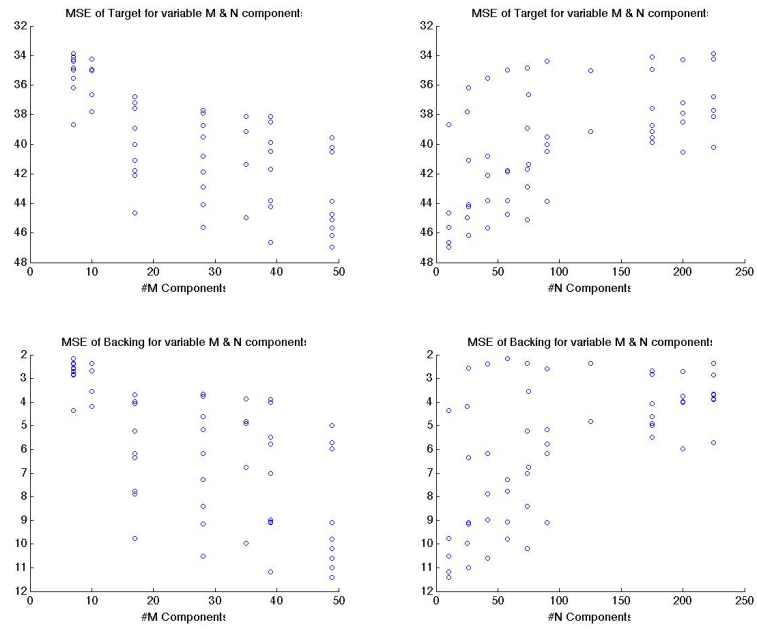
Figure 5-4: Impact of *M* and *N* on MSE for separating lead vocal and backing of Oasis's "Wonder-wall"
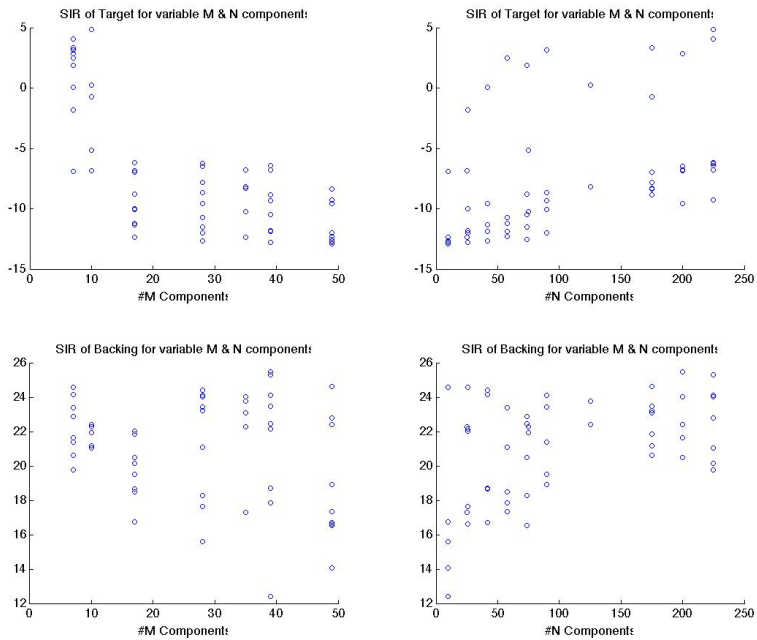
Figure 5-5: Impact of *M* and *N* on SIR for separating lead vocal and backing of Oasis's "Wonder-wall"
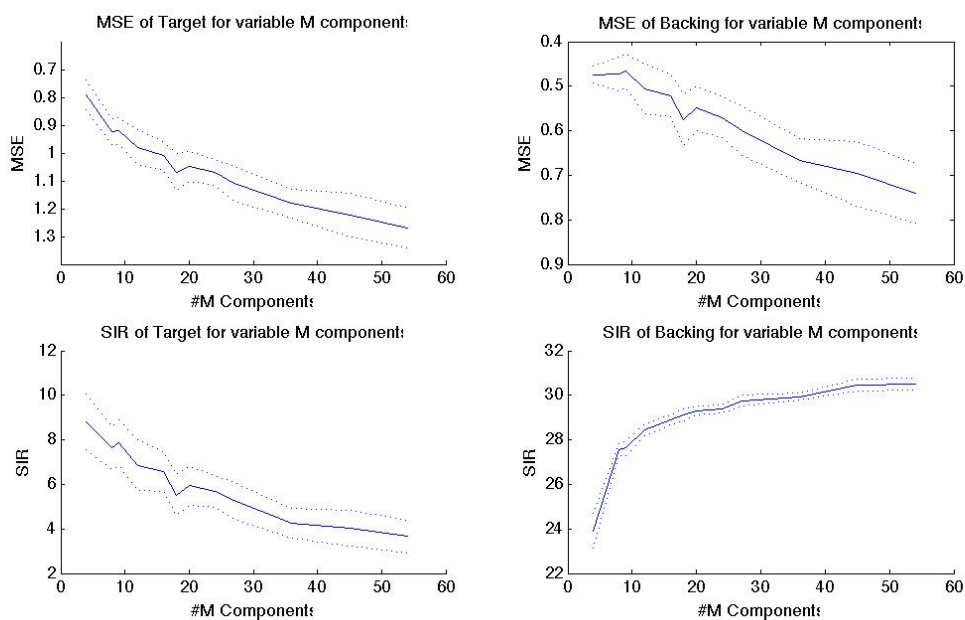
Figure 5-6: Impact of *M* on MSE & SIR for separating solo violin and backing from "Remeji." Only the mean and one standard deviation from the mean are shown.

The results for MSE echo the results for SIR, which is a good finding. Note they both suggest poor performance for a low *M*. In the case of extraction, SIR is most relevant metric to suggest a good extraction. The SIR in particular suggests that between 10 and 16, there is a break point where the extraction considerably improves. Listening to audio reconstructions confirms this finding. Listening tests indicated a minimum of 12 components to sufficiently describe the "Wonderwall" vocal. Beyond this critical point, the extraction quality becomes a trade-off of clarity vs. leakage. Above *M* equal to 12, more backing creeps in even if the extraction is otherwise better. As suggested in Figure 5-2, the nature of the trade-off between clarity and leakage results in questionable usefulness from MSE and BSS evaluation results for higher *M*. However, the breakpoint for minimal *M*, demonstrated through both MSE and the BSS evaluation metrics, will not tightly identify the optimal extraction, but does provide the key target area for optimal *M*. Audition results have determined that minimal but sufficient *M* is where the best extractions tend to occur. Figure 5-6 provides another example of how MSE and SIR track for different *M* in the extraction of the violin solo from the author's "Remeji".

Sparsity   Sparsity is a tempting option to implement. Sparsity will successfully remove bleed-in from the accompaniment. The drawback is that sparsity is also a loss of information. Sparsity in the component latent variable matrices W and H clean-up unwanted frequency elements and temporal insertions respectively. However, due to loss of information, sparsity increases artifact and distortion. Sparsity in W results in frequency wash and sparsity in H results in distortion. While sparsity may be valuable in voice extraction, it did not benefit music extraction. Personally, I preferred results without any sparsity, even if those with sparsity had less bleed from the accompaniment.

Binary Masking   Smaragdis suggests the use of a binary mask to reduce unlikely spectral noise at the end of extraction. A variable Bayes mask was implemented to test this option. Like sparsity, it did clean the audio up somewhat but at the cost of introduced artifact and distortion. However, performing a low thresholded binary mask before extraction did not lead to the same distortion and was similarly effective. The binary mask essentially acts as a spectral and temporal threshold power based filter.

User Impressions on Audio Extraction

Generally, the results from the extraction have been regarded as successful. Again, the only true way to judge the extraction is by listening. When asked about sound quality of the audio during real-time tests, most users gave it high marks with the average sound quality rating 5.11 out of 7. Given that the original recording itself is not of great quality, this seems quite successful. Asked if the solo extraction sounded like a clarinet, the average rating was 4.89 out of 7. Interestingly, all three wind players rated the sound a 6 or 7. Taking into account degradation through time-stretch and potential for the "thinning" timbral effect to reduce the perceived quality, this can be interpreted as being quite successful.

Though not specifically asked about extraction quality, a telling result is that it was not a distraction. At no point did any user complain about sound quality or remark negatively on the extraction results. There was only handful of remarks about a spot where bleed from the piano into the solo caused confusion. Users were not told in advance that the audio in the real-time test was an extraction.

77
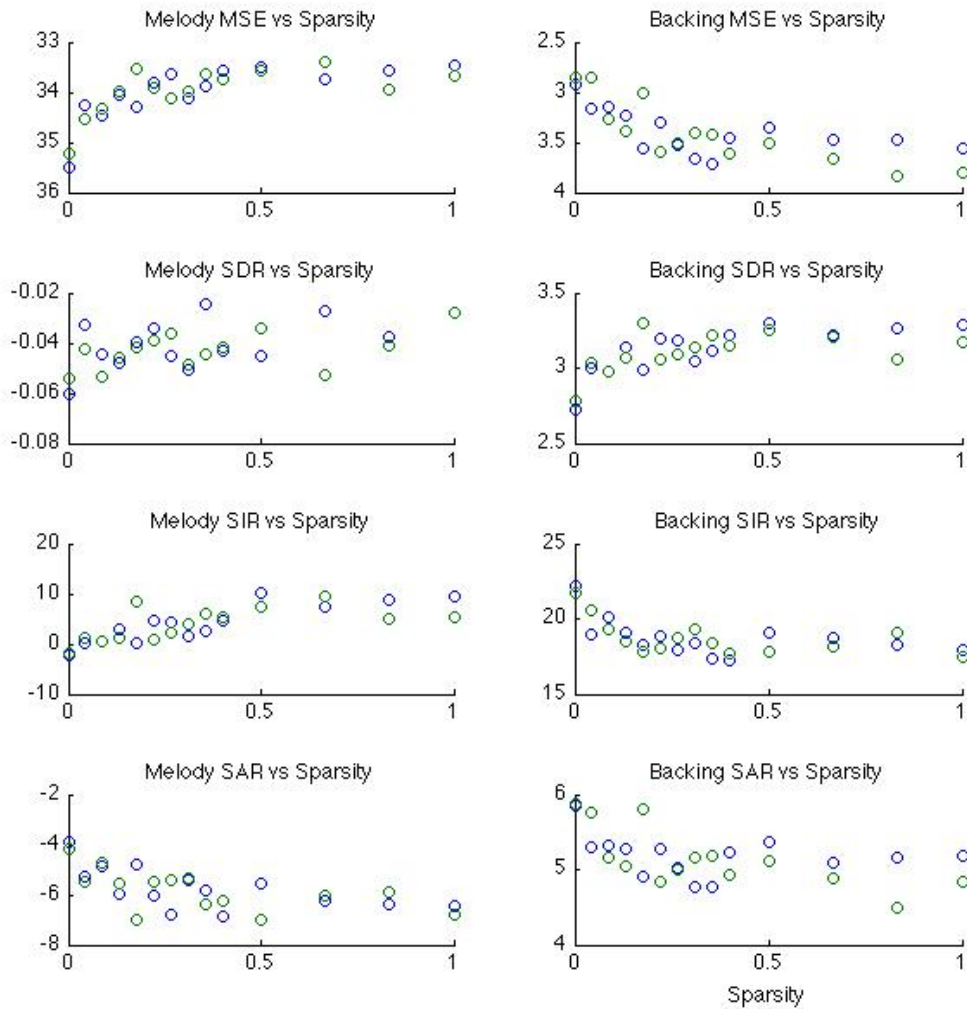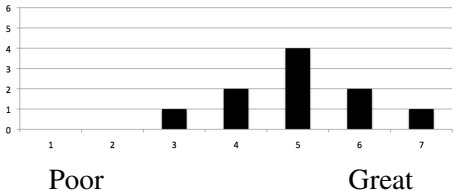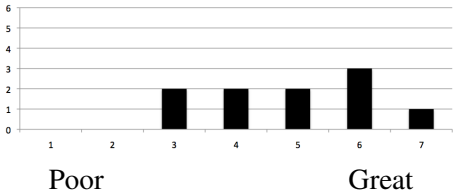
Figure 5-7: Impact of Sparsity equally in W and H on MSE, SDR, SIR, and SIR for separating lead vocal and backing of Oasis's "Wonderwall". Though this example suggests no sparsity is optimal for the sample extraction based on both MSE and SIR findings, for many extractions this will not be the case.

Table 5.1: User Response to Audio Quality

Question	User Responses	Ave Score
1) How did you find the sound quality?	Poor Great	5.0
2) How did the sound measure up to a clarinet sound?	Poor Great	4.9

When told afterwards, test users familiar with audio recording and mixing were highly impressed, even amazed.

5.1.2 Event Analysis & Score Generation

While imperfect, the analysis techniques for finding note starts were quite sufficient. A sample of results from the "Summer Time" event analysis is included in Table 5.2. In addition to finding all of the notes, the analysis also needs to accurately identify the note start location.

The automated analyze˜ based MAX/MSP patch *eventDetect* accurately found 47 of the 49, or 96%, of the notes, only incorrectly added one non-existent note. On the longer, re-attack heavy "One Note Samba" it also did well, correctly detecting all but 3 of the 318 notes for a 99% hit rate. It faired worse for false positives, however, finding notes 36 times that did not exist. Many of these occurred at the tail of held notes where the volume drops somewhat and analyze˜ did not work as well. However, the average absolute error between the actual and estimated placement for the start of the note was 464 samples, which at 10.5 ms is somewhat high. Experience during testing suggests that the accuracy needs to be closer to 5 ms to not be distracting to the user. Worse still, the standard deviation for error from actual start was 913 samples or 20.71 ms. This implies a significant number of substantially unsatisfactory outliers.

<div align="center">"Summer Time" Event Analysis</div>

Recored MIDI		Analyze~ Event Detect					Echo Nest Event Detect			
Start Sample	MIDI Note	Start Sample	MIDI Note	Vol (dB)	Start Error	Classification Error	Start Sample	Scaled Start Sample	Start Error	Classification Error
498677	76	498736	76	85.78	59.18		487808	498266.6	-410.22	
507143	72	507952	72	94.03	808.66		496017	506651.6	-491.74	
517663	76	518192	76	98.84	529.47		506750	517614.72	-47.81	
658868	76	658992	76	78.59	124.22		645221	659054.54	186.76	
677120	74	677424	74	89.99	303.86		662930	677143.22	23.08	
683459	72	684080	72	92.36	621.23		669552	683907.19	448.43	
691191	74	691248	74	89.03	57.08		676719	691227.86	36.94	
711555	76	711728	76	91.45	172.53		696724	711661.76	106.29	
719288	72	719920	72	91.51	632.38					Missed
		723504	72	83.47		Extra				
738933	69	739376	69	96.7	442.5		723061	738563.43	-370.07	
768199	64	768560	64	101.38	360.55		752270	768398.67	199.22	
							859793	878226.96		Extra
952472	76	952368	76	78.26	-104.31		932224	952210.88	-261.43	
958092	72	958512	72	95.75	419.74		937629	957731.77	-360.5	
967218	74	967728	74	93.76	510.07		946782	967081.01	-136.93	
984795	74					Miss				Miss
1004441	74	1004592	74	91.56	150.68		983169	1004248.14	-193.18	
							1045912	1068336.35		Extra
							1059996	1082722.31		Extra
1092959	72	1092656	72	90.69	-302.88		1069704	1092638.45	-320.43	
1109817	69	1110064	69	94.07	247.3		1086866	1110168.41	351.7	
1117774	72	1118256	72	98.32	481.86		1094508	1117974.25	200.11	
1147535	74	1147952	74	93.17	417.5		1123730	1147822.77	288.27	
1160392	72	1160240	72	85.81	-152.19		1136483	1160849.2	457	
1176531	73	1176624	73	88.86	92.66		1151592	1176282.13	-249.2	
1178644	71	1180208	71	91.1	1564.47					Miss
							1300113	1327987.42		Extra
1361299	76	1360432	76	80.57	-866.6		1332350	1360915.58	-383.02	
1371143	76					Miss				Miss
1383776	72	1383984	72	94.57	207.63		1354857	1383905.13	128.76	
.	.	.	.	.	.		.	.	.	
.	.	.	.	.	.		.	.	.	
.	.	.	.	.	.		.	.	.	
2032799	69	2032176	69	93.19	-622.79		1990244	2032914.83	116.04	
		Ave Start Error			11.75		Ave Start Error		0.55	
		Ave Absolute Start Dev			491.91		Ave. Absolute Start Dev		257.3	
		Standard Dev			680.69		Standard Dev		298.62	

Table 5.2: Analysis of the notes in the first 50 seconds of Gershwin's "Summer Time". The left data set is the original MIDI timing, the middle set is notes as found by the *eventDetect* MAX/MSP analysis, with the right-most set presenting the note timings found using Echo Nest's analyzing tool. Missed notes or notes added are indicated by their classification error. The average absolute deviation represents the average of the absolute value of the error between the estimated note and the actual note start.

The Echo Nest based event detect did better with error margins. In the "One Note Samba" sample it suffered from an average absolute discrepancy of only 182 samples from the MIDI note start. At 4.1ms, that error window is much more tolerant. With a standard deviation of 243 samples, or 5.5ms, notes found using the Echo Nest based *en_eventDetect* are placed tightly enough to not require user intervention. The Echo Nest event Detect does not fair so well at finding the notes though. It only recorded 4 false hits, much better than the analyze~ version, but missed 58 times, finding only 82% of the notes. In "Summer Time" it did slightly better, finding 41 of the 49 notes for a 84% accuracy, but this time registered 7 false hits, performing worse than the analyze~ *eventDetect* (which only registered one). For both songs, the notes missed by the analyze~ based analysis were also missed

Table 5.3: Hit performance in Gershwin's "Summer Time".

	Total Events	Correct Hits	Missed	False Hits
Analyze˜ Event Detect	50	47	2	1
Echo Nest Event Detect	56	41	8	7

Table 5.4: Hit performance in Jobim's "One Note Samba"

	Total Events	Correct Hits	Missed	False Hits
Analyze˜ Event Detect	354	315	3	36
Echo Nest Event Detect	322	260	58	4

by the Echo Nest version.

At present, there is no automated means for combining the two results for an improved final automation tally. Instead there is the manual editing GUI for revising and correcting analysis which provides a way to test the results of each detection method, select which option did best for each note, and then merge the two sets. The GUI's editing features allow remaining errors to be fixed. For instance, in practice, the extra notes added in a tail by the analyze˜ version are easy to identify and remove through the GUI. Theses capabilities lighten the need for perfect automated results.

5.2   Real-Time Performance

As noted, there are two measures of whether an artistic system works: the technical aspects, and the expressive aspects. Clearly, given that it ran repeatedly in user tests, the technical side worked. Everything implemented behaved as expected. There were some early problems with improvisation on the EWI, but they were quickly fixed. The only other problem was when the EWI was used in a cool damp environment, which caused the whole interface to behave unpredictably. It turns out the potential for this and the need to re-calibrate is identified in the Akai manual. This did lead to one problem with volume control, discussed later. The general sensitivity of the EWI introduced unintended but correctly interpreted events. Similarly, the use of incorrect fingering if the EWI was used in button mode gave the impression the system was broken when notes did not play. Both of these issues might appear to the user like system failure, but were in fact a result of user error.

Table 5.5: Combined hit performance of event analysis techniques.

	Total Events	Correct Hits	Missed	False Hits
Analyze˜ Event Detect	404	362	5	37
Echo Nest Event Detect	378	301	66	11
Combined	415	362	5	48
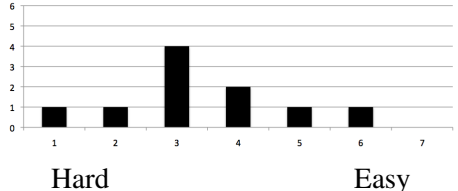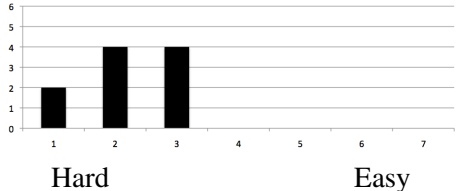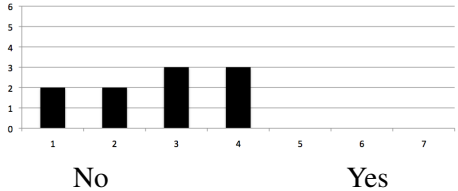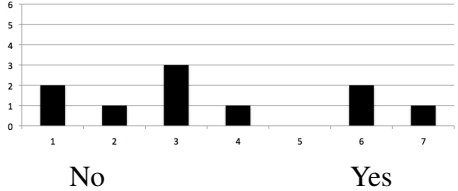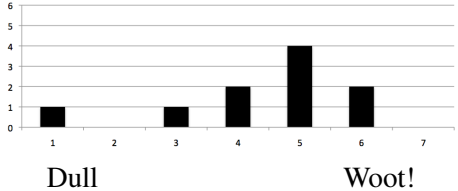
Again, as mentioned in the discussion on audio quality, if no one comments on a potential hidden technical challenge, it generally implies success. For instance, although it was not feasible to test end-to-end latency, the fact that there were no comments about it suggests that it was sufficient. Computational algorithmic requirements mean AUTimePitch˜ must introduce some latency. When used with a smaller window size, the AUTimePitch˜ external did not introduce distractingly-longs delays. In fact, after posting AUTimePitch˜ with Cycling74, a user reportedly tested it and found the general performance tighter than the MAX object groove˜. Again, sound quality was not mentioned as an issue. The time-stretch worked fairly seamlessly.

5.2.1   Re-performance on the EWI

Feedback from the real-time test questionnaire was less favorable. Table 5.6 and later Table 5.8 summarize responses to questions about the general playability and expressiveness of the interaction using both instruments.

As might be expected and as is reflected in user responses, the EWI was definitely more complicated to play. A number of users commented on the difficulty of coordinating breath and fingers. A major issue was the sensitivity of the instrument. Any finger slip changes a note. The fact that pitch is not directly linked to the fingering further dissociates the way fingering works. It is easy to forget that a key is not a button and that picking up a finger will advance a note just as putting the finger down will do. As presented in Section 3.2.3 discussing the EWI mappings, simplifications in code to eliminate accidental slips and the introduction of the button mode were used to try and alleviate difficulty fingering the EWI. Unfortunately, even with the button mode, there are so many buttons and opportunities to rest a finger in the wrong place that it remained an imperfect solution. Wind players, more familiar with thinking about fingerings, did have an easier time with this. Imitating

Table 5.6: User Reaction to Re-Performance using EWI Interface.

EWI Re-Performance Question	User Responses	Ave Score
1) How easy was it to understand how to use the instrument?	Hard ... Easy	3.4
2) How easy was it to control the music the way you wanted?	Hard ... Easy	2.2
3) Did you feel like you were able to use the system in an expressive manner?	No ... Yes	2.7
4) Would you want to spend time using the system in order to use the EWI better?	No ... Yes	3.6
5) Was the interaction fun?	Dull ... Woot!	4.4

how to correctly play the song resulted in quick correct execution. Unexpected improvisation in the recording was the biggest stumbling block for those comfortable with the EWI. Wind players also knew how to hold the instrument better and were unlikely to mis-finger the button mode.
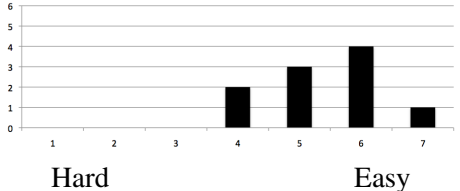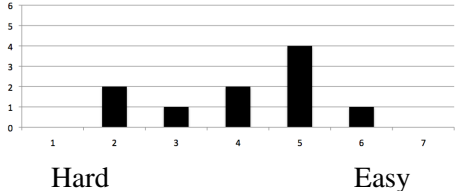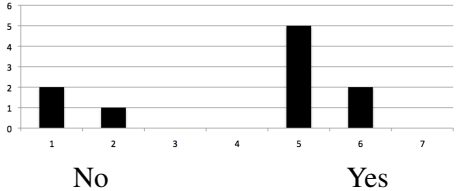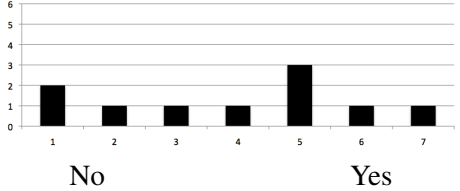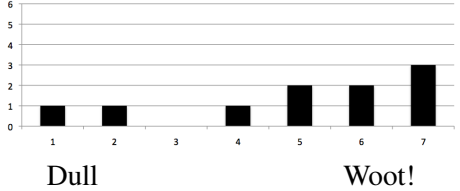
Unfortunately the volume on the EWI ended up being mis-calibrated for the majority of tests. The EWI does require occasional calibration, especially in different weather contexts. One of the first tests was in a very cool, damp environment where the EWI struggled and settings were revised to accommodate. When returned to more normal weather, settings were not revised with the result that intermediate volumes were difficult to achieve. The incorrect calibration was not caught until after a few complaints about a lack of volume control. Volume should be one of the easier control aspects playing the EWI.

Although some users did experiment with the whole range of expressive possibility in the EWI, wind players tended to find it too foreign and novices too new to really consider the expressive range. Frustration over accidental notes and even just learning the EWI is reflected in its low control rating and probably responsible for its low expressivity rating. For instance, most people barely tried out the bite for vibrato or the tonal thinning. The improvisation option ended up being less useful too as it is a bit awkward to engage and therefore less spontaneous and only usable on longer notes.

5.2.2   Re-performance on the Wii-mote

Not surprisingly, the simplicity of the Wii-mote, which uses a single button to advance audio, was rated much easier to understand. As far less mental effort was required to play a note, the expressive areas were easier to explore and it garnered moderate expressivity ratings. Interestingly, one of the things the EWI interaction suffered from was the expectation that it would work like a real clarinet. Not being a traditional instrument, the Wii-mote did not suffer that same expectation. The fact that it is actually quite difficult to separate control of the different expressive variables and play in tune was usually overlooked, probably given the novelty factor and the ease of basic playing. As one user said, "...I felt less expectations with it than with the EWI. So I was more forgiving..." The two people who rated the Wii-mote poorly for expression were the two that wanted to hold it to traditional instrument standards. Still, the Wii-mote was almost universally regarded as fun with the only poor

Table 5.7: User Reaction to Re-Performance using Wii-Mote Interface.

Wii-Mote Re-Performance Question	User Responses	Ave Score
1) How easy was it to understand how to use the instrument?		5.4
2) How easy was it to control the music the way you wanted?		4.1
3) Did you feel like you were able to use the system in an expressive manner?		4.1
4) Would you want to spend time using the system in order to use the Wii-mote better?		3.9
5) Was the interaction fun?		5.0

rating coming from the person who maintained the strongest desire for a traditional interface and detailed control remarking, "Give me back the EWI!" The other traditionalist commented "fun fun fun! :)"

5.2.3   The EWI vs. the Wii-mote

Although comparison of the Wii-mote to the EWI was not an explicit goal of the testing, there were a number of questions comparing the two in an effort to gain additional insight into user experience and the validity of the premise that a user would rather play a mimic of a real-instrument than something entirely new.

Table 5.8: Comparison Between the EWI and Wii-mote.

Question	User Responses	Ave Score
1) Which interface was easier to use?	 EWI .... Wii-mote	6.3
2) Which interface was more expressive?	 EWI .... Wii-mote	4.9
3) Which interface was more fun?	 EWI .... Wii-mote	5.2
4) Which interface did you prefer to spend time with?	 EWI .... Wii-mote	3.3

86

In direct comparison, the Wii-mote out scores the EWI even more clearly for ease of use, with every user rating the Wii-mote equally or easier to use. Similarly, it was strongly rated more fun and more expressive, though those who disagreed, strongly disagreed. When asked which interface was preferred, the EWI out-scored the Wii-mote with only three users preferring the Wii-mote. Comments suggested that the Wii-mote would become boring despite the fact that the two interfaces control exactly the same effects. This highlights an issue with the short time of the test environment and suggests that the extra complexity in the EWI, though frustrating, may be overall more interesting. As one user preferring the EWI expressed, "Challenge!" Overall, it seems the Wii-mote is more of a toy, fun to start with, but the EWI is more of an instrument.

5.2.4 Other User Interactions

The scrolling graphic was generally well received. Added as an after-thought, it significantly eased performance. It gave the user something to look at, but it also helped demonstrate the user's place in the music and identified particularly short notes which, if not quickly stepped through, would result in the audio unintentionally stopping. As the short notes are typically decorations added by the performer, Larry Linkin, this was particularly helpful in learning unexpected improvisations in the song. In a pitch-repetitive piece like "One Note Samba", the graphic was essential to the user for keeping track of progress through the notes. When only using audio, it was easy to get confused as to which one of the many similar notes was being played. There were requests (especially from musicians) for the scrolling graphic to include measure lines and such, but that is neither technically trivial nor necessary once the music is familiar.

As expected, the availability of the accompaniment was almost universally liked. 9 out of 10 people rated playing with the accompaniment a positive experience. For some it provided a framework for better grasping the flow of music. For others, it just made things more enjoyable.

5.2.5 Overall Discussion

An open question about whether the user was able to learn some expression, a major goal of re-performance, yielded disappointing results. Considering the percentage of professional and semi-

professional musicians who were test subjects, this is not too surprising. As professionals, they are presumably already knowledgeable about phrasing and what they might learn is more closely tied to the system, the interface, and the song, not an understanding of musicality. Still, only three users reported learning much about expression or phrasing. Two completely novice users did get excited by their ability to play a non-trivial song. The two also expressed that they enjoyed being asked to think about how to make rather, than execute music.

In retrospect, considering the test environment, the disappointing responses regarding learning about expression were rather predictable . A major issue with the test environment is the length. Although a goal is to simplify music performance, it is neither fair nor realistic to expect a meaningful interaction to develop within just 35 minutes of use. For instance, 35 minutes is a fairly short window compared to the amount of time most active musicians spend practicing each day. It appears the average 35 minute interaction is simply not long enough to allow people to learn how to use both interfaces effectively and then also understanding way the song is played. Many users had to spend too much time figuring out the mundane means to play through the song and did not make it to the expressive parts of the interface. This was especially true with the EWI, which probably takes a few hours to get used to. A common comment was the desire to "grasp the EWI eventually." It seems that expressive music performance cannot be simplified enough to make for a quick learning experience. And frankly, if it was possible, it would not hold people's interest. The Wii-mote was quick to learn, but the more complex and repeatable EWI was considered more interesting long-term.

As for the overall experience, it received positive results with user tests suggesting it was indeed fun. Using the EWI scored a 4.4 fun rating and using the Wii-mote scored a 5.0 fun rating. Re-performance is not intended to only relate to a specific song. In fact, the song choice probably hampered the interaction. The two song options were chosen primarily based on their role in source extraction. Music Minus One is a series of recordings intended for music practice. Each recording has at least two versions of every song, one with the lead and one without. They are neither the best recordings nor the most interesting performances, but because some versions provided the exact accompaniment minus the solo, they were useful for their ability to provide a ground truth in extraction tests. The Larry Linkin recordings were chosen because the two chosen songs, "Summer Time" and "One Note Samba" are familiar jazz standards played on clarinet and recorded in a way

that the backing is identical to the one in the mixed recording with the lead clarinet. Being jazz, the Linkin's performance sometimes deviates from the standard written score.

A number of users expressed frustration at the improvisations added by Linkin. "I felt that including all the grace notes etc in the song was a little counter-intuitive. The way I remember the song is without them, so I pressed a button thinking I would get on to the next main note rather than a grace note. The visual interface helped with this, but it was still a little counter-intuitive." A performer's improvisation is only preferential in the re-performance when the user knows that version exceedingly well. This was certainly not the case here. A better test example would have been something classical as classical music does not deviate from the score.
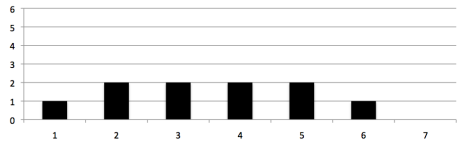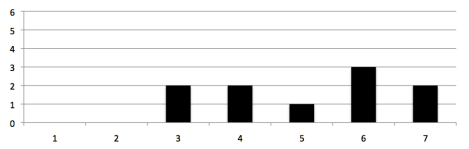
Exacerbating the addition of minor improvisation in the recordings is the fact that a note is only extended to three times its original length. While this was generally fine for the longer notes, short notes, as usually added in Linkin's improvisations, often ended before the user expected, especially when users were less familiar with the recording. This was an almost universal negative finding and was the most frequently criticized impediment to a positive user experience. Unfortunately, the reason for it is the AUTimePitch . A segment cannot be extended beyond four times its original length. This is reduced to just under three times the original length in order to retain the attack and smoothly transition the rate. Although the time limitation on a note was acknowledged as a compromise in advance, the disruptive effect was underestimated. Rubberband, the primary alternative examined instead of the Apple TimePitch plug-in that AUTimePitch~ uses, includes additional algorithms for extremely slow, essentially infinite playback. Switching algorithms will alleviate this problem, but is not something that can be quickly implemented. A solution traditionally used in audio to extend notes is looping part of the sample. Looping a segment might also be workable though again, this is not a quick fix as it requires some significant re-coding.

The last aggressive complaint, was the inability to substantially control pitch. Some people wanted their fingerings to impact pitch and were severely distracted by the dissociation between the two. Not surprisingly, it was the experienced wind players, who were used to fingering on a wind instrument, who felt the strongest about this. This is a problematic complaint seeing as one of the most important ideas in expressive re-performance is to remove the necessity of concentrating on pitch. If a user wants the traditional linkage between fingering and pitch, the re-performance system will not

be interesting to them. That said, there is certainly room for bringing in some level of association if developing re-performance into a learning tool. Slowly increasing the reliance of pitch on fingering would be an obvious learning opportunity.

The desire to control pitch directly does highlight one major failing of the user tests, the user set. As stated, there are two major user groups the expressive re-performance system is targeting: the novice wanting to experience the fantasy of playing their favorite song on the relevant interface; and the somewhat experienced musician wanting the opportunity to focus more intently on just the expression in a piece. None of the users fell into this user set nor would it be easy to find users meeting this criteria, especially when only two songs are presently on offer. As one user said when asked if she would be interested in using the system further, "Not my cup of tea but if I was into playing music then can imagine definitely would want to." Expressive re-performance assumes some level of user investment in both the song and the instrument that was simply lacking in the test environment.

Table 5.9: User Response to Song Re-Performance.

Question	User Responses	Ave Score
1) Would you spend time using the system trying to shape the given song?	Not Interested Very interested	3.5
2) Would you spend time using the system trying to shape a song of your choice?	Not Interested Very interested	5.1

The questionnaire tried to assess foreseeable problems with user investment in the song through two questions: one about whether the user had interest in trying to shape the given song, followed by a question about the desirability of using the system with a song of choice. Table 5.9 details the results. The desire to shape the given song received a slightly negative response, with an average rating of 3.5. On the other hand, the response to the question regarding song of choice was defi-

nitely positive with an average rating of 5.1 and only one negative rating. If a user was invested in the music and interface, the frustration with not knowing the tune would be reduced as presumably it would already be well known, and the knowledge of the interface would be increased to comfortable familiarity, leading to a more positive overall experience. The positive findings for interest in playing a tune of the user's preference is a good finding for the expressive re-performance system.

5.3 A Personal Expressive Experience

As it turns out, the only user meeting the basic criteria of user interest, instrument familiarity, and song knowledge, was me, the author. I clearly have bias as creator and had ruled myself out as a significant source of findings. But while playing with the EWI to ensure it was ready for another test user, I realized I was experiencing re-performance much more as intended. I was seriously considering expression on an instrument I cannot otherwise play. Unlike test checks using the EWI, I rarely test more than necessary using the Wii-mote as I find it too unpredictable and difficult to control nuance. My interest has been in building a quality system and expressive interaction rather than in learning about expression or reliving Larry Linkin's perfomance. In no way did I intend to become a master of the song nor did I intend to learn the EWI. However, during the process of extraction, analysis, system build and test, I became not only intimately familiar with Linkin's version of "Summer Time", but reasonably comfortable using the EWI. I am a semi-professional violinist and know very little about wind instruments. I have absolutely no clue how to properly finger a clarinet or saxophone. Yet at this point I feel I can perform a quite personal version of "Summer Time" enjoyed through the expressive capabilities of the EWI. I am primarily thinking about timing, vibrato, volume, and attack, not basic pitch and tone.

Not troubled with prior experience of traditional wind techniques for vibrato, I find much satisfaction when I achieve good pacing and width in my vibrato. This is something I work on with the violin and the ability to consider the same issues in a natural style but away from the ingrained habits of my fingers is refreshing and even helpful. The more I play, the more I also find myself considering breath; my ability to shape a note envelop with breath pacing and calculating when and where I should take a breath. Although I could have guessed in advance, I can quickly experience

that tonguing is the best way to get an aggressive attack on a note and achieve quick *stacatto*. I do find the improvisation mode on the EWI difficult to use. I usually want to put elaboration at the beginning and end of a note which is exactly when my finger is otherwise occupied toggling the improvisation setting. I have not yet gotten comfortable enough with the timbral mapping to use it very naturally but also find the mapping itself a little clumsy. It does get at the target idea, but the sound effect is a little overdone during the attack, and too subtle in the sustain.

I still have no idea how to finger the instrument playing outside the re-performance concept, but as I have not had pitch to play with during systems tests, I've ended up becoming quite good at what I can control: the expressive aspects of the EWI. My EWI expressive capability exceeds my EWI technical pitch capability, a reverse to the normal learning order. Additionally, considering that I am already able to play difficult rich music, I am not sure my interest in wind expression is sufficient to hold my attention while learning fingerings through simple songs. I do enjoy working on a song with the EWI. As mentioned above, I did not consider myself a user case. However, as someone otherwise meeting the core expected user criteria of basic comfort with the interface and intimate knowledge of the tune, I found expressive re-performance succeeds towards its intended goals. It is both motivational and expressively engaging.

Chapter 6

Conclusions

Expressive re-performance is the sum of its parts. It can be evaluated two ways: does the design implementation work, and is the design good?

6.1 Technical Performance

Considering the various subsystems, starting with the pre-processing, the results for audio extraction are excitingly good, even if the extraction processes still lacks a GUI and could benefit from further improvements to efficiency, accuracy and usability. The basis for a state-of-the-art extraction system has been created. Without better means for mathematically comparing extractions, no direct comparisons are properly possible; but extraction using PLCA and Dirichlet hyperparameters is not only far more general purpose than the present practical means using center channel isolation, it also sounds better. This is a good result.

With a 98.6% hit ratio for catching notes and a 9.1% chance of a false hit, the analyze~ based event detect does a decent job finding notes within the song. When combined with the Echo Nest analysis based event detect, which only had a hit accuracy of 82.0%, but a smaller 2.9% chance for a false hit, and the note detection system behaves admirably. The Echo Nest event detect largely solves the issue of detailed note placement, which the analyze~ based event detect fails, usually placing

a trigger within an adequate 5ms of the correct moment. There is certainly room for improvement in these results, but they just as certainly constitute an acceptable starting point for final corrections and edits through manual means.

The real-time systems also worked largely as intended. Note triggers and continuous controls were successfully and correctly parsed. The AUTimePitch˜ external worked well, providing low latency high quality time-stretched audio. The rate shift over playback worked to capture the characteristic attack without significant artifact through the rate slowdown. The visual GUI to help the user with note progression and upcoming rhythms worked technically and was indeed highly relevant to a successful user experience.

6.2   Expression through Re-performance

Unfortunately, technical successes were not matched by user satisfaction with their experiences from the system overall. Though users generally regarded the interaction as fun and stated that they would be interested in using expressive re-performance for a favorite song, few reported feeling more than minimally in control of expressive aspects other than timing, without which expressive re-performance fails to achieve one of the core goals, letting a novice experience expression.

The failure in user tests to provide an expressive experience is not yet indicative of a failure of concept. There were some incorrect design choices that significantly hampered user enjoyment of the system, most notably, difficulty playing the EWI and the inability to infinitely sustain a note. The lack of indefinite sustain was a design compromise that turned out too important to compromise on. Means to correct sustain are entirely feasible in time. The EWI, an implementation compromise, is more of a problem.

The user test experience was also not able to fully access its target audience, the excited and minimally proficient novice. The high number of professional musicians in the study helped reduce the time required to understand and explore the system, though there was little chance they would learn about expression. The inclusion of four complete novices was relevant for assessing how expressive

94

re-performance stoked the imagination and whether expression was learned. However, as they were not all interested in performance, their reactions and levels of exploration were variable.

The use of the Linkin recording of Gershwins "Summer Time was also problematic. It was not the "favorite song intended. It was a familiar enough song that users had knowledge of the tune, but not Linkins variations. These variations were particularly frustrating for experienced players, especially in combination with the sustain issues. These shortcomings do highlight the challenge of how to deal with musical pieces for which expressivity is part of the style. Obviously, if the user were playing a version that really was their favorite, they would presumably know and enjoy the original performers improvisations. Presuming the sustain was fixed, it would be interesting to see how people reacted to playing their familiar favorite semi-improvised song. Would the variations still be disliked? As expressive re-performance also includes means for improvisation, would using that feature require starting with a recorded version of the song free from improvisation? These are fairly central but still unanswered questions.

Returning to the issue of the instrument, the Wii-mote was found almost universally more enjoyable to play with. For the short term, 9 out of 10 performers preferred the Wii-mote; but for the long term, 6 out of 10 preferred the EWI. This finding suggests that users remain attached to the image of the traditional instrument, a notion supported by one user who stated, "My mind went back to the image of memories of people playing the sax, and I wanted to look like them. Responses also implied that the complexity and control-ability of the EWI were desirable, even if mastering the EWI was challenging in the short term.

But for the test environment, this complexity was definitely problematic and an impediment to expressivity. It is worth noting that internet user reviews of the EWI, completely unrelated to this thesis, comment on the difficulty in becoming accustomed to playing the EWI; so perhaps the similar frustration expressed by users in this study was somewhat predictable. Comments often remarked on both the oddity of placing vibrato in the bite and the difficultly of getting accustomed to the sensitivity of the buttons. Bite and button issues were indeed among the major interface-related critiques in user tests. One novice user remarked that it was not until the very end of an extended session using the EWI that she finally understood what the intent was; the explanation she gave was that it was only then that she was able to relax more into the song, understand the point of the

95

exercise, and start to consider the phrasing. But by that time she was exhausted by the interface and did not want to continue much longer.

The overall implication is that the target user must be invested in the instrument and should have prior experience with it. This was not originally expected as a customer requirement, but in retrospect, is entirely reasonable. Expressivity cannot be learned without some level of instrument and song knowledge, requiring commitment and practice. Much of expression is achieved through nuance, which requires fine control of a broader range within the interface. The success of the Wiimote implies that technical requirements can be alleviated somewhat, but not in their entirety, as it still only received neutral marks about its ability to convey the feeling of expressive music making.

The requirement that the user be invested in the instrument is part of the problem with the EWI. It is sufficiently difficult to use that it takes time and patience to become comfortable using it, even in the re-performance environment. Yet it is insufficiently similar to a clarinet, saxophone, oboe, or other wind instrument to relate to and interest wind players, who would presumably be the target users. Wind players were some of the most critical test users of the EWI interface. There clearly are people who play the EWI seriously, but they are few in number, and it is even more unlikely that they started with an EWI rather than another wind instrument. The Yamaha WX5, which was not used for this project, is reported to be closer to a real clarinet, with reeds and a more similar mouth piece. It is possible that it would be more successful than the EWI used here. As a tool for facilitating learning and inspiration, the EWI is too far off the mark.

What about the Fender controller? Unlike the EWI, the Fender is actually a normal instrument that can be played just like any electric guitar. It is clearly a step in the right direction. The problem with the Fender was more that it did not work entirely well. As mentioned earlier, there was little dynamic control through the plucking, as it only responded to strong strikes.

In the end, expressive re-performance will only be as strong as the interface and the interface will need to better to match the real instrument. The only way to better reflect the instrument is through the development of custom electronic versions or sensor additions, similar to the Fender, just better implemented (presumably part of the challenge with the Fender is that it is an inexpensive functional electric guitar, and presumably some concessions had to be made to achieve that.)

Expressive re-performance remains a valid interesting goal even if imperfectly implemented at present. Much of the core technology works well. There remain challenges related to interface and finding an appropriate test audience to assess whether some of the obstacles found in this round of user test—namely frustration with ties to the existing recording, the timing investment required to know the rhythmic details, discrepancies between the traditional interface and any mimic interface, and dissociation due to pitch having nothing to do with fingering—are true impediments. If they are, expressive re-performance ends up of very limited interest. If they are only minor obstacles, then it has real potential as a more generally develop-able tool.

As an accidental target user, I have found it successful as a means to separate expression from technique, and as a valuable means to break out of the mental box framed by technicality. I am an experienced musician not in need of learning about expression generally, but expression within a song is still a frequent challenge. Even though the EWI is a very different interface, many of the expressive outcomes are related. If I had an extraction of some of the classical works with which I struggle with pacing the vibrato and bow appropriately, I think the expressive re-perfomance system would be helpful. It is easier to overcome some of the habits of my violin vibrato if I can independently get used to hearing the vibrato as I want, with pacing that I have deliberately chosen. Breath, naturally constrained on the EWI but easier to control than bow, would allow me to experiment with phrasing more freely. If I know explicitly what I want and can get it in my ear, then it will be easier to get my fingers to cooperate. True, it may take time to get where I want with the expression on the EWI; but considering that I already spend hours each day just trying to get the technique right on the violin, expressive re-performance definitely has potential as a valuable tool for expressive thinking.

Chapter 7

Future Work

There is ample opportunity for future work on virtually all aspects of re-performance, both conceptual and technical. The use of expressive re-performance as a learning tool proffers an interesting range of possibilities and challenges. Additionally, continuing to build a user-friendly extraction tool will have definite value within the audio community. This section will discuss planned future work on the technical subsystems and finish with future development possibilities for expressive re-performance as a whole.

7.1   A Tool for Audio Extraction

A definite follow-on from this thesis will be the development of a MAX/MSP-based GUI implementing Smaragdis's extraction algorithm. Not only is an extraction GUI core to expressive re-performance, but its results are impressive enough that it will be valuable to the wider audio community. A reasonable set of guidelines for effective parameterization has been found, and the means for estimating optimal M do exist. Along with a variety of further algorithm implementation optimizations, work will continue, investigating how to improve initial estimates to converge to a more correct maxima faster. A GUI is also a helpful necessity. One of the cumbersome items in the present extraction process is cutting up the audio segments. This is done to group like notes together for improved performance, and to keep the problem tractable. Extraction is very computationally

demanding. Integrating the cutting process into a waveform-based GUI would be a dramatic improvement. Furthermore, porting computation out of MATLAB into C would expand its audience base.

Additionally, the quality of the mimic used for the extraction cannot be ignored. While this paper has worked with and assumed a good mimic, making one is not entirely trivial. Building some basic audio editing into the GUI would be very helpful. This could smooth out issues within the mimic, and also be used to further post-process the extraction.

7.2 Present System Enhancements

7.2.1 Event Analysis

Event analysis, in its present form, is sufficient. However, an obvious, simple upgrade would be an automated integration of the two analysis systems. An event found using both systems is highly likely to be a correct event. Highlighting the notes found by one and not the other system would increase the ease of fixing any incorrect findings. Further improvements to the event analysis might be possible through analysis improvements. The simplest improvement is to decrease the present FFT hop size for the analyze~ classification. It is presently 512 samples, which is 11ms and rather large. The large hop size is probably part of the reason the Echo Nest solution is more accurate at determining precise timing. The code was not structured to accommodate easy changes to hop size, so it was left as is. Additionally, the present analyze~ based analysis was hand scripted using knowledge of audio behavior. Using a machine-learning approach might be a better means of refining the analysis and determining whether a note change is actually taking place between any two hops. Generally, apart from an automated merge and further verification testing of the editing GUI, the event analysis already performs satisfactorily for use.

7.2.2 Time-stretch and Synthesis

Unlike with the event analysis, it is imperative to switch the time-stretch algorithm. The Results discussion suggested two possible options: looping the end of the sample, and switching to Rub-

berband. Learning an application programming interface (API) and building framework to use it are not trivial challenges, but, provided Rubberband works in real-time, as advertised, maintaining audio quality similar to what it achieves when pre-rendering stretched audio, it should be entirely feasible and worthwhile.

The option of looping the end of a segment is also worth investigating. Automated cross-fade and looping have already been built into AUTimePitch. The cross-fade is key to eliminating phase issues that might otherwise complicate choosing a loop point. Without cross-fade, the loop would probably pop. There are other complications that could arise with finding an appropriate loop point; for example, it becomes difficult if the note is very short, or if the pitch varies within the note through excessive performer vibrato.

One benefit of the loop-based approach is that it is easy to imagine instances when a trigger might control multiple notes. Grace notes have already been identified as annoying to play. An extremely fast run might also be unwieldy and combined into one event as a result. Using the sliding rate time-stretch will result in the beginning of the run playing at full speed and the end dramatically slowing down. This would rarely be desirable. With looping, only the last note need be held so that the rest of the run sounds as expected. Then again, the run would probably need to be rate-adjusted to match the user's tempo. If the level of intelligence required to rate-adjust the run existed in the time-stretch control system, then the system could probably also handle infinite stretch for only the last note in the run. Though each approach has its advantages and disadvantages, it is clear one or the other will need to be implemented in any future re-performance system.

7.2.3   System Flexibility and Expandability

This paper has discussed expressive re-performance in the context of extracted audio, but there is no inherent reason for that source limitation. If there are master tracks available of the audio, the source audio can simply replace the extraction. Everything else proceeds the same from event analysis on-ward. Similarly, if a MIDI mimic was used for the extraction, the event analysis should no longer be required. If the user wanted to do away with the use of recorded audio all-together, it would be reasonably simple to build in means to use the traditional options of a pre-existing score

101

to trigger synthesized audio. Adding MIDI score and synth style tapper capability would merely require skipping the time-stretch functionality and passing pitch events to a synth engine. There are plenty of synths existing in Ableton Live.

Additionally, as has been already been discussed and demonstrated through the EWI and Wii-mote, the implementation and effectiveness of expressive re-performance are dramatically impacted by the instrumental interface. It would be valuable to integrate new interfaces into expressive re-performance. Anything MIDI, OSC, or serial capable would be adaptable and mappable. The interfaces selected were chosen in part due to a lack of options, implying that additional interfaces would have to be predominately custom-built. Instrument interfaces are a huge area for potential complex exploration. Options include adding sensors to existing, traditional instruments, and creating new electronic versions. With the addition of a few systems control buttons, any sensing system that does well at capturing the instrument performance normally would likely be appropriate within re-performance. Each instrument would require its own parsing and mapping.

7.3   Tool For Learning

One of the most exciting prospects for expressive re-performance is its us as a learning tool. Expressive re-performance presented as a fantasy enabler is more of an educational motivator than teaching tool. For the other use case, the semi-experienced musician wanting to concentrate on expression, expressive re-performance is a practice tool. Approaching expression early on is unconventional, in part because technique has always preceded expression.

Expressive re-performance for the novice can be expanded beyond fantasy motivation. If the interface is sufficiently close to the real thing, technical expression skills learned through the re-performance interface should be transferable. Further, pitch, which has been removed, can re-enter into the learning process. For example, as with RockBand, the performer might have to correctly finger a note for it to play. Difficult sections, like a fast run, are simplified to a single trigger. Gradually adding simplified fingering relations, like requiring fingering in the correct direction, up or down, or automatically correcting accidentals are all possible means to encourage the development of correct fingering. With a violin, the requirement could be that fingering within a roughly correct

zone on the finger board is what is necessary to play the note, thus reducing the onerous demand of intonation. Further, what about expanding to an ensemble? This is an extra challenge for extraction but would certainly be more fun. Many of these possibilities break away from expressive re-performance and into technical re-performance. Yet the novelty of using expressive real world instrument interfaces to activate real audio remains.

Appendix A

User Performance Instructions

Instructions for Play!

Accompaniment can be turned on and reset through the space bar on the computer

Wii:
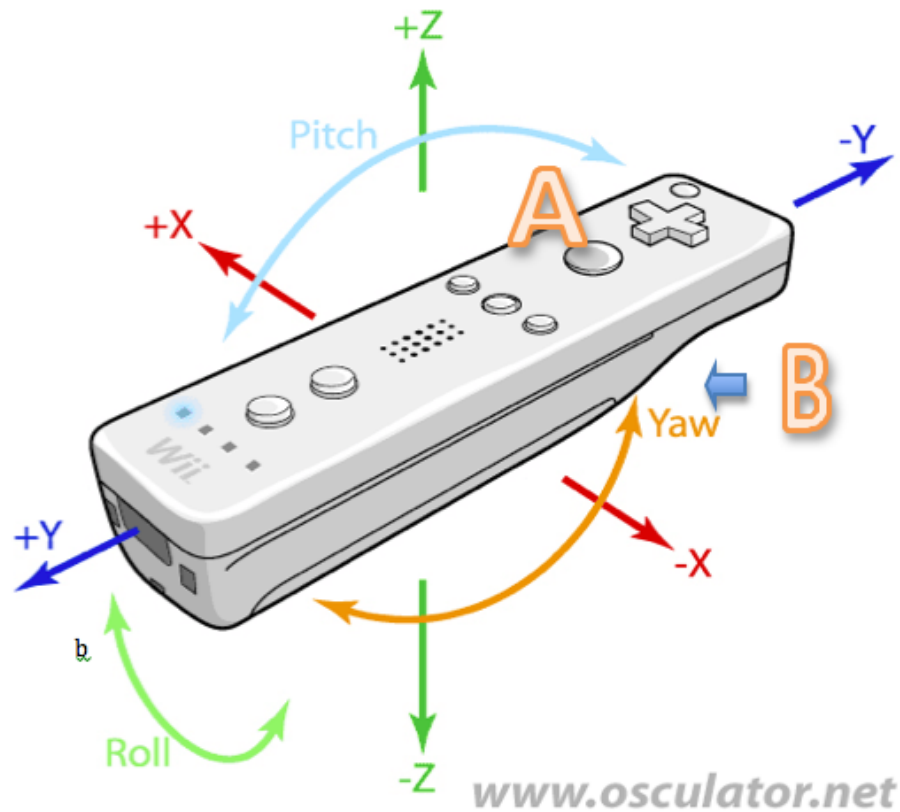
Note Advance- A button

Note Repeat- hold B button and press A again.

Trills/Improv- up-down-left-right

Volume – Roll

Vibrato/pitch change- Wobble

Tone – Pitch

Restart- Home

Note: Explore continue controls as the instructions below are fairly lose.

EWI:

Make sure to touch one of the two ground plates with your thumbs. There are two modes for note advance selectable through the Live/MAX interface.
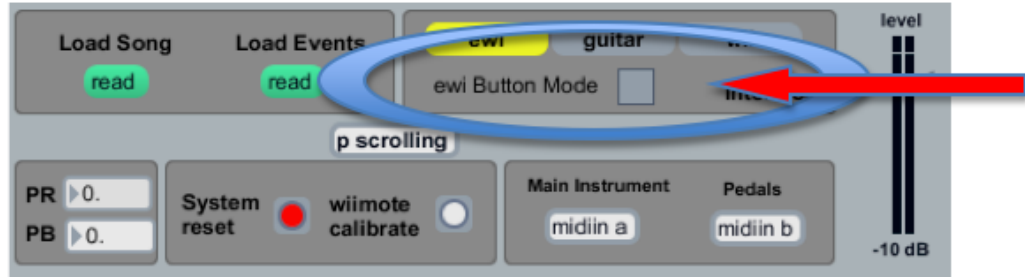
With EWI button mode off (as pictured), just like the real instrument, any change in fingering or breath will trigger a new note.

With EWI button mode on (orange), there are two possible holds where you act like you are pressing a button in order to advance a note. The trigger note is in red. The two holds are:
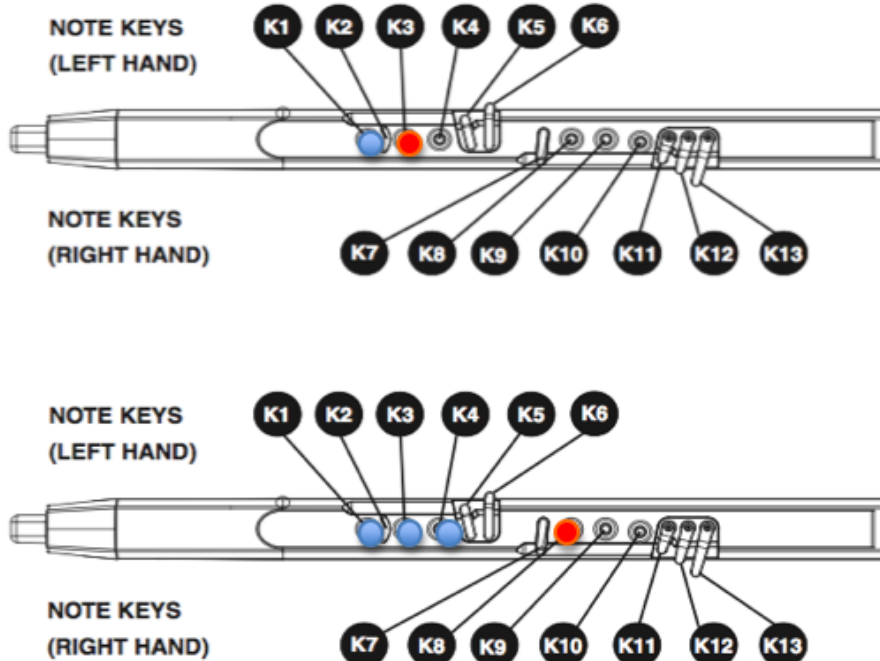
Note Repeat - hold the repeat button and press the "advance button".

Trills/Improv - toggle the improv button and use keys. A new breath will advance to the next note.

Volume – Blow!

Vibrato - light bite of the mouthpiece

Tone – the strip to the right of rollers

Restart – remove all fingers and

Repeat ⤴ ⤵ Improv

Tone Bar (pressure)

Must Contact Ground Plates
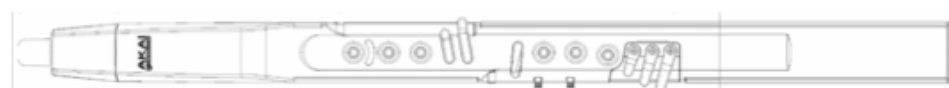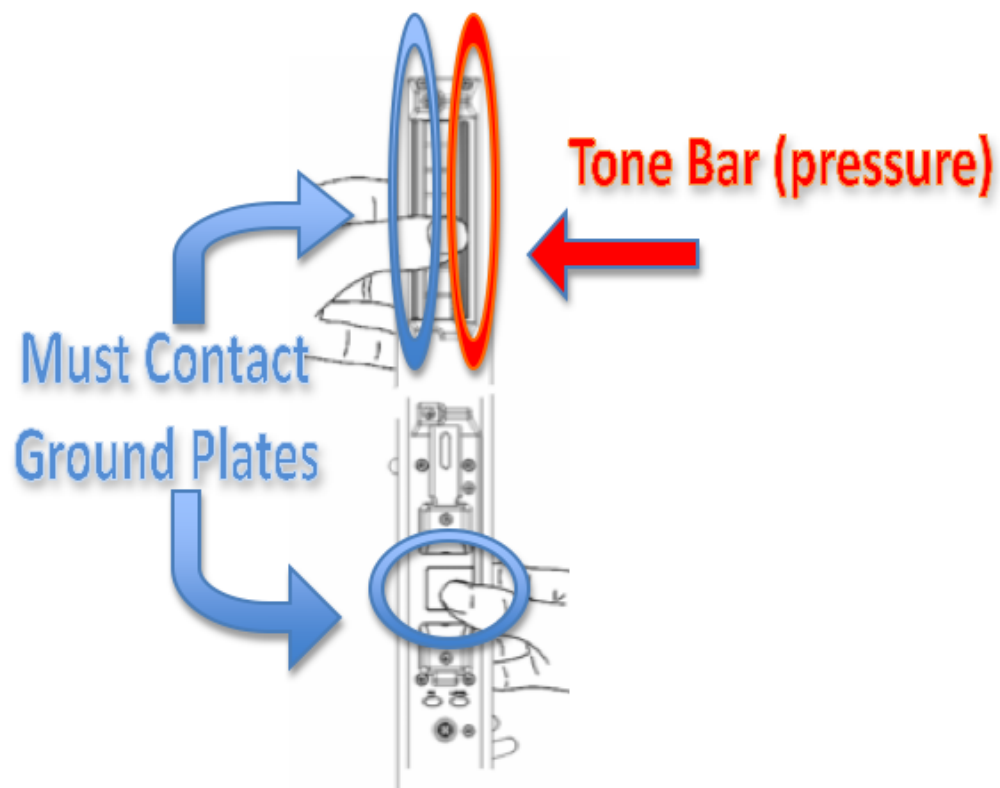
Appendix B

Sample Real-Time Evaluation User Survey

Table B.1: The Real-Time Interaction User Questionnaire (pg.1)

| 23 Questions, 4 Pages. Please answer. Thanks!			
Name:			
	(results will be made anonymous)		
What is your basic musical background?			
Have you ever played saxophone or clarinet?			
	Clarinet - years		Saxophone - years
After playing with the EWI (electronic wind controller) please answer the following questions:			
1) How easy was it to understand how to use the instrument?			
	(not at all easy) 1 2 3 4 5 6 7 (crazy easy!):		
	comments:		
2) How easy was it to control the music the way you wanted?			
	(not at all easy) 1 2 3 4 5 6 7 (crazy easy!):		
	comments:		
3) Did you feel like you were able to use the system in an expressive manner?			
	(not at all expressive) 1 2 3 4 5 6 7 (very expressive):		
	comments:		
4) Would you want to spend time using the system in order to use the EWI better?			
	(not at all) 1 2 3 4 5 6 7 (can I play more?)		
	comments:		
5) Was the interaction fun?			
	(really dull) 1 2 3 4 5 6 7 (woot!)		
	comments:		

Table B.2: The Real-Time Interaction User Questionnaire (pg.2)

Now Same set of questions but for the Wiimote.			
6) How easy was it to understand how to use the instrument?			
(not at all easy) 1 2 3 4 5 6 7 (crazy easy!)			
comments:			
7) How easy was it to control the music the way you wanted?			
(not at all easy) 1 2 3 4 5 6 7 (crazy easy!)			
comments:			
8) Did you feel like you were able to use the system in an expressive manner?			
(not at all expressive) 1 2 3 4 5 6 7 (very expressive)			
comments:			
9) Would you want to spend time using the system in order to use the Wii better?			
(not at all) 1 2 3 4 5 6 7 (can I play more?)			
comments:			
10) Was the interaction fun?			
(really dull) 1 2 3 4 5 6 7 (woot!)			
comments:			
Now some overall questions including comparison of the the two instrument controllers:			
11) How did you find the sound quality?			
(complete tosh) 1 2 3 4 5 6 7 (sounds great)			
comments:			
12) How did the sound measure up to a clarinet sound?			
(that was a clarinet?) 1 2 3 4 5 6 7 (just like one)			
comments:			
comments:			
13) Did you enjoy playing with the accompaniment?			
(prefer solo) 1 2 3 4 5 6 7 (prefer backing)			
comments:			

Table B.3: The Real-Time Interaction User Questionnaire (pg.3)

14) Which interface was easier to use?			
(EWI) 1 2 3 4 5 6 7 (Wiimote)			
comments:			
15) Which interface was more expressive?			
(EWI) 1 2 3 4 5 6 7 (Wiimote)			
comments:			
16) Which interface was more fun?			
(EWI) 1 2 3 4 5 6 7 (Wiimote)			
comments:			
17) Which interface did you prefer to spend time with?			
(EWI) 1 2 3 4 5 6 7 (Wiimote)			
comments:			
18) Would you spend time using the system trying to shape the given song?			
(not interested) 1 2 3 4 5 6 7 (very interested)			
comments:			
19) Would you spend time using the system trying to shape a song of your choice?			
(not interested) 1 2 3 4 5 6 7 (very interested)			
comments:			
20) Please provide any general comments or thoughts about the system overall (i.e. sound and accompaniment). Are there any improvements that come to mind?			

Table B.4: The Real-Time Interaction User Questionnaire (pg.4)

21) Do you have comments on the instrument control interfaces? Any improvements how to make them more intuitive or expressive?
22) What did you learn about the music? Did you think at all about musical phrasing?
23) Anything else?

Appendix C

MAX/MSP Code

C.1 Event Analysis & Detection

C.1.1 Analysis Related MAX/MSP Patches

Figure C-1: Patch *EventDetect.maxpat* for finding note starts using Analyzer˜ .
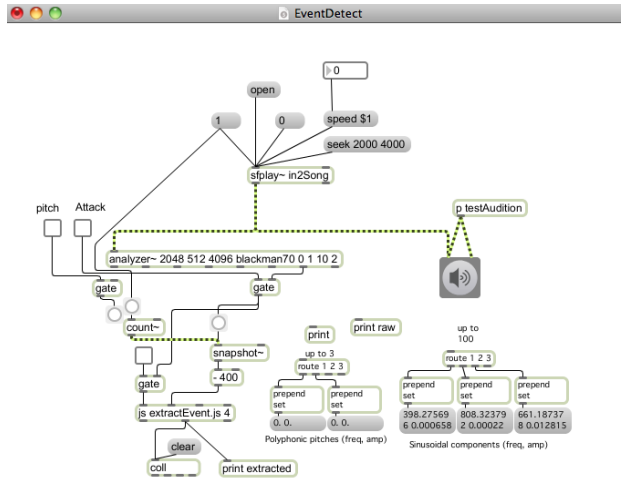

Figure C-2: Patch *En_EventDetect.maxpat* for finding note starts using Echo Nest analysis.
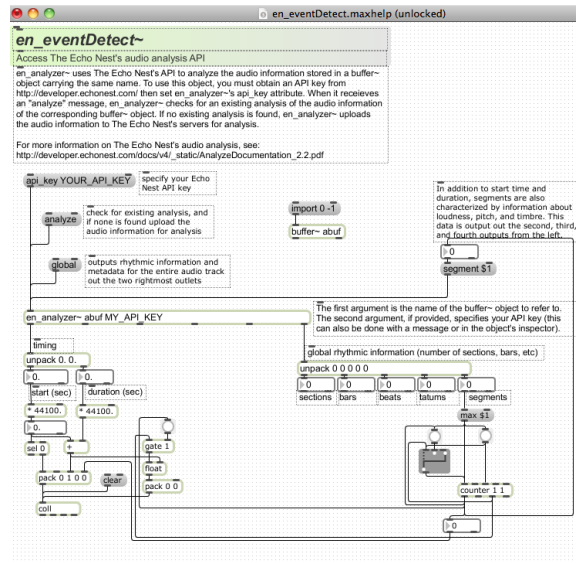
Figure C-3: Patch *chooseEvent.maxpat* for editing and merging event analysis. It is shown with no audio or event files loaded.
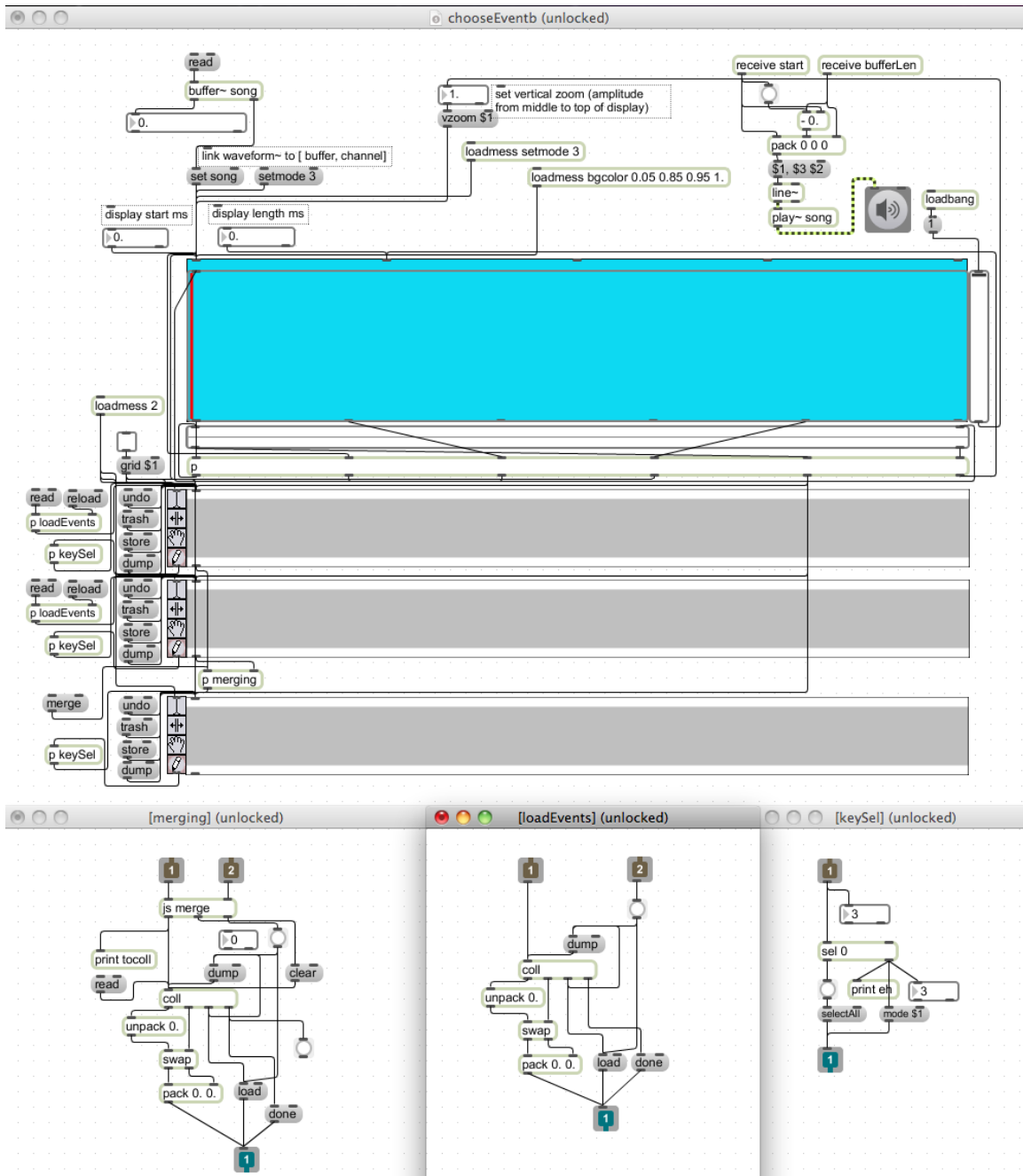
C.1.2 Analysis Related MAX/MSP Java Script

ExtractEvent.js

ExtractEvent.js used inside *eventDetect.maxpat* to discover note start locations.

```
 1  autowatch = 1;
    // arguments −
    // 1 − difference to look for in volume change (midi right now)
    // 2 − min note length
    // 3 − cut off volume beneath which a sound discarded

    // first inlet is list of changing midi
    // second inlet is sample count
    inlets = 4;
    // Other inlets were for when using db rather than MIDI stats− Old code deleted
11  // first outlet is a bang
    // second is the sample point, note and volume
    outlets = 3;

    lastNote = 0;
    lastNoteStart = 0;
    sample = 0;
    verified = true;
    volVerified = true;
    unverNoteStart = 0;
21  unverNote = 0;
    unverLVol = 0;

    var noteDB = 0;

    var attackDif = 10.0;
    var minNoteLength = 3400; // just under 100ms
    var volCutOff = 65;  // cutoff at 20DB  or 24 for less extras

    function list(a){
31      if (inlet == 0) {
            if (arguments.length < 2) {
                post("Insufficient arguments to ExtractEvent");
                return;
            }
            else {
                noteVal = Math.round(arguments[0]);
                noteVol = arguments[1];
                if (noteVol < volCutOff) {
                  noteVal = 0;
41                noteVol = 0;
              }
                post("Sample", sample, "Note",noteVal,"Vol",noteVol, "InDb:", noteDB);

                post();
      //           this will not catch the momentary errors
                if ((verified) && (volVerified)) {
                  if ((sample − lastNoteStart) > minNoteLength) {
                    // if sample below certain volume, unlikely to be reliable and is
                        probably supposed to be silence
                    if (noteVal != lastNote) {
51  //                    if (noteVal != 0) {    // not a reattack if a zero
                            verified = false;
                            unverNoteStart = sample;
                            unverNote = noteVal;
                        post("verifying", unverNote);
                        post();
      //                    }
                    }
```

```
                       else if ((noteVol >= (lastMidiVol[0] + attackDif)) && (noteVol
                          >= (lastMidiVol[1] + attackDif) )){
                    //    if (noteVal != 0) {     //not a reattack if a zero
61                           volVerified = false;
                             unverNoteStart = sample;
                             unverNote = noteVal;
                             unverLVol = lastMidiVol[0]; //oldest first
                         post("verifying", unverNote, noteVol);
                         post();
                    //    }

                    }
                }
71          }
             else if (!verified) {
                 if (noteVal == unverNote) {
                   lastNote = noteVal;
                   lastNoteStart = unverNoteStart;
                   verified = true;
      //           if (lastNote != 0) {
                   outlet(2, lastNoteStart);
                         outlet(1, lastNoteStart, noteVal, noteVol);
                         outlet(0, "bang");
81
     //                 }
                 }
                 else if (noteVal == lastNote) { //guess temporary anomaly
                   //no change.  Either error blip or check for re-attack
                   if (noteVol > (lastMidiVol[0] + attackDif)) {   //two samples ago
                        vol
                        lastNoteStart = sample;   //use second because volume change
                             here... PROBLEM
      //               if (noteVal != 0) {
                             outlet(0, "bang");
                             outlet(1, lastNoteStart, noteVal, noteVol);
91 //                 }
                   }
                   verified = true;
                 }
                 else { // changed notes but there was a blip inbetween maybe.  Check
                        for constancy
                   unverNote = noteVal;
                   unverNoteStart = sample;   //May lose volume info, but if notes did
                        change, should be caught.
                   //leave unverified.
                 }
               }
101          else { // vol change not verified
               //the way volume is working, to trigger, the diff must have
                    sufficient differential over both the last two notes.
               //if this is true, we confirm the next note is too and that it is
                    not decreasing (attack should increase? will see if this fails
                    but hopefully won't)
               if ((noteVol > lastMidiVol[1]) && (noteVol > (lastMidiVol[0] +
                   attackDif)) && (noteVol > (unverLVol + attackDif))) {
                   lastNoteStart = unverNoteStart;
                   lastNote = noteVal;
     //               if (noteVal != 0) {
                         outlet(0, "bang");
                         outlet(1, lastNoteStart, noteVal, noteVol);
     //               }
111          }
               volVerified = true;   //no extra chances/confirmations here
             }
             lastMidiVol.shift();
```

```
            lastMidiVol.push(noteVol);   //this should always update
            }
         }
      }

      function msg_int(ina){
121      if (inlet == 1){
            sample = ina;
            if (sample < lastNoteStart)
               lastNoteStart = 0;   //sample clock restarted.
         }
         else
            post("Integer Wrong Inlet.");
      }

      function main() {
131      switch(jsarguments.length) {
            case 4 : volCutOff = jsarguments[3];
            case 3 : minNoteLen = jsarguments[2];
            case 2 : attackDif = jsarguments[1];
         }
      }

      main();
```

C.2 Real-Time Expressive Re-Performance

### C.2.1 Real-Time MAX/MSP Patches

Figure C-4: Patch *reperformance.maxpat* centrally controling real-time experience.
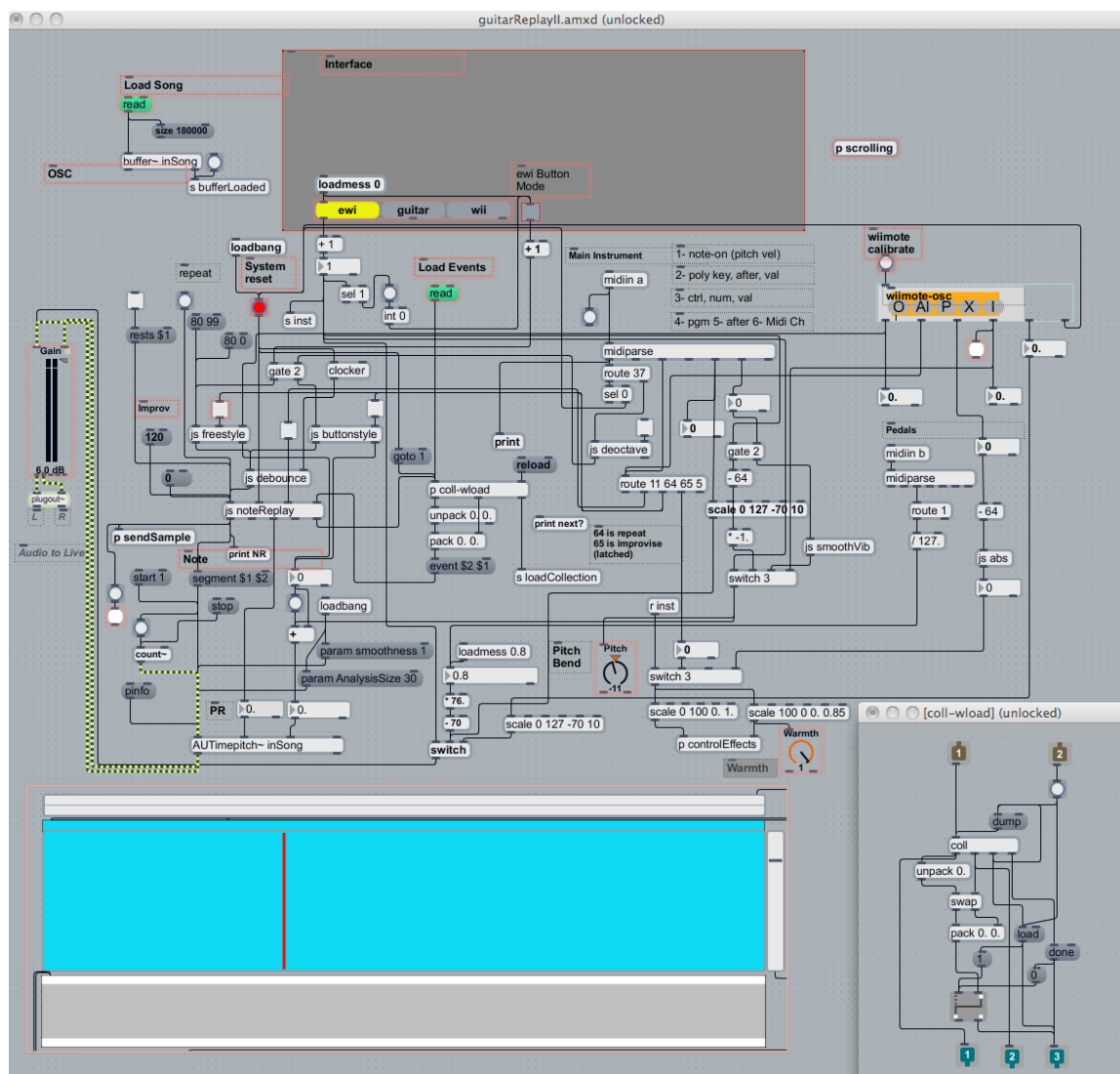
Figure C-5: Patch *wiimoteosc* for wiimote parsing and interpretation. Based off *aka.wiiremote* by Masayuki Akamatsu. [17]
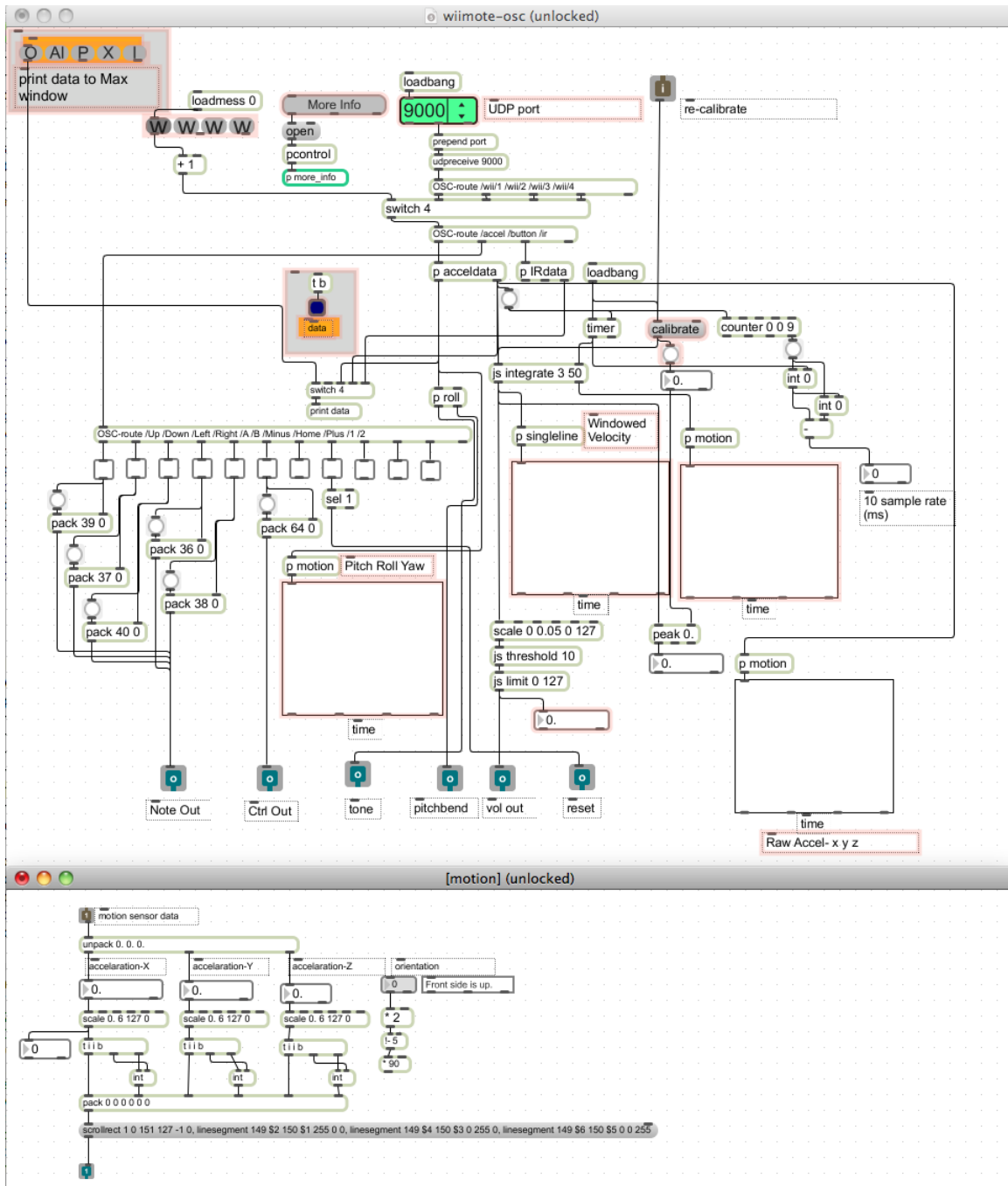
Figure C-6: Patch *scrollNotes* to provide user visuals tracking event progress.
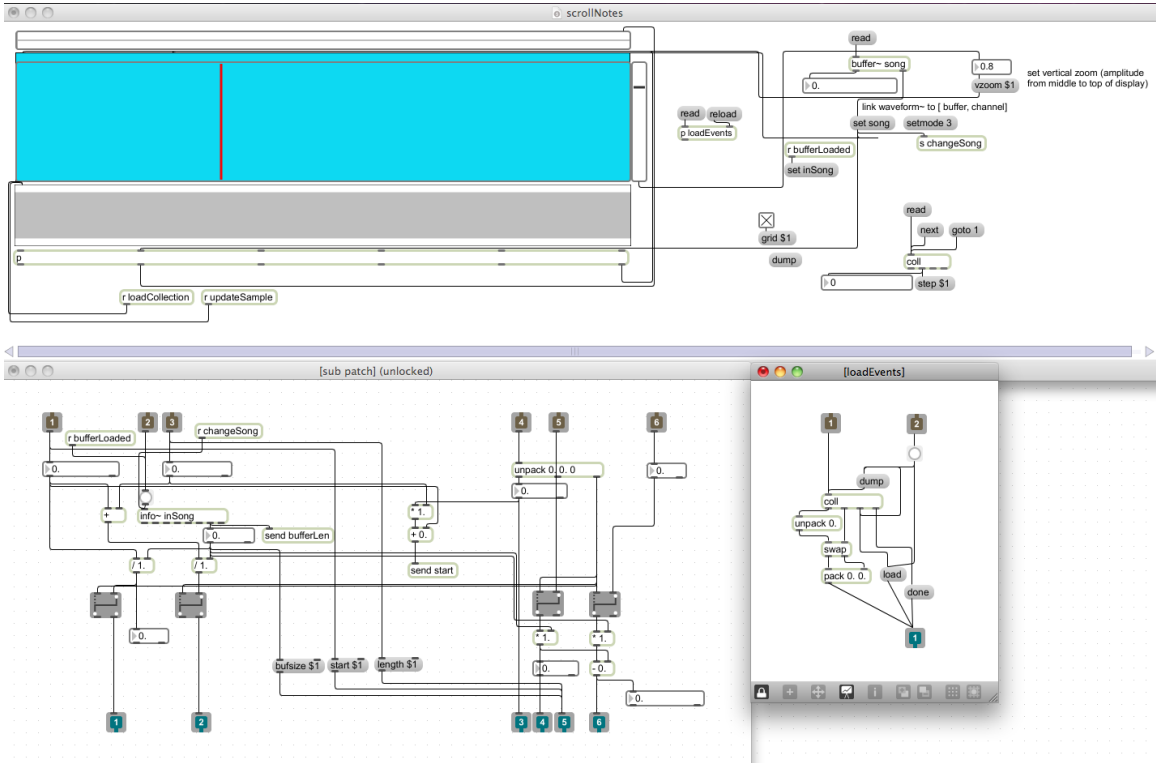
Figure C-7: Sub-patch *controlEffects* for controlling Ableton Live effects.
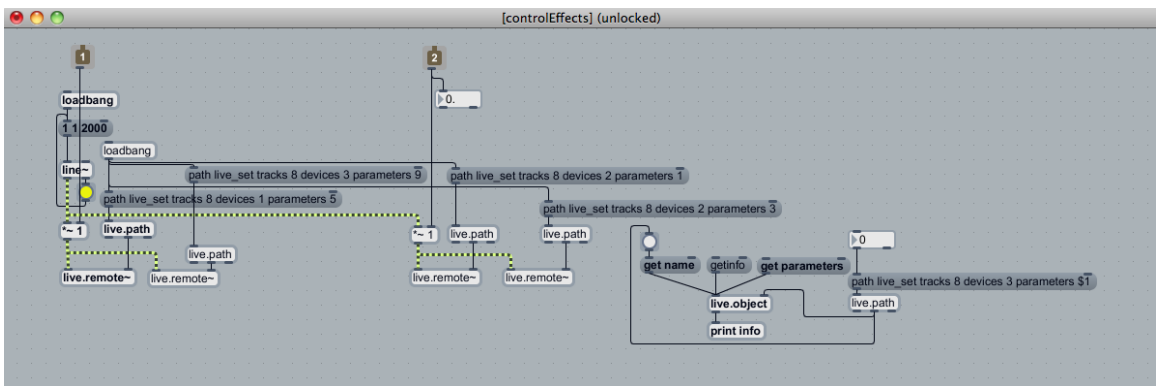
C.2.2 Real-Time MAX/MSP Java Script

noteReplay.js

noteReplay.js controls whether and which trigger to play and then adjusts playback rate appropriate

to note length within textitreperformance.maxpat.

```
autowatch = 1;
2
// arguments −

// first inlet is note event (integer)
// second inlet is bang for reset
// third −− repeat note
// fifth −− actual note control info
inlets = 4;

// first outlet is the start time (in ms)
12 // second length
// third is the fractional play rate
// fourth triggers note stop
// fifth triggers collection
// no output until two notes in.
outlets = 5;

var playRatio = .7;
var minRate = .125;

22 var iter = 0;
// var per = [.11, .23, .22, .22, .22];
// var rat = [1, .9, .6, .4, .3];
var per = [.1, .15, .1, .1, .55];
var rat = [1, .9, .6, .4, .25]; // 2.8 times note length

var noteLength;
var totLength;

var playRests = false;
32 var repeat = false;

var playRate;
var tsk = new Task(updateRate, this); // our main task

var noteSchedule;

var sampleRate = 44100;
var inEvents = [];
lastOut = [0,0,0,0];
42
var noteOn = false;
var noteAutoAdvanced = false;
var autoRestInserted = false;

// Debouncing variables
// var lastNoteTime = 0;
// var currNoteTime;
// var debounceTime = 100;

52 function bang(){
    if (inlet == 0) {
            outlet(2, lastOut[2]);
            outlet(1, lastOut[1]);
            outlet(0, lastOut[0], lastOut[3]);
    }
    if (inlet == 1){
      playRate = 1;
```

```
            inEvents = [];
            lastOut = [0,0,0,0];
 62         post();
            tsk.cancel();
            noteOn = false;
            noteAutoAdvanced = false;
            lastNoteTime = 0;
            // pre-load one sample
            outlet(4, "next");
        }
    }

 72 function setRepeat(inVal){
        repeat = (inVal != 0);
    }

    function hold(ina){
        if ((inlet == 0) && (arguments.length == 1)){
            setRepeat(arguments[0]);
        }
        else post("Invalid Use of \'hold\'.");
    }
 82
    function rests(inb){
      if ((inlet == 0) && (arguments.length == 1)){
        playRests = (arguments[0] == 1);
      }
      else post("Invalid input");
    }

    function startAudio(instart, inend) {
      noteLength = (inend - instart)*1000/sampleRate; //in ms
 92    playRate = 1;   // reset the play rate to normal
    //          tsk.interval = noteLength * ratio;  //right now this stays the
        same
      iter = 0;
        tsk.interval = noteLength*per[iter]/playRate;  //right now this stays the
            same

      totLength = tsk.interval;

      lastOut[0] = (instart/sampleRate);
        lastOut[1] = noteLength;
        lastOut[2] = playRate;
102      lastOut[3] = inend/sampleRate;

    //     post("NR Start:", lastOut[0]*44.1, lastOut[0], "ms End :", inend/1000, "
       Length:", lastOut[1]*44.1, "ms");//, "debug0", inTimes[0], "debug1", inTimes
       [1]);
      post();

      outlet(2, lastOut[2]);      //send play rate
        outlet(1, lastOut[1]);      //send note length
        outlet(0, lastOut[0], lastOut[3]);     //send start time
        tsk.repeat();               //kick off the time count
    }
112
    // Getting a note I know the next one so I can predict here.
    //1) a new note comes in. triggers collection. If it is a non-zero volume,
       play it back. Otherwise find next non zero note (should be next)
    //1a) next a note off comes in. triggers collection. If next note is note off,
        step to that. Otherwise wait till next note on
    //1b) or a new note on comes in- return to 1
    //1c) person holds - note lasts for *2, get next note, if it is noteoff, play
       that. If not, pause.
      //-- this leaves a state where the present note state is being played.

    function event(vals) {
      if ((inlet == 3) && (arguments.length == 2)) {
```

```
122          post("Event_Received", arguments[0], "_", arguments[1]);
         post();
         if (inEvents.push(arguments) > 1) {
           if (noteOn) {
              var immEvent = inEvents.shift();  //get info to be played.  Guarantees
                 note step
    //               post("Vals: ", immEvent[0], ", ", immEvent[1], " ");
              if (immEvent[1]) {               //verify it is a note to play
                var nextEvent = inEvents[0];
                startAudio(immEvent[0], nextEvent[0]);
              }
132           else{
                post("skipping_rest?");
                outlet(4, "next");           //go to next note as no playing silence
              }
    //         autoRestInserted = false;       //not relevant for a note-on so clear
         flag
           } // note off
           else{
              var immEvent = inEvents[0];    //don't shift yet
              if (immEvent[1]){
                noteAutoAdvanced = true;  // a pause has been inserted so wait
142             outlet(3, "bang");          // stop playback
                tsk.cancel();
    //            autoRestInserted = false;    // user explicitly inserted un-scored
         rest, handled
              }
              else{
                // play note off
                inEvents.shift();       // step note
                if (playRests){
                  if (!autoRestInserted) { // only play potential rest decay if not
                       already resting
                     startAudio(immEvent[0], inEvents[0][0]);
152               }
                }
                else {
                  outlet(3, "bang");        // stop playback
                  tsk.cancel();
                }
              }
            }
            autoRestInserted = false;      // clear any rest flag
         }
162    }
      else {
         post("Invalid_Input");
      }
    }

    function msg_int(ina) {
       if (inlet == 0) {
           // debounce done externally using js debounce
    //        if ((currNoteTime - lastNoteTime) > debounceTime) {
172            if (repeat) {
                  startAudio(lastOut[0]*sampleRate, lastOut[3]*sampleRate);
              }
              else {
                  if (ina) noteOn = true;
                  else noteOn = false;
                  post("Note_in:", noteOn);
                  post()

                  if (noteAutoAdvanced) { // note has already advanced so set
                       flags & send
```

126

```
182                         noteAutoAdvanced = false;
                          if (noteOn) {
                              var immEvent = inEvents.shift();   //get info to be
                                  played.   Guarantees note step
                              if (immEvent[1]) {               //verify it is a note to
                                  play
                                  var nextEvent = inEvents[0];
                                  startAudio(immEvent[0], nextEvent[0]);
                              }
                              else{
                                  outlet(4, "next");           //go to next note as no
                                      playing silence
                              }
192                         }
                          else {
                              post("Auto-advanced from a note-off.  When would this
                                  happen?");
                              post();
                          }
                      } // otherwise fetch next data set
                      else {
                          if (noteOn){
                              outlet(4, "next");
      //                        lastNoteTime = currNoteTime;
202                         }
                          else if (playRests) {
                              outlet(4, "next");
                          }
                      }
                  }
      //        } if debounce
          }
          else if (inlet == 2) {
              setRepeat(ina);
212     }
          else post("Wrong Inlet.");
      }

      //function msg_float(ina){
      //      if (inlet == 3) {
      //      currNoteTime = ina;
      //      }
      //      else post("Wrong Inlet.");
      //}
222
      function updateRate(){
          iter++;
          if (iter < rat.length) {
              playRate = rat[iter];
              arguments.callee.task.interval = noteLength*per[iter]/playRate;
              outlet(2, playRate);

              totLength += arguments.callee.task.interval;
          }
232 //   if (playRate > minRate) {
      //      playRate = playRate*ratio;
      //      outlet(2, playRate);
      //   }
          else {
              autoRestInserted = true;
              outlet(3, "bang"); // stop playback
              arguments.callee.task.cancel();
      //      post("Note Lasted", totLength);
      //      post();
242   }
```

```
        // I should program this so if you miss the note, it is totally skipped.Or
            make an option.)
    }

    function main() {
        switch(jsarguments.length) {
//          case 4 : volCutOff = jsarguments[3];
            case 3 : minRate = jsarguments[2];
            case 2 : ratio = jsarguments[1];
        }
252 }

    main();
```

buttonstyle.js

buttonstyle.js simplifies button press to implement "button mode". Also determines improvisational

outcomes when improv mode on. Shares similar code to *freestyle.js* also in *reperformance.maxpat*.

```
    autowatch = 1;
    //arguments -

    // first inlet is note event (integer)
 6  // second inlet is bang for reset
    inlets = 2;

    // note out
    outlets = 2;

    var base = 0;
    var improv = false;
    var last = {note : 4, vol : 0};
    var noteOn = false;
16
    // order is new note on followed by old note off so two same notes in
    // order (followed by notes off)

    function msg_int(ina){
        if (inlet == 1) {
            if (ina) {
                improv = true;
                base = 0;
            }
26          else improv = false;
        }

    }

    function allNotesOff(){
        base = 0;
        noteOn = false;
        //update outlets
        outlet(1, 0);
36      outlet(0, 0);   // pass on vol as note change event
        post("Notes_off!");
    }

    function calculateCents(inBase, inDelt){
        var centsDif = 100*(inDelt - inBase);
        return centsDif;
    }
```

128

```
     function determineImprov(inNote, inVol){
46       if (base){
             if (inVol) {
                 last.note = inNote;
                 last.vol = inVol;
                 noteOn = true;   // if there is a baseNote this should be true;

                 outlet(1, calculateCents(base, inNote));
             }
             else if (last.note == inNote) allNotesOff();   // base is reset to 0
         }
56       // gots to find base
         else {
             post("Finding base");
             if (noteOn) {
                 if (inVol) {
                     base = last.note;
                     // first improv note;
                     outlet(1, calculateCents(base, inNote));
                     last.note = inNote;
                     last.vol = inVol;
66                   noteOn = true;
                 }
                 else { // are we turning all notes off?
                     if (inNote == last.note) allNotesOff();
                 }
             }
             else{
                 if (inVol) {
                     base = inNote;
                     // trigger note
76                   outlet(1, 0);
                     outlet(0, inVol);
                     // update note info;
                     last.note = inNote;
                     last.vol = inVol;
                     noteOn = true;

                 }
                 else ("Unexpected: Last Note is off followed by new note off");
             }
             post("Base", base);
86           post();
         }
     }

  function list(a){
      if (arguments.length == 2) {
          var note = arguments[0];
          var vol = arguments[1];

          if (improv) {
96            determineImprov(note, vol);

              if (false) {
                  if (!vol) {
                      if ((last.vol == 0) || ((note == base) && (last.note == base
                          )) ) {
                          noteOn = false;
                      }
                  }
                  else {
                      var centsDif = 100*(note - base);
106                   outlet(1, centsDif);

                      if (!noteOn) {
```

129

```
                           outlet(0, vol); // next note it.
                           noteOn = true;
                       }
                   }
               }
           }
116        else { // passing through all button info
               if ((note%12 == 9) || (note%12 == 5) || (note%12 == 6)) {
                   outlet(1, 0);
                   outlet(0, vol);
                   noteOn = vol;
               }

           last.note = note;
           last.vol = vol;
126        }

       }
       else post("Invalid_use_of_'button'._ _Send_note_and_vol.");
   }

   function main() {
   }
136 main();
```

integrate.js

integrate.js performs windowed integration on Wiimote acceleration data within the *Wiimote-Osc*

patch.

```
   autowatch = 1;

   // arguments −
 4
   // first inlet is note event (integer)
   // second inlet is bang for reset
   // third −− repeat note
   // fifth −− actual note control info
   inlets = 2;

   // first outlet is the start time (in ms)
   // second length
   // third is the fractional play rate
14 // fourth triggers note stop
   // fifth triggers collection
   // no output until two notes in.
   outlets = 2;

   var numDerivatives = 1;
   var window = 25;   //window size for finding deriviative 3 data points per ms
   var windowSum;
   var values;
   var time;
24 var rawIn;
   var calibrations;

   function calibrate(ina){
       var i;
```

```
        post("length_rawIn", rawIn.length);
        post();
        for (i = 0; i < rawIn.length; i ++){
            calibrations[i] = rawIn[i];   //otherwise 0
            post(calibrations[i]);
34          windowSum[i] = 0;
            values[i] = [];
        }
    }

    function list(ina) {
      if ((inlet == 0) && (arguments.length == numDerivatives)) {
            var i;
            var sum = 0;
            var dif = new Array(numDerivatives);
44          var old = {data: 0, time: 0};

            for (i = 0; i < numDerivatives; i++){
                var newVal = {data : 0, time : 0};
                rawIn[i] = arguments[i];
                newVal.data = arguments[i] - calibrations[i];
                newVal.time = time;
                if (newVal.data < 0.0) newVal.data = newVal.data * -1;   //(take abs
                    val)
                values[i].push(newVal);
                windowSum[i] = windowSum[i] + newVal.data;
54              if (values[i].length >= (window)) {
                    old = values[i].shift();
                    dif[i] = (newVal.data - old.data)/(newVal.time - old.time)*
                        1000;
                    sum = sum + windowSum[i];
                    windowSum[i] = windowSum[i] - old.data;
    //                 post(i, " : ", newVal.data);

    //                 post(i, " : ", newVal.data, " : ", old.data, " : ", windowSum[
        i], " , ", dif[i], " ");
    //                 post();

64              }

            }

    //          post();
            outlet(1, windowSum);
            if (sum == 0) outlet(0,0);
            else outlet(0, sum/(window*numDerivatives));

    }
74  else {
        post("Invalid_Input");
    }
    }

    function msg_int(ina){
        if (inlet == 1) {
        time = ina;
    //          post("time", time);
        }
84      else post("Wrong_Inlet.");
    }

    function main() {
        switch(jsarguments.length) {
    //          case 4 : volCutOff = jsarguments[3];
            case 3 : window = jsarguments[2];
            case 2 : numDerivatives = jsarguments[1];
        }
```

131

```
        values = new Array(numDerivatives);
        windowSum = new Array(numDerivatives);
        calibrations = new Array(numDerivatives);
        rawIn = new Array(numDerivatives);
        var i;
        for (i = 0; i < numDerivatives; i++) {
            values[i] = new Array();
            windowSum[i] = 0;
            calibrations[i] = 0;
            rawIn[i] = 0;
        }
    }

    main();
```

Appendix D

AUTimePitch MSP External

D.1 AUTimePitch Help Example
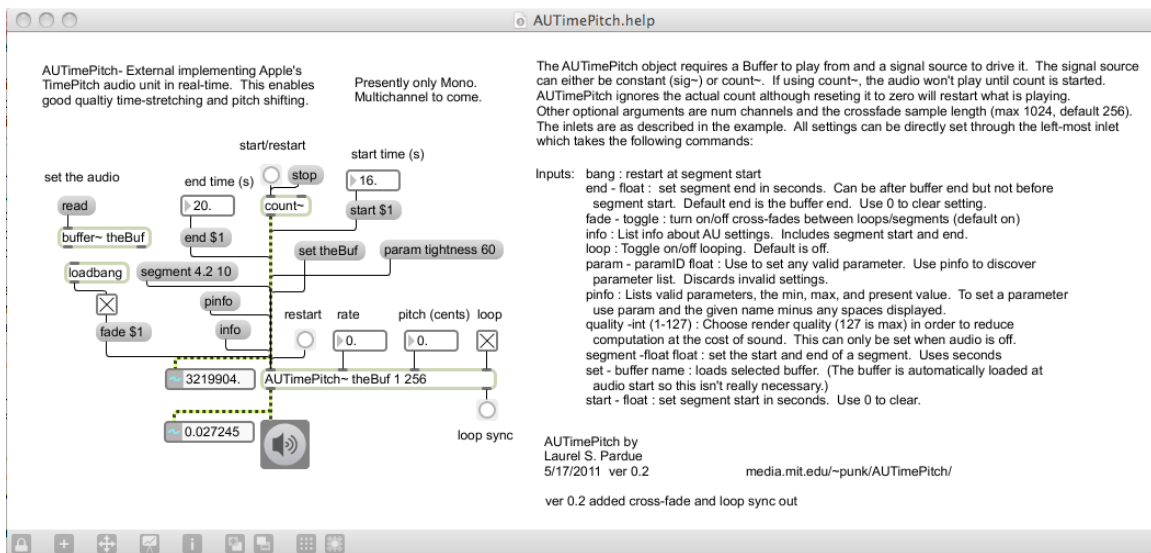
Figure D-1: The MSP help file describing the AUTimePitch external.

D.2 AUTimePitch Code

D.2.1 AUConvert˜.h

```
/*
 *   AUConver ˜. h
 *   DiscoverComponents
 *
 *   Created  by  static  on  5/12/11.
 *   Copyright  2011  MIT  Media  Lab .  All  rights  reserved .
 *
 */
#if  ! defined ( __AU_CONVERT_H__ )
#define  __AU_CONVERT_H__

#define  SPLIT_WORD( split )  ( split >>24),  ( split >>16),  ( split >>8),  split

// render  quality  set  through  properties

void  ∗ auconvert_class ;

enum  eTimePitch_Parameters {
    kTimePitchParam_rate = 0,  // The  first  two  are  found  in  AudioUnitParameters . h .
        For  some  reason ,  the  rest  aren ' t
    kTimePitchParam_pitch = 1,
    kTimePitchParam_Smoothness = 3,
        kTimePitchParam_Tightness = 4,
    kTimePitchParam_AnalysisSize = 5,
    kTimePitchParam_AnalysisWindow = 6,
    kTimePitchParam_ResynthWindow = 7,
    kTimePitchParam_Overlap = 8,
```

```c
        kTimePitchParam_TransientFollow = 9,
        kTimePitchParam_TriggerThreshold1 = 10,
        kTimePitchParam_TriggerThreshold2 = 11,
        kTimePitchParam_WeakThreshold = 12,
32      kTimePitchParam_ResetPower = 13,
        kTimePitchParam_ResetScale = 14,
        kTimePitchParam_HardResetAmount = 15,
        kTimePitchParam_HardResetThreshold = 16,
        kTimePitchParam_LockingThreshold = 17,
        kTimePitchParam_HighFreqEnhance = 18,
        kTimePitchParam_QuietEnhance = 19,
        kTimePitchParam_LoudAggrK = 20,
        kTimePitchParam_SelectionFreq = 21,
        kTimePitchParam_PeakLocking = 22,
42      kTimePitchParam_PeakTransients = 23,
        kTimePitchParam_FullFrequency = 24
    };

    typedef struct _t_sInOutCallBack
    {
        const AudioTimeStamp *      mFirstOutputTS;
        Float64             mFirstInputSample;
        Float64             mNumInputSamples;
    } t_sInOutCallBack;
52
    typedef struct _t_AU_ParameterIDKey{
        char *  mStrId;
        int     mCode;
    } t_AU_ParameterIDKey;

    typedef struct _t_AU_ParameterIDList{
        t_AU_ParameterIDKey * mList;
        t_int           mSize;
    } t_AU_ParameterIDList;
62
    typedef struct _auconvert
    {
        t_pxobject  x_obj;
        void *      outlet2_loopSync; //loop sync out
        UInt32      inChannels;
        UInt32      outChannels;

        //USER Settings
        Boolean     loop;
72      UInt32      userStartTime;
        UInt32      userEndTime;

        //BUFFER VARIABLES
        t_symbol *ac_sym;
          t_buffer *ac_buf;

        //FADE VARIALES
        Boolean     fadeEnable;   // Enable mixing
        Boolean     fadeIn;       // perform fade in
82      Boolean     fadeOut;      // perform fade out
        Boolean     segEnded;     // to know if there is a cross-fade
        t_int       fadeInCount;  // number of samples faded (goes neg)
        UInt32      fadeStart;    // end of sample to blend.
        t_int       fadeOutCount; // number of samples faded
        UInt32      fadeTail;     // end of sample to blend.

        Boolean     fade;
        Boolean     crossFade;
        UInt32      fadeCount;
92
        //AU VARIABLES
        AudioComponentInstance theAU;
        Boolean     initialized;  // is AU ready to process
        Boolean     render_ready; // has a render process completed?
```

```
          Boolean    loopSyncFlag; // Used to send loop sync
          UInt32     frameCount;   // counts overall frames
          UInt32     startFrame;   // frame in buffer to read from
          Float64    externCount;  // incoming count requests.  Used to know when to
              reset.
          AudioBufferList* inBufs;
102       AudioBufferList* outBufs;
          t_int    render_framesCompleted; // not actually useful from what I can tell (
              mimiced from Ardour's example)
          t_sInOutCallBack callBackSampInfo;

          t_int     sampleRate;
      } t_auconvert;

      typedef t_float Sample;

      const t_int cMax_Channels = 4;
112   const t_int cMax_BufSize = 1024;   //frame size turns out to be 64 //confirmed
          data in is a float

      //#define PI 3.14159265
      const UInt32 cMaxCosTableSize = 1024; // must be factor of 2
      UInt32 cCosTableSize = 256;          //default
      t_float cosTable[cMaxCosTableSize];

      t_symbol *ps_buffer;

      //okay the correct generic way would be to pull the name from the AU and then
          pull out the spaces.  I'm too lazy at this point
122   //and half this list and use of it already exists.  Unless I support other AUs.
          Stick with this for now.
      static t_AU_ParameterIDKey sTimePitchParamsList[] = {
        {"rate"  , kTimePitchParam_rate}, //The first two are found in
            AudioUnitParameters.h.  For some reason, the rest aren't
        {"pitch", kTimePitchParam_pitch}, // 2
        {"smoothness", kTimePitchParam_Smoothness},
          {"tightness", kTimePitchParam_Tightness},
        {"AnalysisSize", kTimePitchParam_AnalysisSize},
        {"AnalysisWindow", kTimePitchParam_AnalysisWindow},
        {"ResynthWindow", kTimePitchParam_ResynthWindow},
        {"Overlap",kTimePitchParam_Overlap},
132     {"TransientFollow", kTimePitchParam_TransientFollow},
        {"TriggerThreshold1",kTimePitchParam_TriggerThreshold1},
        {"TriggerThreshold2",kTimePitchParam_TriggerThreshold2},
        {"WeakThreshold",kTimePitchParam_WeakThreshold},
        {"ResetPower",kTimePitchParam_ResetPower},
        {"ResetScale",kTimePitchParam_ResetScale},
        {"HardResetAmount",kTimePitchParam_HardResetAmount},
        {"HardResetThreshold",kTimePitchParam_HardResetThreshold},
        {"LockingThreshold", kTimePitchParam_LockingThreshold},
        {"HighFreqEnhance", kTimePitchParam_HighFreqEnhance},
142     {"QuietEnhance", kTimePitchParam_QuietEnhance},
        {"LoudAggrK", kTimePitchParam_LoudAggrK},
        {"SelectionFreq", kTimePitchParam_SelectionFreq},
        {"PeakLocking", kTimePitchParam_PeakLocking},
        {"PeakTransients", kTimePitchParam_PeakTranisents},
        {"FullFrequency", kTimePitchParam_FullFrequency}
      };

      static t_AU_ParameterIDList sTimePitchParams = {sTimePitchParamsList, 24};

152   void auconvert_dsp(t_auconvert *x, t_signal **sp);
      void auconvert_assist(t_auconvert *x, void *b, long m, long a, char *s);
      void *auconvert_new(t_symbol *s, long chan, long fadeLen);
      void *auconvert_free(t_auconvert *x);

      void auconvert_setNumChannels(t_auconvert * x, t_int inChan, t_int outChan);
```

```
    void auconvert_loop_in3(t_auconvert *x, t_int inVal);
    void auconvert_pitch_ft2(t_auconvert *x, double inVal);
    void auconvert_rate_ft1(t_auconvert *x, double inVal);
162 void auconvert_restart(t_auconvert *x);

    // Buffer Functions
    void auconvert_newBuffer(t_auconvert * x, t_symbol *s);
    void auconvert_setBuffer(t_auconvert *x, t_symbol *s);
    void auconvert_printBufInfo(t_auconvert * x);
    void auconvert_dblclick(t_auconvert *x);

    // AU Related Functions
172 void auconvert_newAU(t_auconvert * x);
    void auconvert_turnAuOn(t_auconvert *x);
    void auconvert_turnAuOff(t_auconvert *x);
    void auconvert_stopAU(t_auconvert *x);

    void auconvert_listAUs();
    void auconvert_getAU(t_auconvert *x, char *auName, OSType inType);
    void auconvert_printGenInfo(t_auconvert *x);
    void auconvert_prepAU(t_auconvert *x);
    void auconvert_prepAUwInSamplesCallback(t_auconvert *x);
182 void auconvert_configureIO(t_auconvert *x, t_int inC, t_int outC);

    int getParamIdFromStr(t_AU_ParameterIDList * inIDs, t_symbol * s);
    void auconvert_printAUParameters(t_auconvert * x);
    void auconvert_setAuParameter(t_auconvert *x, t_int inParamId, t_float
        inParamVal);
    void auconvert_setAUTimePitchParameter(t_auconvert *x, t_symbol *inStrID, double
        inVal);
    void auconvert_setRenderQuality(t_auconvert *x, t_int inVal);

    void auconvert_setFade(t_auconvert *x, t_int inVal);
    void auconvert_setLoop(t_auconvert *x, t_int inVal);
192 void auconvert_setStartPoint(t_auconvert *x, double inVal);
    void auconvert_setEndPoint(t_auconvert *x, double inVal);
    void auconvert_setSegmentPoints(t_auconvert *x, double inStartVal, double
        inEndVal);

    t_int *auconvert_run_mono(t_int *w);
    // t_int *auconvert_run_stereo(t_int *w);
    static OSStatus auconvert_render_callback(void * inRefCon,
                    AudioUnitRenderActionFlags *ioActionFlags,
                    const AudioTimeStamp      *inTimeStamp,
                    UInt32          inBusNum,
202                 UInt32          inNumFrames,
                    AudioBufferList*        ioData);

    static void auconvert_in_samples_per_output_callback(void *           inRefCon,
                        const AudioTimeStamp *    inOutputTimeStamp,
                        Float64               inInputSample,
                        Float64               inNumberInputSamples);


    #endif // __AU_CONVERT_H__
```

D.2.2   AuConvert˜ .c

```
/**@auconvert.c
 auconvert ˜ − code for hosting Apple's AU TimePitch
 06/14/2011 LsP

 @ingroup examples
 */

  #include "ext.h"
 9 #include "ext_obex.h"
  #include "ext_common.h" // contains CLIP macro
  #include "z_dsp.h"
  #include "buffer.h" // this defines our buffer's data structure and other
      goodies
```

137

```c
    #include "ext_atomic.h"
    #include "math.h"

    #include "AUConvert~.h"
    #include "CAAUParameter.h"
    #include <CoreServices/CoreServices.h>
19  #include <AudioUnit/AudioUnit.h>

    // ———————————————————————————— MSP MAIN ——— SETUP PATCH

    int main(void)
    {
        t_class *c;

        c = class_new("AUTimePitch~", (method)auconvert_new, (method)auconvert_free, (
            short)sizeof(t_auconvert), 0L, A_SYM, A_DEFLONG, A_DEFLONG, 0);
        class_dspinit(c);
29      class_addmethod(c, (method)auconvert_dsp, "dsp", A_CANT, 0);
        class_addmethod(c, (method)auconvert_setBuffer, "set", A_SYM, 0);
        class_addmethod(c, (method)auconvert_setRenderQuality, "quality", A_LONG, 0);
        class_addmethod(c, (method)auconvert_setAUTimePitchParameter, "param", A_SYM,
            A_FLOAT, 0);
        class_addmethod(c, (method)auconvert_printAUParameters, "pinfo", 0);
        class_addmethod(c, (method)auconvert_printGenInfo, "info", 0);
        class_addmethod(c, (method)auconvert_setNumChannels, "nchan", A_LONG, 0);
        class_addmethod(c, (method)auconvert_setFade, "fade", A_LONG, 0);
        class_addmethod(c, (method)auconvert_setLoop, "loop", A_LONG, 0);
        class_addmethod(c, (method)auconvert_setStartPoint, "start", A_FLOAT, 0);
39      class_addmethod(c, (method)auconvert_setEndPoint, "end", A_FLOAT, 0);
        class_addmethod(c, (method)auconvert_setSegmentPoints, "segment", A_FLOAT,
            A_FLOAT, 0);
        class_addmethod(c, (method)auconvert_loop_in3, "in3", A_LONG, 0);
        class_addmethod(c, (method)auconvert_pitch_ft2, "ft2", A_FLOAT, 0);
        class_addmethod(c, (method)auconvert_rate_ft1, "ft1", A_FLOAT, 0);
        class_addmethod(c, (method)auconvert_restart, "bang", 0);
        class_addmethod(c, (method)auconvert_dblclick, "dblclick", A_CANT, 0);
        class_addmethod(c, (method)auconvert_assist, "assist", A_CANT, 0);

        class_register(CLASS_BOX, c);
49      auconvert_class = c;

        ps_buffer = gensym("buffer~");

        return 0;
    }

    //
    void auconvert_dsp(t_auconvert *x, t_signal **sp)
    {
59      auconvert_setBuffer(x, x->ac_sym);
        dsp_add(auconvert_run_mono, 4, x, sp[0]->s_vec, sp[1]->s_vec, sp[0]->s_n);
        //      else
        //          dsp_add(auconvert_run_stereo, 5, x, sp[0]->s_vec, sp[1]->s_vec, sp
        [2]->s_vec, sp[0]->s_n);
    }

    // ———————————————————————— INLETS
    void auconvert_loop_in3(t_auconvert *x, t_int inVal){
        auconvert_setLoop(x, inVal);
    }
69
    void auconvert_pitch_ft2(t_auconvert *x, double inVal){
        auconvert_setAuParameter(x, kTimePitchParam_pitch, inVal);
    }

    void auconvert_rate_ft1(t_auconvert *x, double inVal){
        auconvert_setAuParameter(x, kTimePitchParam_rate, inVal);
    }

    void auconvert_restart(t_auconvert *x){
```

```
79    //don't want to touch x->userStartTime as that could interfere with buffer
          read processes
      //changing external count to its max should trigger restart (Also triggers
          loopSync to reset)
      x->externCount = INFINITY;
    }

    // ————————————————————————————— USER SETTINGS
    void auconvert_printGenInfo(t_auconvert *x){
      post("AU TimePitch Info: ");
      post("  Channels In:%d   Channels Out: %d \n", x->inChannels, x->outChannels)
          ;
      UInt32  rq;
89    UInt32  propSize;
      if (x->theAU != NULL) {
        AudioUnitGetProperty(x->theAU, kAudioUnitProperty_RenderQuality,
            kAudioUnitScope_Global, 0, &rq, &propSize);
        post("  Sample Rate:%d   Render Quality is %d (Best Quality is 127)\n", x->
            sampleRate, rq);
      }
      else post("  Sample Rate:%d ",  x->sampleRate);

    //   Float64 prop;
    //   AudioUnitGetProperty(x->theAU, kAudioUnitProperty_Latency,
          kAudioUnitScope_Global, 0, &prop, &propSize);
    //   post(" Latency from AU: %.8f ms", prop);  //claims it is 0 anyway from what
          I can tell
99    post("  CrossFade is: %s\n", (x->fadeEnable) ? "ON" : "OFF");
      post("  Loop is: %s", (x->loop) ? "ON" : "OFF");
      auconvert_printBufInfo(x);
      if (x->userStartTime || x->userEndTime) post("  Segment Start:%.3f and Ends:
          %.3f",(x->userStartTime / (double) x->sampleRate)   ,(x->userEndTime / (
          double) x->sampleRate));
    }

    void auconvert_setRenderQuality(t_auconvert *x, t_int inVal){
      //only set render quality while AU can be paused.  Requires uninitializing
      Boolean isOn = x->initialized;
      if(x->x_obj.z_disabled) {
109      if (-1 < inVal < 128){
          AudioUnitUninitialize(x->theAU);
          x->initialized = false;
          AudioUnitSetProperty(x->theAU, kAudioUnitProperty_RenderQuality,
              kAudioUnitScope_Global, 0, &inVal, sizeof(inVal));
          if (isOn) {
            AudioUnitInitialize(x->theAU);
            x->initialized = true;
          }
        }
        else post("Invalid Input: Valid Range for Render Quality 0 (poorest) — 128 (
            best but more computationally intensive)");
119   }
      else post("Audio must be stopped in order to change render quality.");
    }

    void auconvert_setFade(t_auconvert *x, t_int inVal){
      if (inVal == 0) x->fadeEnable = false;
      else x->fadeEnable = true;
    //   post("Fade Set %s\n", (x->fadeEnable) ? "On" : "Off");
    }

129 void auconvert_setLoop(t_auconvert *x, t_int inVal){
      if (inVal == 0) x->loop = false;
      else x->loop = true;
    }

    //takes in ms
```

```c
    void auconvert_setStartPoint(t_auconvert *x, double inVal){
      t_buffer * b = x->ac_buf;

      if (b != NULL){
        // convert from ms to samples
        UInt32 startSamp = (UInt32) (inVal * x->sampleRate);
        ATOMIC_INCREMENT(&b->b_inuse);
        if (!b->b_valid) {
          post("No Buffer Loaded.  Unable to set start point.");
        }
        else {
          if (startSamp < b->b_frames)
            if ((x->userEndTime == 0) || (startSamp < x->userEndTime)) x->
                userStartTime = startSamp;
            else post("Invalid Start Time: Start at or after Segment End time %.4f s
                ", ((float) x->userStartTime / x->sampleRate));
          else post("Invalid start time:  Buffer is only %d s long.", (int) (b->
              b_frames / x->sampleRate));
        }
    //    post("StartPoint Set to: %d", x->userStartTime);
        ATOMIC_DECREMENT(&b->b_inuse);
      }
      else post("No Buffer Loaded.  Unable to set start point.");
    }

    // takes in ms
    void auconvert_setEndPoint(t_auconvert *x, double inVal){
      t_buffer * b = x->ac_buf;

      if (b != NULL){
        // convert from ms to samples
        UInt32 endSamp = (UInt32) (inVal * x->sampleRate);
        ATOMIC_INCREMENT(&b->b_inuse);
        if (!b->b_valid) {
          post("No Buffer Loaded.  Unable to set end point.");
        }
        else {
          if ((endSamp != 0) && (endSamp < x->userStartTime)) post("Invalid End time
              : End time is before the requested start at %d s.", (x->userStartTime
              / x->sampleRate));
          else {
            if (endSamp > b->b_frames) post("Warning: End time is after buffer end.
                 Buffer is only %d s long.", (b->b_frames / x->sampleRate));
            x->userEndTime = endSamp;
          }
        }
    //    post("EndPoint Set to: %d", x->userEndTime);

        ATOMIC_DECREMENT(&b->b_inuse);
      }
      else post("No Buffer Loaded.  Unable to set end point.");
    }

    void auconvert_setSegmentPoints(t_auconvert *x, double inStartVal, double
        inEndVal){
      t_buffer * b = x->ac_buf;

      // convert from ms to samples
      UInt32 inStartSamp = (UInt32) (inStartVal * x->sampleRate);
      UInt32 inEndSamp = (UInt32) (inEndVal * x->sampleRate);

      if (b != NULL){
        ATOMIC_INCREMENT(&b->b_inuse);
        if (!b->b_valid) {
          post("No Buffer Loaded.  Unable to set segment.");
        }
        else {
```

```
            if (inStartSamp > inEndSamp) {
                post("Invalid_Input_Times:_Start_is_after_end.");
            }
            else if (inStartSamp < b->b_frames) {
199             x->userStartTime = inStartSamp;
                if (inEndSamp > b->b_frames) post("Warning:_End_time_is_after_buffer_end
                    .__Buffer_is_only_%d_s_long.", (b->b_frames / x->sampleRate));
                x->userEndTime = inEndSamp;
            }
            else  post("Invalid_start_time.__Buffer_is_only_%d_s_long.", (b->b_frames /
                x->sampleRate));
        }

        ATOMIC_DECREMENT(&b->b_inuse);
    }
    else  post("No_Buffer_Loaded.__Unable_to_set_segment.");
209 }


    // ————————————————————————————— PARAMETERS
    void auconvert_setAuParameter(t_auconvert *x, t_int inParamId, t_float
        inParamVal){
        if (x->theAU) {
            Float32 theParam;
            AudioUnitSetParameter(x->theAU, inParamId, kAudioUnitScope_Global, 0,
                inParamVal, 0);
            AudioUnitGetParameter(x->theAU, inParamId, kAudioUnitScope_Global, 0, &
                theParam);
    //      post("Parameter Changed to: %.3f", theParam);
219     }
    }

    t_int getParamIdFromStr(t_AU_ParameterIDList * inIDs, t_symbol * s){

        t_int i;

        for (i = 0; i < inIDs->mSize; i++){
            if (strcmp(inIDs->mList[i].mStrId, s->s_name) == 0)
                return (inIDs->mList[i].mCode);
229     }

        return (-1); // code not found
    }

    void auconvert_printAUParameters(t_auconvert * x){
        if (x->theAU){
            UInt32 listSize;
            AudioUnitParameterID paramList[256];
            AudioUnitGetProperty(x->theAU, kAudioUnitProperty_ParameterList,
                kAudioUnitScope_Global, 0, &paramList, &listSize);
239         post("Supported_Parameters_Are:_");
            post("___Name_____Min_____Max_____Present");
            UInt32 i;
            for (i = 0; i < (listSize/sizeof(AudioUnitParameterID)); i++){

                CAAUParameter aParam(x->theAU, paramList[i] , kAudioUnitScope_Global, 0);
                const AudioUnitParameterInfo& info  = aParam.ParamInfo();

                Float32 pval;
                AudioUnitGetParameter(x->theAU, paramList[i], kAudioUnitScope_Global, 0, &
                    pval);
249             if (info.cfNameString){
                    char cName[256];
                    CFStringGetCString(info.cfNameString, cName, 256, kCFStringEncodingASCII
                        );
                    post("__%-20s__%10.2f__%10.2f__%10.2f", cName, info.minValue, info.
                        maxValue, pval);
                }
```

```
        }
        post("To set parameter, type as above but remove space keeping
            capitalization.");
    } else post("AU unavailable;");
}

259
    void auconvert_setAUTimePitchParameter(t_auconvert *x, t_symbol *inStrID, double
        inVal) {
        t_int paramId = getParamIdFromStr(&sTimePitchParams, inStrID);

        if (paramId != -1) {
            auconvert_setAuParameter(x, paramId, inVal);
        }
        else post("Could not find requested parameter %s.", inStrID->s_name);
    }

269 // ————————————————————————— GENERAL AU INFO
    void auconvert_listAUs(){
        AudioComponentDescription looking;
        looking.componentType            = kAudioUnitType_FormatConverter; //
            kAnyComponentType;
        looking.componentSubType         = 0;// kAudioUnitSubType_Varispeed; //
            kAnyComponentSubType;
        looking.componentManufacturer    = 0; // kAnyComponentManufacturer;
        looking.componentFlags           = 0;
        looking.componentFlagsMask       = 0; // cmpIsMissing;

        UInt32 numComponents = AudioComponentCount ( &looking );
279
        post("Found %ld components.\n", numComponents);

        AudioComponent component = 0;
        int i = 0;
        while (true)
        {
            component = AudioComponentFindNext(component, &looking);

            if ( 0 == component )
289         break;

            AudioComponentDescription desc;
            OSErr err = AudioComponentGetDescription(component, &desc);

            if ( err != noErr )
            {
                printf("Couldn't find any info on component %d of %ld in list!\n", i
                    , numComponents);
                break;
299         }

            CFStringRef componentName;
            err = AudioComponentCopyName(component, &componentName);
            char cName[256];
            CFStringGetCString(componentName, cName, 256, kCFStringEncodingASCII);

            post("%d of %ld: '%c%c%c%c', '%c%c%c%c', '%c%c%c%c', '%s'\n",
                i, numComponents,
                SPLIT_WORD( desc.componentManufacturer ),
309             SPLIT_WORD( desc.componentType ),
                SPLIT_WORD( desc.componentSubType ),
                cName);

            ++i;
        }

    }

    // check for any required properties to set
319 void auconvert_getTimePitch(t_auconvert *x){
```

```
        if (x−>theAU != NULL){
           AudioUnitUninitialize(x−>theAU);
           AudioComponentInstanceDispose(x−>theAU);
        }

        AudioComponentDescription looking;
        looking.componentType           = kAudioUnitType_FormatConverter;
        looking.componentSubType         = kAudioUnitSubType_TimePitch; //
           kAnyComponentSubType;
        looking.componentManufacturer    = 0; //kAnyComponentManufacturer;
329     looking.componentFlags           = 0;
        looking.componentFlagsMask       = 0; //cmpIsMissing;

        AudioComponent component = 0;
        component = AudioComponentFindNext(component, &looking);
        AudioComponentInstanceNew(component, &(x−>theAU));

    }
    // ─────────────────────────────────────────── FIND AND GET SPECIFIC AU OF GIVEN
        TYPE AND NAME
339
    void auconvert_getAU(t_auconvert *x, char *auName, OSType inType){

        if (x−>theAU != NULL){
           AudioUnitUninitialize(x−>theAU);
           AudioComponentInstanceDispose(x−>theAU);
        }

        AudioComponentDescription looking;
        looking.componentType           = inType; //kAnyComponentType;
349     looking.componentSubType         = 0;//kAudioUnitSubType_TimePitch; //
           kAnyComponentSubType;
        looking.componentManufacturer    = 0; //kAnyComponentManufacturer;
        looking.componentFlags           = 0;
        looking.componentFlagsMask       = 0; //cmpIsMissing;

        char fullName[256] = "Apple:_";
        strcat(fullName, auName);

        AudioComponent component = 0;
        while (true)
359     {
           component = AudioComponentFindNext(component, &looking);

           if ( 0 == component )
             break;

           CFStringRef componentName;
           OSErr err = AudioComponentCopyName(component, &componentName);
           char cName[256];
           if (!err){
369          CFStringGetCString(componentName, cName, 256, kCFStringEncodingASCII);
             if (strcmp(fullName, cName) == 0){
               post("Initializing_%s.\n", cName);
               AudioComponentInstanceNew(component, &(x−>theAU));
             }
           }
        }

        if (x−>theAU == NULL){
           post("Audio_Unit_%s_couldn't_be_found.", auName);
379     }
    }
    //─────────────────────────────────────────── CONFIGURE IO

    void auconvert_configureIO(t_auconvert *x, t_int inC, t_int outC){

        AudioStreamBasicDescription streamDesc;
```

```
        Boolean running = x->initialized;
389
        //  post("AU is turned on? %s\n", x->initialized ? "yes" : "no");

        if (x->initialized) {
          UInt32 propSize;
          AudioUnitGetProperty(x->theAU, kAudioUnitProperty_StreamFormat,
              kAudioUnitScope_Input, 0, &streamDesc, &propSize);
          t_int presInC = streamDesc.mChannelsPerFrame;

          //      post("Stream Format In: Format- %c%c%c%c Flags %d, Sample Rate %6.0f,
          //      Bit Depth- %d, PacketSize- %d, Frames in Packet- %d, #Channels- %d
          //      ioSize? %d",
          //          SPLIT_WORD(streamDesc.mFormatID), streamDesc.mFormatFlags,
          //      streamDesc.mSampleRate, streamDesc.mBitsPerChannel, streamDesc.
          //      mBytesPerPacket, streamDesc.mFramesPerPacket, streamDesc.
          //      mChannelsPerFrame, propSize);
399
          AudioUnitGetProperty(x->theAU, kAudioUnitProperty_StreamFormat,
              kAudioUnitScope_Output, 0, &streamDesc, &propSize);
          t_int presOutC = streamDesc.mChannelsPerFrame;

          //      post("Stream Format Out: Sample Rate %6.0f, Bit Depth- %d, PacketSize-
          //      %d, Frames in Packet- %d, #Channels- %d ioSize? %d",
          //          streamDesc.mSampleRate, streamDesc.mBitsPerChannel, streamDesc.
          //      mBytesPerPacket, streamDesc.mFramesPerPacket, streamDesc.
          //      mChannelsPerFrame, propSize);

          //      post("Why is this wrong? in %d, %d,  out %d, %d", inC, presInC, outC,
          //      presOutC);

409     // if we are already running with the requested i/o config then no change
          if ( (inC == presInC) && (outC == presOutC) ) {
            return;
          }
          else {
            AudioUnitUninitialize(x->theAU);
            x->initialized = false;
          }
        }

419     streamDesc.mSampleRate = x->sampleRate;
        streamDesc.mFormatID = kAudioFormatLinearPCM;
        //CHECK THIS ONE COULD BE WRONG
        //  streamDesc.mFormatFlags = kAudioFormatFlagIsFloat|kAudioFormatFlagIsPacked
        //  |kAudioFormatFlagIsNonInterleaved;
        streamDesc.mFormatFlags = kAudioFormatFlagIsFloat|
            kAudioFormatFlagIsNonInterleaved;
        streamDesc.mFramesPerPacket = 1;

        /* apple says that for non-interleaved data, these
         values always refer to a single channel.
         */
429
        streamDesc.mBitsPerChannel = 32;
        streamDesc.mBytesPerPacket = 4;
        streamDesc.mBytesPerFrame = 4;

        streamDesc.mChannelsPerFrame = inC;

        // set input
        OSErr err;
        if ((err = AudioUnitSetProperty(x->theAU, kAudioUnitProperty_StreamFormat,
            kAudioUnitScope_Input, 0, &streamDesc, sizeof(streamDesc))) != 0) {
439     post("Problem_Setting_Up_Input:_%d", err);
        }
        x->inChannels = inC;
```

144

```
         UInt32 propSize;
         AudioUnitGetProperty(x->theAU, kAudioUnitProperty_StreamFormat,
            kAudioUnitScope_Input, 0, &streamDesc, &propSize);

         if (!running) {//should be first time
           post("Stream_Format_In_Set_To:_Format−%c%c%c%c_Flags_%d,_Sample_Rate_%6.0f,
              _Bit_Depth−%d,_PacketSize−%d,_Frames_in_Packet−%d,_#Channels−%d_
              ioSize?_%d",
             SPLIT_WORD(streamDesc.mFormatID), streamDesc.mFormatFlags, streamDesc.
                mSampleRate, streamDesc.mBitsPerChannel, streamDesc.mBytesPerPacket,
                streamDesc.mFramesPerPacket, streamDesc.mChannelsPerFrame, propSize);
449    }

         streamDesc.mChannelsPerFrame = outC;

         //set output
         if ((err = AudioUnitSetProperty(x->theAU, kAudioUnitProperty_StreamFormat,
            kAudioUnitScope_Output, 0, &streamDesc, sizeof(streamDesc))) != 0) {
           post("Problem_Setting_Up_Output:_%d", err);
         }
         x->outChannels = outC;

459
         if (running) {
           auconvert_turnAuOn(x);
         }

     }

     //Apparently buffers only get 4 channels max
     void auconvert_setNumChannels(t_auconvert * x, t_int inChan, t_int outChan){

469    if (inChan)
           x->inChannels = CLIP(inChan,1,x->ac_buf->b_nchans);
         else
           x->inChannels = 1;

         if (outChan)
           x->outChannels = CLIP(outChan,1,x->ac_buf->b_nchans);
         else
           x->outChannels = 1;
     }
479
     void auconvert_prepAU(t_auconvert *x){
         if (x->theAU != NULL) {
           //this is where setup should happen but I don't actually know what I'd need
              to do right now
           AURenderCallbackStruct renderCallbackInfo;

           renderCallbackInfo.inputProc = auconvert_render_callback;
           renderCallbackInfo.inputProcRefCon = x; //think this is the class to call
              from i.e this->auconvert_render_callback

489      OSErr err = AudioUnitSetProperty(x->theAU,
              kAudioUnitProperty_SetRenderCallback, kAudioUnitScope_Input,
                        0, &renderCallbackInfo, sizeof(renderCallbackInfo));
           if (err) {
             post("Problem_Setting_Callback");
           }
           else auconvert_turnAuOn(x);

         }
         else {
           post("Error_Loading_AU");
499    }

     }

     void auconvert_prepAUwInSamplesCallback(t_auconvert *x) {
```

```
        if (x−>theAU != NULL){
          AUInputSamplesInOutputCallbackStruct inSampPerOutCBInfo;
          inSampPerOutCBInfo.inputToOutputCallback =
              auconvert_in_samples_per_output_callback;
          inSampPerOutCBInfo.userData = x;

509
          OSErr err = AudioUnitSetProperty(x−>theAU,
              kAudioUnitProperty_InputSamplesInOutput, kAudioUnitScope_Global,
                          0, &inSampPerOutCBInfo, sizeof(inSampPerOutCBInfo));
          if (err) {
            post("Problem Setting Callback");
          }
          else {
            auconvert_prepAU(x);    //sets render callback
            auconvert_turnAuOn(x);
          }
519   }
        else {
          post("Error Loading AU");
        }

    }

    void auconvert_turnAuOn(t_auconvert *x) {
      if ((x−>theAU != NULL) && (!x−>initialized)) {
        //should add error checking here
529     AudioUnitInitialize(x−>theAU);
        x−>initialized = true;
        x−>frameCount = 0;
    //    post("AU is turned on? %s\n", x−>initialized ? "yes" : "no");
      }
    }

    void auconvert_turnAuOff(t_auconvert *x) {
      if (x−>initialized) {
        //should add error checking here
539     AudioUnitUninitialize(x−>theAU);
        x−>initialized = false;
    //    post("AU is turned on? %s\n", x−>initialized ? "yes" : "no");
      }
    }

    void auconvert_stopAU(t_auconvert *x){
      if (x−>theAU != NULL){
        auconvert_turnAuOff(x);
        AudioComponentInstanceDispose(x−>theAU);
549   }
    }

    //————————————————————————————————————————— RENDER CALLBACK
    Boolean isFading(t_auconvert * x){
      if ((x−>fadeIn) || (x−>fadeOut))
        return true;
      else return false;
    }

559 //Drops whatever fade−in was happening and switches to new
    void startFadeIn(t_auconvert * x, Float64 fadeStart){
      x−>fadeInCount = cCosTableSize −1;
      x−>fadeStart = fadeStart;
      x−>fadeIn = true;
    //  cpost("Fade In Started");
    }

    //Contines it's first fade out.
    void startFadeOut(t_auconvert * x, Float64 tailStart){
```

146

```c
569     if (!x->fadeOut){
    //      cpost("FadeOutStarted");
          x->fadeOutCount = 0;
          x->fadeTail = tailStart;
          x->fadeOut = true;
        }
    }

    //Expects buffer handling to already be complete
    void auconvert_fetchNoFade(t_auconvert * x, t_buffer * inReadBuf, uint32_t
        inStartSamp, uint32_t inNumSamps, uint32_t inSegEnd){
579     if (inSegEnd > inStartSamp) {
            float * readBuf = inReadBuf->b_samples;
            uint32_t numBufChannels = inReadBuf->b_nchans;

    //      post("Fetching no fade Start: %d, numSamps %d, segEnd %d", inStartSamp,
        inNumSamps, inSegEnd);

            uint32_t numSamps = inNumSamps;
            if ((inStartSamp + inNumSamps) >= inSegEnd) numSamps = inSegEnd -
                inStartSamp; //no overruns
            //Buffers are saved interleaved and plugin uses non-interleaved
            if (x->inChannels == 1) memcpy(x->inBufs->mBuffers[0].mData, readBuf +
                inStartSamp, numSamps*sizeof(Sample));
589         else {
              uint32_t i;
              //copy each channel independently
              for (i = 0; i < x->inChannels; ++i) {
                uint32_t n;
                // fetch all samples in a particular channel
                for (n = 0; n < numSamps; n++) {//memcopy?
                  //must increment raw pointer by data size. Array handles that
                      automatically just need
                  *((float *) ((int) x->inBufs->mBuffers[i].mData + n*sizeof(Sample))) =
                      readBuf[(n+inStartSamp)*numBufChannels + i];
                }
599         }
          }

        }
        else x->segEnded = true;
    }

    // if end is within 50% of note end, start an end
    // if loop is on, crossfade next in.
    //if I get a trigger to start next note, then a fade-in begins
609 OSErr auconvert_fetchInput(t_auconvert * x, uint32_t inStartSamp, uint32_t
        inNumSamps){

    //      ioData->mBuffers[i].mData = x->inBufs->mBuffers[i].mData + x->
        render_framesCompleted;

        t_int numSamps = inNumSamps;  //seems like this should always be an integer.
            It'd be really odd otherwise.
        if (numSamps > cMax_BufSize) {
          post("ERROR_IN_BUF_REQUEST.__BUFF_TOO_LARGE");
          numSamps = cMax_BufSize;
        }

619     // preset input buffers to be zero
        uint32_t i;
        for (i = 0; i < x->inChannels; i++) {
          memset(x->inBufs->mBuffers[i].mData, 0, numSamps*sizeof(Sample));
        }

        t_buffer *b = x->ac_buf;

        if (!b)
          return noErr; //buffers already set to zero
```

```
629    ATOMIC_INCREMENT(&b->b_inuse);
       if (!b->b_valid) {
         ATOMIC_DECREMENT(&b->b_inuse);
          return noErr;  //not returning an error as this should not cause
              computational issues
       }
       float * readBuf = b->b_samples;
       uint32_t readBufTotFrames = b->b_frames;
       uint32_t numBufChannels = b->b_nchans;

       //rounding the float to nearest int
639    uint32_t startSamp = inStartSamp;

       uint32_t endFrame = ((x->userEndTime) && (x->userEndTime < readBufTotFrames))
          ? x->userEndTime : readBufTotFrames;

       if (numBufChannels < x->inChannels) {
         post("Error:_Insufficient_Channels_in_Buffer");
          return noErr;
       }

       cpost("TotSamples_%d_:_Num_Samples_Requested_%d_:_End_Samp_%d_", endFrame,
          numSamps, numSamps + startSamp);
649
       if (((startSamp + (cCosTableSize/2)) > endFrame) || (startSamp+numSamps >
          endFrame)) {
         if (!x->loopSyncFlag){
           outlet_bang(x->outlet2_loopSync);
           x->loopSyncFlag = true;
           if (x->fadeEnable && (!x->loop)) startFadeOut(x, startSamp);
         }

         // if loop is enabled, start reading from loop start point
         if (x->loop){
659        uint32_t newStart = (x->userStartTime) ? x->userStartTime : 0;
           //get fading info
           if (x->fadeEnable) {
             startFadeIn(x, newStart);
             startFadeOut(x, startSamp);
   //        post("Fade at %d, loop end at %d with dif being %d", startSamp,
       endFrame, (endFrame-startSamp));
           }

           //if user end time has been set, loop then. If not default to buffer
               length
           startSamp = newStart;
669        x->startFrame = startSamp; //not keen on this architecturally.
           x->loopSyncFlag = false;  // clear loop flag so it can send again. Non-
               looping this is only reset by reset
           x->segEnded = false;
         }
       }

       // want min of user requested finish and buffer validity
       if (startSamp < readBufTotFrames) {  //never read past end file. Just a bad
          idea

         if (isFading(x)) {
679        if ((x->fadeOut) || (startSamp < endFrame)){
             // repeat for however many channels
             uint32_t i;
             for (i = 0; i < x->inChannels; i++) {

   //            post("Performing Fade, Start Samp %d Tail : %d \n", startSamp, x->
       fadeTail);
               //get all samples in one channel
               t_int n;
               for (n = 0; n < numSamps; n++) {
```

```c
689                     // note that this is only a single channel fade.  (seeing as
                           increments happen within n loop)
                        Float64 dataStart = 0;
                        if ((n+startSamp) < endFrame){
                           if (x->fadeIn) dataStart = readBuf[(n+startSamp)*numBufChannels +
                               i]*cosTable[x->fadeInCount--];
                           else if (!x->fadeOut) dataStart = readBuf[(n+startSamp)*
                               numBufChannels + i];   //implies no fades
                        }
                        else x->segEnded = true;

                        Float64 dataEnd = 0;
                        if ((x->fadeOutCount+x->fadeTail) > readBufTotFrames) x->fadeOut =
                            false;
699                     if (x->fadeOut) dataEnd = readBuf[(x->fadeOutCount +  x->fadeTail)*
                            numBufChannels + i]*cosTable[x->fadeOutCount];

                        *((float *) ((int) x->inBufs->mBuffers[i].mData + n*sizeof(Sample)))
                            = dataStart+dataEnd;

                        if (x->fadeInCount < 0) x->fadeIn = false;

                        x->fadeOutCount++;
                        if (!(x->fadeOutCount < cCosTableSize)) x->fadeOut = false;
     //               post("FADE: SampleOld: %d SampleNew %d\n", (x->fadeTail + x->
         fadeOutCount), (startSamp+n);
     //               post("FadeInfo: New %.6f, Old %.6f, Sum %.6f, CountIn %d, CountOut
         %d, NumTotalSamples %d\n", dataStart, dataEnd, *((float *) ((int) x->inBufs
         ->mBuffers[i].mData + n*sizeof(Sample))), x->fadeInCount, x->fadeOutCount,
         numSamps);
709                  }
                  }
               }
            }
            else {
               auconvert_fetchNoFade(x, b, startSamp, numSamps, endFrame);
            }

     //      cpost("TotSamples %d : Num Samples Requested %d : End Samp %d ", endFrame,
         numSamps, numSamps + startSamp);
            //copy each channel
719         // x->inChannels is clamped to num of buffers
            //buffer sounds to be interleaved. AU set for non-interleaved (I could fix
                that to be interleaved which would be better probably)
     //         post("Single Channel request %d frames.  in: %2.6f\n", numSamps, *((
         float*) x->inBufs->mBuffers[0].mData));

      }
      else x->segEnded = true;

      ATOMIC_DECREMENT(&b->b_inuse);

      return noErr;
729 }

     // I CAN NOT TELL WHERE THIS FUNCTION REALLY HAS RELEVANCE.  IT IS CALLED EVERY
         REQUESTED FRAME BUT RENDER ACTUALLY WORKS REGARDLESS OF THE
     //COUNTS SPECIFIED HERE WHICH DON'T ADD UP TO MATCH WHAT CALLBACK REQUESTS
         ANYWAY
     static void auconvert_in_samples_per_output_callback(void *            inRefCon,
                            const AudioTimeStamp *    inOutputTimeStamp,
                            Float64              inInputSample,
                            Float64              inNumberInputSamples)
      {
         t_auconvert * x = (t_auconvert *) inRefCon;
739
         x->callBackSampInfo.mFirstOutputTS = inOutputTimeStamp;
         x->callBackSampInfo.mFirstInputSample = inInputSample;
         x->callBackSampInfo.mNumInputSamples = inNumberInputSamples;
```

```
//    post("InSamplesPerOutputCallback Called: First In %.2f numIn %.2f",
     inInputSample, inNumberInputSamples);
}

     static OSStatus auconvert_render_callback(void * inRefCon,
                    AudioUnitRenderActionFlags *ioActionFlags,
749                    const AudioTimeStamp      *inTimeStamp,
                    UInt32          inBusNum,
                    UInt32          inNumFrames,
                    AudioBufferList*          ioData) {
     //verify buffer is filled.
     //fill io buffers I think.
     //update frame count
     //verify there is data to grab

     t_auconvert * x = (t_auconvert *) inRefCon;
759
     if (!x->render_ready) {  //this verifies buffers ready
       post ("AUPlugin:_render_callback_called_illegally!");
       return kAudioUnitErr_CannotDoInCurrentContext;
     }

     //gots to grab appropriate audio buffers.
     //these should be the same
     uint32_t limit = ((uint32_t) ioData->mNumberBuffers <  x->inChannels) ? (
         uint32_t) ioData->mNumberBuffers : x->inChannels;
     uint32_t i;
769
     auconvert_fetchInput(x, x->startFrame , inNumFrames);
     //    post("In render startFrame: %.2f, inTS %.2f, inSampPerOut %.2f", x->
         startFrame, x->callBackSampInfo.mFirstInputSample);

     for (i = 0; i < limit; ++i) {
        ioData->mBuffers[i].mNumberChannels = x->inBufs->mBuffers[i].mNumberChannels
            ;
        ioData->mBuffers[i].mDataByteSize = x->inBufs->mBuffers[i].mDataByteSize;
        ioData->mBuffers[i].mData = (void *) ((int) x->inBufs->mBuffers[i].mData + x
            ->render_framesCompleted);
     }

779 //    post("io data in %2.6f, +16 %2.6f\n", *((t_float *)ioData->mBuffers[0].mData
     ), *((t_float *) ioData->mBuffers[0].mData + (16*sizeof(Sample)))));

     x->startFrame += inNumFrames;
     x->render_framesCompleted += inNumFrames;  //This is unlikely to ever not be a
         single call, but just in case

     return noErr;
     }
     //───────────────────────────────────────────────── RUN

789 t_int *auconvert_run_mono(t_int *w){

     t_auconvert * x = (t_auconvert *) w[1];
     t_int numFrames = (t_int) w[4];
     t_float * out = (t_float *) w[3];
     auconvert_configureIO(x, 1, 1);

     while(1){
        // is audio on?
        if(x->x_obj.z_disabled) break;
799
        // is buffer valid?
        t_buffer *b = x->ac_buf;

        if (!b){
          while (numFrames--) *out++ = 0.;
          break;
        }
```

```
        ATOMIC_INCREMENT(&b->b_inuse);
        if (!b->b_valid) {
809         ATOMIC_DECREMENT(&b->b_inuse);
          while (numFrames--) *out++ = 0.;
          break;
        }
        ATOMIC_DECREMENT(&b->b_inuse);

        // reusable?
        AudioUnitRenderActionFlags flags = 0;
        AudioTimeStamp ts;
        OSErr err;
819
        if (numFrames > cMax_BufSize) {
          post("ERROR: TOO MANY FRAMES.  REDUCE AND TRY AGAIN");
          w[3] = w[2];
          break;
        }

        // set up input buffers
        x->inBufs->mNumberBuffers = x->inChannels;

829     Float64 inCount = *((t_float *) w[2]);
        // is incoming count continuously increasing or stable? If yes, keep going.
            If not, reset start point.  Only reset call
        // Only change to startframe outside of the actual increment in render call
           (appart from inits)
        if (x->externCount > inCount){
          UInt32 newStart = (x->userStartTime) ? x->userStartTime : 0;
          if (x->fadeEnable) {
            if (!x->segEnded) startFadeOut(x, x->startFrame);
            startFadeIn(x, newStart);
          }
          x->startFrame = newStart;
839       x->loopSyncFlag = false;
          x->segEnded = false;
        }
    //  post("In: %.2f eC: %.2f SF %.2f", inCount, x->externCount, x->startFrame);
        x->externCount = inCount;

        uint32_t k;
        // nice mixture of mono and stero coding no?
        for(k = 0; k < x->inChannels; k++) {
          x->inBufs->mBuffers[k].mNumberChannels = 1;
849       x->inBufs->mBuffers[k].mDataByteSize = numFrames*sizeof(Sample);
          // data filled in in fetch function
        }

        // in comparison, I never have a offset from the buffer.  Except short.  W[]
            is always the start
        x->render_framesCompleted = 0;
        x->render_ready = true;
        //  ATOMIC_INCREMENT(x->render_ready); // this caused crash and I don't
            really need it

        // does this actually make sense? or should it get set locally?  Fine for now
            .
859     x->outBufs->mNumberBuffers = x->outChannels;

        // the render_call will fills in the rendering buffers
        // clear output buffer
        uint32_t i;
        for (i = 0; i < x->outBufs->mNumberBuffers; ++i) {
          x->outBufs->mBuffers[i].mNumberChannels = 1;
          x->outBufs->mBuffers[i].mDataByteSize = numFrames * sizeof (Sample);   //
              should set Samplesize in this in ioConfig
          memset((t_float *) (x->outBufs->mBuffers[i].mData), 0, numFrames*sizeof(
              Sample));
        }
```

```
        ts.mSampleTime = x->externCount;  // this only appears to actually effect
             later systems when audio clock reset
        ts.mFlags = kAudioTimeStampSampleTimeValid;

        if ((err = AudioUnitRender(x->theAU, &flags, &ts, 0, numFrames, x->outBufs))
            == noErr) {

          // stating buffers empty
          //    ATOMIC_DECREMENT(x->render_ready);
          x->render_ready = false;
          x->frameCount += numFrames;
```
```
          uint32_t i;

          // might be incorrectly limiting this.  See audio_unit.cc
          for (i = 0; i < x->outBufs->mNumberBuffers; ++i) {
            if (out != x->outBufs->mBuffers[i].mData) {
              memcpy (out, x->outBufs->mBuffers[i].mData, numFrames * sizeof (Sample
                 ));
            }
          }
        }
```
```
    //     post("An out Sample: %2.6f\n", *((t_float *)x->outBufs->mBuffers[0].
    mData));

        /* now silence any buffers that were passed in but the that the plugin
         did not fill/touch/use.-- IN MY CASE NOT DOING THAT AS SHOULD ALWAYS
            MATCH
         */

      }

    break;
    }
```
```
// out:
//  post("out %2.6f", *((t_float*) w[3]));
  return (w + 5);
}

// STILL MONO FOR NOW!!
// t_int *auconvert_run_stereo(t_int *w){
//  t_auconvert * x = (t_auconvert *) w[1];
//  auconvert_configureIO(x, 2, 2);
//
```
```
//  if(x->x_obj.z_disabled) goto out;
//
//  //sp[0] for mono, sp[1] have to check out AU workings
//  //need x to get au
//  //sp[2] is output.
//
// out:
//  return (w + 6);
// }
```
```
//————————————————————————— BUFFER HANDLING
// Taken pretty straight from index~
// here's where we set the buffer~ we're going to access
void auconvert_setBuffer(t_auconvert *x, t_symbol *s)
{
  t_buffer *b;

  if (s) {
    x->ac_sym = s;
    if ((b = (t_buffer *)(s->s_thing)) && ob_sym(b) == ps_buffer) {
```
```
      x->ac_buf = b;
    } else {
      object_error((t_object *)x, "no buffer~ %s", s->s_name);
      x->ac_buf = 0;
    }
```

```
    } else {
      // this will reappear every time the dsp is restarted; do we really want it?
      object_error((t_object *)x, "no_buffer~_object_specified");
    }
  }
}
939
  void auconvert_printBufInfo(t_auconvert * x){
    t_buffer *b = x->ac_buf;
    t_symbol *s = x->ac_sym;
    if (s && b) {
      ATOMIC_INCREMENT(&b->b_inuse);
      post("__Buffer_\"%s\"_has_%d_channel(s)_and_%d_frames.__Length_is_%.2f_sec.\
          n", s->s_name, b->b_nchans, b->b_frames, ((double) b->b_frames/ (double)
          x->sampleRate));
      ATOMIC_DECREMENT(&b->b_inuse);
    }
    else {
949   post("Buffer_did_not_open_properly.\n");
    }
  }

  // this lets us double-click on index~ to open up the buffer~ it references
  void auconvert_dblclick(t_auconvert *x)
  {
    t_buffer *b;

    if ((b = (t_buffer *)(x->ac_sym->s_thing)) && ob_sym(b) == ps_buffer)
959     mess0((t_object *)b,gensym("dblclick"));
  }

  //———————————————————————————————— CONSTRUCT & DESTRUCT FUNCTIONS

  void auconvert_assist(t_auconvert *x, void *b, long m, long a, char *s)
  {
    if (m == ASSIST_OUTLET)
      sprintf(s,"(signal)_Sample_Value_at_Index");
969   else {
      switch (a) {
        case 0: sprintf(s,"(signal)_Sample_Index"); break;
        case 1: sprintf(s,"Audio_Channel_In_buffer~"); break;
      }
    }
  }

  void auconvert_newAU(t_auconvert * x){

979   x->initialized = false;
    x->render_ready = 0;
    x->frameCount = 0;
    x->inChannels = 0;
    x->outChannels = 0;

    x->externCount = 0;
    x->startFrame = 1;

    t_int i;
989
    AudioBufferList *inBufList = (AudioBufferList*) sysmem_newptr(sizeof (
        AudioBufferList) + (cMax_Channels - 1) * sizeof(AudioBuffer));
    AudioBufferList *outBufList = (AudioBufferList*) sysmem_newptr(sizeof (
        AudioBufferList) + (cMax_Channels - 1) * sizeof(AudioBuffer));

    for (i = 0; i < cMax_Channels; i++){
      inBufList->mBuffers[i].mNumberChannels = 0;
      inBufList->mBuffers[i].mDataByteSize = 32;
      inBufList->mBuffers[i].mData = sysmem_newptr(sizeof(Sample)*cMax_BufSize);

      outBufList->mBuffers[i].mNumberChannels = 0;
```

153

```
 999        outBufList->mBuffers[i].mDataByteSize = 32;
           outBufList->mBuffers[i].mData = sysmem_newptr(sizeof(Sample)*cMax_BufSize);
       }

       x->inBufs = inBufList;
       x->outBufs = outBufList;

       x->render_framesCompleted = 0;

   //  auconvert_listAUs();
1009   auconvert_getAU(x, "AUTimePitch", kAudioUnitType_FormatConverter);
       auconvert_prepAUwInSamplesCallback(x);
   }

   void auconvert_newBuffer(t_auconvert * x, t_symbol *s){
       x->ac_sym = s;
       x->ac_buf = NULL;
   }

   void auconvert_buildCosTable(long fadeLen){
1019   if (fadeLen > 0){
           // make sure it is even
           if (fadeLen & 1) fadeLen++;
           if (fadeLen > cMaxCosTableSize) cCosTableSize = cMaxCosTableSize;
           cCosTableSize = fadeLen;
       }

       UInt32 i;
       for (i = 0; i < cCosTableSize; i++){
           cosTable[i] =.5*(cos(i*PI/cCosTableSize) + 1);
1029 //      post("Table %d   %.6f\n", i, cosTable[i]);
       }
   }

   void *auconvert_new(t_symbol *s, long chan, long fadeLen)
   {
       t_auconvert *x = (t_auconvert*) object_alloc((t_class*)auconvert_class);
       dsp_setup((t_pxobject *)x,1); //the in signal is the frame count.  pulls
               direct from buffer
       intin((t_object *)x, 3);      // Loop
       floatin((t_object *)x, 2);    // Pitch
1039   floatin((t_object *)x, 1);    // Speed

       x->outlet2_loopSync = bangout((t_object *)x);
       outlet_new((t_pxobject *)x, "signal");

       x->loop = false;
       x->loopSyncFlag = false;
       x->fadeEnable = true;
       x->fadeIn = false;
       x->fadeOut = false;
1049   x->segEnded = false;
       x->userEndTime = 0;
       x->userStartTime = 0;
       x->sampleRate = 44100;   // default

       auconvert_buildCosTable(fadeLen);

       auconvert_newBuffer(x, s);
       auconvert_setNumChannels(x, chan, chan);
       auconvert_newAU(x);
1059
       return (x);
   }

   void *auconvert_free(t_auconvert *x){
       dsp_free((t_pxobject *) x);
       auconvert_stopAU(x);

       // deallocate buffers
```

154

```
          t_int i;
1069      for (i = 0; i < cMax_Channels; i++){
              sysmem_freeptr(x->inBufs->mBuffers[i].mData);
              sysmem_freeptr(x->outBufs->mBuffers[i].mData);
          }
          sysmem_freeptr(x->inBufs);
          sysmem_freeptr(x->outBufs);

          return(x);
      }
```

Appendix E

Audio Extraction Code

E.1 PLCA with Dirichlet Hyperparameters

plca2dwd.m is the core algorithm for extraction. It carries out the PLCA computation using Dirichlet Hyperparameters. Along with *W*, frequency and *H*, temporal posterior estimates derived from the mimic, the main arguments are the number of components in the mimic-*M*, the number of components in the accompaniment-*N*-, number of iterations-k and the degree of sparsity in each dimension. This code is based off original source code from Paris Smaragdis.

```
function [w,h,z, sampMse] = plca( x, K, iter, sz, sw, sh, z, w, h, pl, lw, lh,
    sampFl, m, n, mcp, bcp)
2 % function [w,h,z] = plca( x, K, iter, sz, sw, sh, z, w, h, pl, lw, lh)
%
% Perform 2-D PLCA
%
% If using separate input that initial training, learn all new bases.
%
% Inputs:
%  x       input distribution
%  K       number of components
%  iter    number of EM iterations [default = 100]
12 %  sz     sparsity of z [default = 0]
%  sw      sparsity of w [default = 0]
%  sh      sparsity of h [default = 0]
%  z       initial value of p(z) [default = random]
%  w       initial value of p(w) [default = random]
%  h       initial value of p(h) [default = random]
%  pl      plot flag [default = 1]
%  lw      columns of w to learn [default = 1:K]
%  lh      rows of h to learn [default = 1:K]
%  Remaining Inputs tied to tracking behavior over recursive iterations
22 %  sampFl enables tracking of how MSE changes
%  m       number target components
%  n       number background components (m+n = K)
```

```
%  mcp     target source FFT for comparison
%  bcp     backing source FFT for comparison
%
% Outputs:
%  w    p(w) − vertical bases
%  h    p(h) − horizontal bases
%  z    p(z) − component priors
32 %  sampMse Only returned if sampFl = 1.   Array of listing iterations to
%             target and background MSE
%

% Paris Smaragdis 2006−2008, paris@media.mit.edu

% 30−May−08 − Fixed update of z and problem with lh
% Added a bunch of stuff LSP

   testvals = [];
42 if ~exist('sampFl')
       sampFl = 0;
   else
       sampleIters = 10:10:300;
   end

   % Get sizes
   [M,N] = size( x);

   % Default training iterations
52 if ~exist( 'iter')
     iter = 100;
   end

   % Default plot flag
   if ~exist( 'pl')
     pl = 1;
   end

   % Learn w by default
62 if ~exist( 'lw')
     lw = 1:K;
   end

   % Learn h by default
   if ~exist( 'lh')
     lh = 1:K;
   end

   origw = [w  zeros(M, K−size(w,2))];
72 origh = [h; zeros(K − size(h,1), N)];

   % Initialize
   if ~exist( 'w') || isempty( w)
     w = rand( M, K);

   elseif size( w, 2) ~= K
     w = [w rand( M, K−size( w, 2))];
   end
   w = w ./ repmat( sum( w, 1), M, 1);
82
   if ~exist( 'h') || isempty( h)
     h = rand( K, N);
   elseif size( h, 1) ~= K
     h = [h; rand( K−size( h, 1), N)];
   end
   h = h ./ repmat( sum( h, 2), 1, N);

   if ~exist( 'z') || isempty( z)
     z = rand(1, K);
92   z = z /sum(z);
   end

   % Sort out sparsity parameters
   if ~exist( 'sw', 'var')
```

```matlab
        sw = 0;
      end
      if numel( sw) == 1
        sw = sw*ones( 1, K);
      end
102   if size( sw, 1) == 1
        sw = repmat( sw, iter, 1);
      elseif size( sw, 1) ~= iter
        sw = interp2( sw, linspace( 1, size( sw, 1), iter)', 1:K, 'linear');
      end
      isws = sum( sw(:));

      if ~exist( 'sh', 'var')
        sh = 0;
      end
112   if numel( sh) == 1
        sh = sh*ones( 1, K);
      end
      if size( sh, 1) == 1
        sh = repmat( sh, iter, 1);
      elseif size( sh, 1) ~= iter
        sh = interp2( sh, linspace( 1, size( sh, 1), iter)', 1:K, 'linear');
      end
      ishs = sum( sh(:));

122   %sh
      size(sh);

      if ~exist( 'sz', 'var')
        sz = 0;
      end
      if numel( sz) == 1
        sz = sz*ones( 1, K);
      end
      if size( sz, 1) == 1
132     sz = repmat( sz, iter, 1);
      elseif size( sz, 1) ~= iter
        sz = interp1( sz, linspace( 1, size( sz, 1), iter), 'linear');
      end
      iszs = sum( sz(:));

      multip= linspace(1,0, iter);

    % Iterate
142 tic; lt = toc;
    for it = 1:iter

        % E-step
        zh = diag( z) * h;
        R = x ./ (w*zh);

          dw = origw*multip(it);
          dh = origh*multip(it);
        % M-step
152     if ~isempty( lw)
          nw = w .* (R*zh')+dw;
        end
        if ~isempty( lh)
          nh = zh .* (w'*R)+dh;
        end
        if ~isempty( lw)
          z = sum( nw, 1);
        elseif ~isempty( lh)
          z = sum( nh, 2);
162     end

      % Impose sparsity constraints
      for k = lw
        if isws
              nw(:,k) = lambert_compute_with_offset( nw(:,k), sw(it,k), 0) + eps;
        end
```

159

```
                end
            for k = lh
                if ishs
172             nh(k,:) = lambert_compute_with_offset( nh(k,:), sh(it,k), 0) + eps;
                end
            end
            if iszs
                z = lambert_compute_with_offset( z, sz(it), 0) + eps;
            end

            % Assign and normalize
            if ~isempty( lw )
                w(:,lw) = nw(:,lw)./ (repmat( sum( nw(:,lw), 1), M, 1));
182         end
            if ~isempty( lh )
                h(lh,:) = nh(lh,:) ./ repmat( sum( nh(lh,:), 2), 1, N);
            end
            z = z / sum(z);

            % Show me
            if (toc -lt > 1 || it == iter) && pl
                subplot( 2, 2, 1), imagesc( x/max(x(:))), title( num2str( it))
                subplot( 2, 2, 2), imagesc( w*diag(z)*h)
192             subplot( 2, 3, 4), stem( z), axis tight
                subplot( 2, 3, 5), multiplot( w'), view( [-90 90])
                subplot( 2, 3, 6), multiplot( h)
                drawnow
                lt = toc;
                end

            %trends
            if (sampFl)
                if any(it == sampleIters)
202                 sampMse(it/10, 1) = it;
                    sampMse(it/10, 2) = evalAudio(x, w, h, z, 0, 0, [1 m], mcp)
                    sampMse(it/10, 3) = evalAudio(x, w, h, z, 0, 0, [(m+1) (m+n)], bcp)
                end
            end
        end


%─────────────────────────────────
function thet = lambert_compute_with_offset(omeg, z, lam_offset)
212 % Perform Labert's W iterations
    % fixed-point iterations of eqns 14 and 19 in [1]
    % [1] Brand, M. "Pattern Discovery via Entropy Minimization"
    % Madhu Shashanka, <shashanka@cns.bu.edu>
    % 01 Aug 2006
    % ASSUMPTION: ─────> z is a scalar

    sz = size( omeg);
    omeg = omeg(:)+eps;

222 oz = -omeg/z;
    sgn = sign(-z);

    if z>= 0
        br = -1; % the branch of Lambert Function to evaluate
        lambda = min( z * (log(z) - log(omeg) - 2) - 1) - lam_offset; % initialization
        geval = 'gidx = find( la > -745 );';
    else
        br = 0;
        lambda = - sum(omeg) - min(log(omeg));
232     geval = 'gidx = find( la < 709 );';
    end
    lambda = lambda*ones( size( omeg));
    thet = zeros( size( omeg));
    for lIter = 1:2
        la = log(sgn*oz) + (1+lambda/z);
        eval(geval);
        bidx = setdiff( 1:length(omeg), gidx);
```

160

```
        thet(gidx) = oz(gidx) ./ lambertw_new(br, sgn*exp(la(gidx)) );
        thet(bidx) = oz(bidx) ./ lambert_arg_outof_range( la(bidx) );
242     thet = thet / sum(thet);
        lambda = -omeg./thet - z.*log( thet) - z - lam_offset;
    % lambda = mean(-omeg./thet - z.*log( thet) - z - lam_offset);
    end

    thet = reshape( thet, sz);


    %————————————————————————————————————————
    function w = lambert_arg_outof_range(x)
252 % Computes value of the lambert function W(z)
    % for values of z = -exp(-x) that are outside the range of values
    % of digital floating point representations.
    %
    % Algorithm:
    % Eq (38) and Eq (39), page 23 from
    % Brand, M. "Structure learning in Conditional Probability
    % Models via an entropic prior and parameter extinction", 1998.
    % Available at: http://www.merl.com/reports/docs/TR98-18.pdf
    %
262 % Madhu Shashanka, <shashanka@cns.bu.edu>

    w = x;
    if ~isempty( x)
       while 1
          wo = w;
          w = x - log( abs( w));
          if max( abs( (w-wo)./wo)) < eps
             break
          end
272    end
    end
```

E.2   Sample Song Extraction Code

ExtST.m is provided as a sample of the segmentation and computation for completing an extraction.

```
    targName = 'ST-GuitarMimic.wav';
    mixName = 'ST-FullMix';
    backName = '06_no_Summertime.wav';

    startClock = fix(clock);

  7 %1st Seg
    M = 16;   %want low M
    segLeng = (30.5-11.2)*44100;
    segStart = (11.2*44100);
    [inTargFft inTargWav] = prepAudioSamp(targName, segLeng, segStart);
    [inMixFft inMixWav] = prepAudioSamp(mixName, segLeng, segStart);
    [inBacFft inBacWav] = prepAudioSamp(backName, segLeng, segStart);
    inTargFft = filterTF(inTargFft);
    runTests;
    set1  = mnkLAudM;
 17 set1ws = swMelSirAud;
    set1hs = shMelSirAud;

    segStart = segStart +segLeng;

    %2nd Seg
    M = 20;   %want low M
    segLeng = (50-30.5)* 44100;
    [inTargFft inTargWav] = prepAudioSamp(targName, segLeng, segStart);
```

```matlab
        [ inMixFft inMixWav ] = prepAudioSamp ( mixName , segLeng , segStart ) ;
27    [ inBacFft inBacWav ] = prepAudioSamp ( backName , segLeng , segStart ) ;
        inTargFft = filterTF ( inTargFft ) ;
        runTests ;
        set2 = mnkLAudM ;
        set2ws = swMelSirAud ;
        set2hs = shMelSirAud ;

        segStart = segStart + segLeng ;

        finishClock = fix ( clock ) ;
```

E.3   Basic Test Set

runTests.m includes full suite of tests to run on an audio segment. End result is the optimal find-

ing for each set of parameters. User auditions to pick preferred extraction and then reassemble.

Currently commented for quickest execution.

```matlab
    M = 20;      N = 100;
    %[ tmse  tw  th  tz ] = runBasics ( inTargFft , M ) ;
    %disp ( 'First Analysis Complete.   Checking Performance per PLCA iteration ' )
 4  %[w h z iterMse ] = plca2dwd ( abs ( inMixFft ) , 180 , 300 , 0 , 0 , 0 , [] , tw , th , 0 , [ 1
         180 ] , [ 1 180 ] , 1 , 30 , 150 , inTargFft , inBacFft ) ;
    disp ( 'Done.   Checking out what happens as M varies ' ) ;
    [ mkInvMse kldiv ] = mkInvest ( inTargFft , 50 , 1 ) ;
    if ~ exist ( 'inGTWav ' )
    %    disp ( 'Done.   Now running M vs. M over wide range of values ' ) ;
    %    [ mnkBMse tra trw trh trz trk mnkPerM mnkAudM ] = kInvestLite ( inTargFft ,
         inMixFft , inBacFft , 1 , 0 , 0 , inTargWav , inBacWav , 30 , 0 ) ;
        disp ( 'Done.   Now running N vs. M over narrow range of values ' ) ;
        [ mnkLMse tra trw trh trz trk mnkLPerM mnkLAudM ] = kInvestLite ( inTargFft ,
             inMixFft , inBacFft , 0 , 18 , 60 , inTargWav , inBacWav , 30 , 0 ) ;

        disp ( 'Done.   Now running Sparsity tests . ' )
14      disp ( 'Start with W ' ) ;
    %    [ swkmse trw trh trz swMseSpar swMseAud swMelSir swMelSirSpar swMelSirAud
         swBacSir swBacSirSpar swBacSirAud ] = sparseInvestLite ( inTargFft , inMixFft ,
         inBacFft , M , N , 'w' , inTargWav , inBacWav , 30 ) ;
        disp ( 'Now H ' ) ;
    %    [ shkmse trw trh trz shMseSpar shMseAud shMelSir shMelSirSpar shMelSirAud
         shBacSir shBacSirSpar shBacSirAud ] ...
    %                   = sparseInvestLite ( inTargFft , inMixFft , inBacFft , M , N , 'h
         ', inTargWav , inBacWav , 30 ) ;
    %    disp ( 'Now Sparisty on Z ' ) ;
    %    [ szkmse trw trh trz szMseSpar szMseAud szMelSir szMelSirSpar szMelSirAud
         szBacSir szBacSirSpar szBacSirAud ] ...
    %                   = sparseInvestLite ( inTargFft , inMixFft , inBacFft , M , N , 'z
         ', inTargWav , inBacWav , 30 ) ;
    %    disp ( 'Now Sparisty on wh ' ) ;
    %    [ swhkmse trw trh trz swhMseSpar swhMseAud swhMelSir swhMelSirSpar
         swhMelSirAud swhBacSir swhBacSirSpar swhBacSirAud ] ...
24  %                   = sparseInvestLite ( inTargFft , inMixFft , inBacFft , M , N , 'wh
         ', inTargWav , inBacWav , 30 ) ;
    %    disp ( 'Now Sparisty on whz ' ) ;
    %    [ swhzkmse trw trh trz swhzMseSpar swhzMseAud swhzMelSir swhzMelSirSpar
         swhzMelSirAud swhzBacSir swhzBacSirSpar swhzBacSirAud ] ...
    %                   = sparseInvestLite ( inTargFft , inMixFft , inBacFft , M , N , '
         whz ', inTargWav , inBacWav , 30 ) ;
```

```matlab
        disp('Whew!_Sparsity_is_done._Last_thing_is_Bayes_masking!');
%         [w h z] = plca2dwd(abs(inMixFft), M+N, 100, 0, 0, 0, [], tw, th, 0);
%         [bsdr bisr bsir bsar] = bayesInvest(w, h, z, M, inMixFft, inTargWav,
    inBacWav);
        disp('Totally_Done!');
    else
        disp('Have_Ground_Truth_for_Comparison');
34      disp('Done._Now_running_M_vs._M_over_wide_range_of_values');
        [mnkBMse tra trw trh trz trk mnkPerM mnkAudM] = kInvestLite(inTargFft,
            inMixFft, inBacFft, 1, 0, 0, inGTWav, inBacWav, 30, 0);
        disp('Done._Now_running_N_vs._M_over_narrow_range_of_values');
        [mnkLMse tra trw trh trz trk mnkLPerM mnkLAudM] = kInvestLite(inTargFft,
            inMixFft, inBacFft, 0, 20, 30, inGTWav, inBacWav, 30, 0);
        N = 100;
        disp('Done._Now_running_Sparsity_tests.')
        disp('Start_with_W');
        [swkmse trw trh trz swMseSpar swMseAud swMelSir swMelSirSpar swMelSirAud
            swBacSir swBacSirSpar swBacSirAud] = sparseInvestLite(inTargFft,
            inMixFft, inBacFft, M, N, 'w', inGTWav, inBacWav, 30);
        disp('Now_H');
        [shkmse trw trh trz shMseSpar shMseAud shMelSir shMelSirSpar shMelSirAud
            shBacSir shBacSirSpar shBacSirAud] ...
44                      = sparseInvestLite(inTargFft, inMixFft, inBacFft, M, N, 'h',
                            inGTWav, inBacWav, 30);
        disp('Now_Sparisty_on_Z');
        [szkmse trw trh trz szMseSpar szMseAud szMelSir szMelSirSpar szMelSirAud
            szBacSir szBacSirSpar szBacSirAud] ...
                            = sparseInvestLite(inTargFft, inMixFft, inBacFft, M, N, 'z',
                                inGTWav, inBacWav, 30);
        disp('Now_Sparisty_on_wh');
        [swhkmse trw trh trz swhMseSpar swhMseAud swhMelSir swhMelSirSpar
            swhMelSirAud swhBacSir swhBacSirSpar swhBacSirAud] ...
                            = sparseInvestLite(inTargFft, inMixFft, inBacFft, M, N, 'wh'
                                , inGTWav, inBacWav, 30);
        disp('Now_Sparisty_on_whz');
        [swhzkmse trw trh trz swhzMseSpar swhzMseAud swhzMelSir swhzMelSirSpar
            swhzMelSirAud swhzBacSir swhzBacSirSpar swhzBacSirAud] ...
                            = sparseInvestLite(inTargFft, inMixFft, inBacFft, M, N, 'whz
                                ', inGTWav, inBacWav, 30);
54      disp('Whew!_Sparsity_is_done._Last_thing_is_Bayes_masking!');
        [w h z] = plca2dwd(abs(inMixFft), M+N, 100, 0, 0, 0, [], tw, th, 0);
        [bsdr bisr bsir bsar] = bayesInvest(w, h, z, M, inMixFft, inGTWav, inBacWav)
            ;
        disp('Totally_Done!');
    end
```

E.4 Sample Parameter Test

kInvest.m provides an example of testing on various parameters. textitkInvest.m will test for lowest

MSE and SIR for a variety of combinations of different component numbers, *M* and *N*.

```matlab
function [kmse kldiv bestw besth bestz bestk bestPerM, bestAudM] = kInvest(
    targFft, compleFft, bacFft, expTk, melWav, bacWav, plcaIters, doplot)
2 %Optimize for best first approximation− we know sparsity will not be
  %particularly helpful at this stage.

    if ~exist('plcaIters')
        plcaIters = 100;
    end
```

```matlab
        if ~exist('doplot')
            doplot = 0;
        end
12
        bestTw = []; bestTh =[]; bestTz =[];

        kmse = [];
        bestmse = inf;
        ind = 0;
        mi = 0; %tK based index
        bestAudM = [];

        minTk = expTk/2.5;
22      maxTk = expTk*2.5;
   %     minTk = expTk/2;
   %     maxTk = expTk*2;

        tKsteps = round(linspace(minTk, maxTk, 8));
        bKsteps = 175:25:250;
   %     bKsteps = 20:20:80;
        for tk = tKsteps
            tk
            mi = mi+1;
32          bestThisk = inf;
            for iter = 1:4
                [tw th tz] = plca2d(abs(targFft), tk, plcaIters, 0, 0, 0, [], [],
                    [], 0);
                [tmse kldiv] = evalAudio(targFft, tw, th, tz);
                if (tmse < bestThisk)
                    bestThisk = tmse;
                    besttw = tw;
                    bestth = th;
                    besttz = tz;
                end
42          end;

            bestOthisTk = inf;
            for bk = bKsteps
                bk
                ind = ind +1;
                kmse(ind, 1) = tk;
                kmse(ind, 2) = bk;
                for iter = 1:3
52                  [bw bh bz] = plca2dwd(abs(compleFft), tk+bk, plcaIters, 0, 0, 0,
                        [], besttw, bestth, 0);
                    [mse kldiv sdr isr sir sar wavs] = evalAudioWbss(compleFft, bw,
                        bh, bz, tk, melWav, targFft, bacWav, bacFft);
                    kmse(ind, 8*iter -3) = mse(1);
                    kmse(ind, 8*iter -2) = mse(2);
                    kmse(ind, 8*iter -1) = sdr(1);
                    kmse(ind, 8*iter) = sdr(2);
                    kmse(ind, 8*iter+1) = sir(1);
                    kmse(ind, 8*iter+2) = sir(2);
                    kmse(ind, 8*iter+3) = sar(1);
                    kmse(ind, 8*iter+4) = sar(2);
62
                    if (mse(2) < bestmse)
                        bestmse = mse(2);
                        bestw = bw;
                        besth = bh;
                        bestz = bz;
                        bestk(1) = tk;
                        bestk(2) = bk;
                    end;

72                  if (mse(2) < bestOthisTk)
                        bestPerM(mi) = bk;
                        bestKwavs = wavs;
                    end
```

```matlab
                end ;
            end ;
            bestAudM = [bestAudM bestKwavs ] ;
        end ;

        if doplot
82          figure
            subplot(321);
            hold on ;
            for iter = 1:3
                scatter(tKsteps, kmse(:, 8*iter-3));
            end ;
            xlabel('k');
            ylabel('ave_mse');
            title('Target_Error_vs_Components');

92          subplot(322);
            for iter = 1:3
                scatter(bKsteps, kmse(:, 8*iter-2));
            end ;
            xlabel('k');
            ylabel('ave_mse');
            title('Backing_Error_vs_Components');

            subplot(323);
            for iter = 1:3
102             scatter(tKsteps, kmse(:, 8*iter-1));
            end ;
            xlabel('k');
            ylabel('SDR');
            title('Targ_SDR');

            subplot(324);
            for iter = 1:3
                scatter(bKsteps, kmse(:, 8*iter));
            end ;
112         xlabel('k');
            ylabel('SDR');
            title('Backing_SDR');

            subplot(325);
            for iter = 1:3
                scatter(bKsteps, kmse(:, 8*iter+1));
            end ;
            xlabel('k');
            ylabel('SIR');
122         title('Target_SIR');

            subplot(326);
            for iter = 1:3
                scatter(bKsteps, kmse(:, 8*iter+2));
            end ;
            xlabel('k');
            ylabel('SIR');
            title('Backing_SIR');

132         hold off ;
        end ;

    end
```

Bibliography

[1] http://breakfastquay.com/rubberband/.

[2] http://cycling74.com/.

[3] http://developer.apple.com/library/ios/#documentation/audiounit/reference/audiounit_framework/_index.html.

[4] http://developer.echonest.com/docs/v4/.

[5] http://hypermammut.sourceforge.net/paulstretch/.

[6] http://mpex.prosoniq.com/.

[7] http://usa.yamaha.com/products/music-production/midi-controllers/wx5/?mode=model.

[8] http://www-acad.sheridanc.on.ca/d̃egazio/aboutmefolder/musicpages/ewi-wx5.html.

[9] http://www.ableton.com/.

[10] http://www.akaipro.com/ewi4000s/.

[11] http://www.celemony.com/cms/.

[12] http://www.digitaltrumpet.com.au/.

[13] http://www.dspdimension.com/technology-licensing/dirac/.

[14] http://www.elasticmax.co.uk/.

[15] http://www.fender.com/rockband3/.

[16] http://www.guitarhero.com/.

[17] http://www.iamas.ac.jp/˜ aka/max/#aka_wiiremote.

[18] http://www.keithmcmillen.com/.

[19] http://www.nintendo.com/wii/console/.

[20] http://www.rockband.com/.

[21] http://www.surina.net/soundtouch/soundstretch.html.

[22] http://www.virtualdub.org/blog/pivot/entry.php?id=102.

[23] C. Avendano. Frequency-domain source identification and manipulation in stereo mixes for enhancement, suppression and re-panning applications. In *Applications of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop on.*, pages 55–58. IEEE, 2004.

[24] A. Ben-Shalom and S. Dubnov. Optimal filtering of an instrument sound in a mixed recording given approximate pitch prior. In *Proceedings of the International Computer Music Conference*, 2004.

[25] S.M. Bernsee. Pitch shifting using the fourier transform, 1999.

[26] S.M. Bernsee. Time stretching and pitch shifting of audio signals. *The DSP Dimension*, 2003.

[27] D. Blum. *Casals and the Art of Interpretation*. Univ of California Pr, 1980.

[28] Jan O. Borchers, Wolfgang Samminger, and Max Mühlhäuser. Conducting a realistic electronic orchestra. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 161–162, New York, NY, USA, 2001. ACM.

[29] R. Boulanger, M. Mathews, B. Vercoe, and R. Dannenberg. Conducting the MIDI Orchestra, Part 1: Interviews with Max Mathews, Barry Vercoe, and Roger Dannenberg. *Computer Music Journal*, 14(2):34–46, 1990.

[30] J.C. Brown and M.S. Puckette. A high resolution fundamental frequency determination based on phase changes of the fourier transform. *JOURNAL-ACOUSTICAL SOCIETY OF AMERICA*, 94:662–662, 1993.

[31] George Cochrane. "Entertaining Circumstances', 2010.

[32] Arshia Cont. *Modeling musical anticipation: from the time of music to the music of time*. PhD thesis, University of California at San Diego, La Jolla, CA, USA, 2008. Adviser-Dubnov, Shlomo.

[33] J. Eggink and G.J. Brown. Application of missing feature theory to the recognition of musical instruments in polyphonic audio. In *Proc. ISMIR*, pages 125–131. Citeseer, 2003.

[34] C. Févotte, R. Gribonval, and E. Vincent. Bss_eval toolbox user guide–revision 2.0. 2005.

[35] C. Févotte, R. Gribonval, and E. Vincent. A toolbox for performance measurement in (blind) source separation, 2008.

[36] Noel Gallagher. "Wonderwall". In *(What's the Story) Morning Glory?* Creation, 1995. Oasis.

[37] George Gershwin. "Summer Time". In *Jazz Standards*. Music Minus One, 1998. performed by Larry Linkin.

[38] T. Inoue and S. Sugishita. Compression/expansion method of time-scale of sound signal, July 14 1998. US Patent 5,781,885.

[39] T. Jehan and B. Schoner. An audio-driven perceptually meaningful timbre synthesizer. *Analysis*, 2(3):4, 2002.

[40] Antonio Carlos Jobim. "One Note Samba". In *Jazz Standards*. Music Minus One, 1998. Performed by Larry Linkin.

[41] C.M. Johnson. The performance of mozart: Study of rhythmic timing by skilled musicians. *Psychomusicology: Music, Mind and Brain*, 15(1-2):87–109, 2011.

[42] T.W. Lee, M.S. Lewicki, M. Girolami, and T.J. Sejnowski. Blind source separation of more sources than mixtures using overcomplete representations. *Signal Processing Letters, IEEE*, 6(4):87–90, 1999.

[43] J. London. Musical rhythm: Motion, pace and gesture. *Music and gesture*, pages 126–141, 2006.

[44] R.C. Maher. An approach for the separation of voices in composite musical signals. Technical report, 1989.

[45] D. Malah. Time-domain algorithms for harmonic bandwidth reduction and time scaling of speech signals. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 27(2):121–133, 1979.

[46] Stephen Malinowski. http://www.musanim.com/tapper/conductorprogram.html.

[47] Max V. Mathews and John R. Pierce, editors. *Current directions in computer music research.* MIT Press, Cambridge, MA, USA, 1989.

[48] M.V. Mathews. The radio baton and conductor program, or: Pitch, the most important and least expressive part of music. *Computer Music Journal*, 15(4):37–46, 1991.

[49] J.S. Plank. Computer assisted performance using midi-based electronic musical instruments. Technical report, University of Tennessee Knoxville, TN, USA, 1999.

[50] M.D. Plumbley, SA Abdallah, JP Bello, ME Davies, G. Monti, and M.B. Sandler. Automatic music transcription and audio source separation. *Cybernetics &Systems*, 33(6):603–627, 2002.

[51] M.S. Puckette, M.S.P. Ucsd, T. Apel, et al. Real-time audio analysis tools for pd and msp. *Proceedings of International Computer Music Conference (ICMC)*, 1998.

[52] C. Raphael. A classifier-based approach to score-guided source separation of musical audio. *Computer Music Journal*, 32(1):51–59, 2008.

[53] Andrew Robertson and Mark Plumbley. B-keeper: a beat-tracker for live performance. In *NIME '07: Proceedings of the 7th international conference on New interfaces for musical expression*, pages 234–237, New York, NY, USA, 2007. ACM.

[54] MR Schroeder, JL Flanagan, and EA Lundry. Bandwidth compression of speech by analytic-signal rooting. *Proceedings of the IEEE*, 55(3):396–401, 1967.

[55] P. Smaragdis and J.C. Brown. Non-negative matrix factorization for polyphonic music transcription. In *Applications of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop on.*, pages 177–180. IEEE, 2003.

[56] P. Smaragdis and G.J. Mysore. Separation by humming: User-guided sound extraction from monophonic mixtures. In *Applications of Signal Processing to Audio and Acoustics, 2009. WASPAA'09. IEEE Workshop on*, pages 69–72. IEEE, 2009.

[57] P. Smaragdis and B. Raj. Shift-invariant probabilistic latent component analysis. *Journal of Machine Learning Research*, 2008.

[58] N.A. Steiner. The electronic valve instrument (evi), an electronic musical wind controller for playing synthesizers. *The Journal of the Acoustical Society of America*, 115:2451, 2004.

[59] C.E. Strangio. Computer-aided musical apparatus and method, May 10 1977. US Patent 4,022,097.

[60] traditional. "Greensleeves". In *Live! At the Village Vanguard*. Impulse! Records, 1961. Performed by John Coltrane.

[61] Barry Vercoe and Miller Puckette. Synthetic rehearsal: Training the synthetic performer. *Proceedings of International Computer Music Conference (ICMC)*, 1985.

[62] Thom Yorke. "Nude". In *In Rainbows*. XL Recordings, 2007. Radiohead.

[63] D. Zicarelli. M and jam factory. *Computer Music Journal*, 11(4):13–29, 1987.

[64] Thomas G. Zimmerman, & Samuel P. Wantman. Electronic musical re-performance and editing system, January 1996.